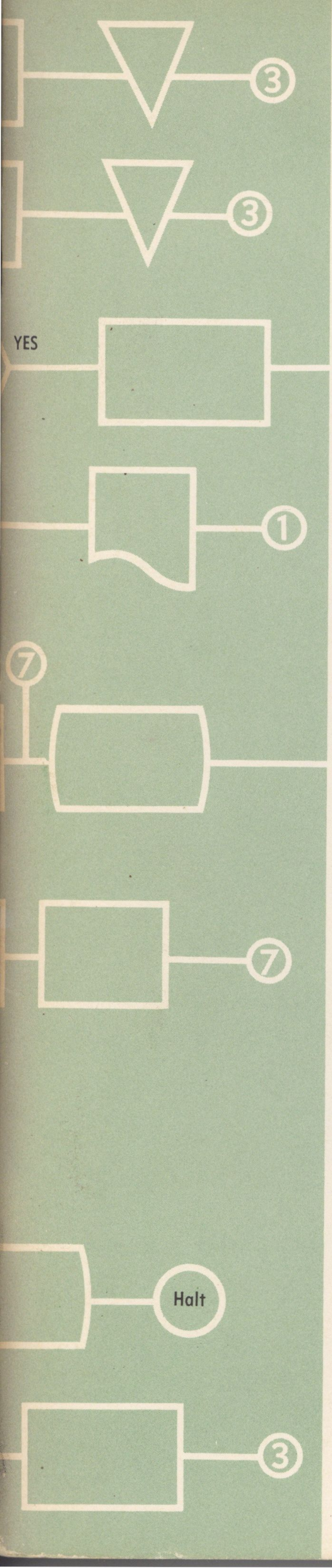


AN INTRODUCTION TO CODING THE BURROUGHS 220

ELECTRONIC DATA PROCESSING SYSTEMS

Bulletin 5019
December, 1958

AN INTRODUCTION TO CODING THE BURROUGHS 220

ELECTRONIC DATA PROCESSING SYSTEMS

Burroughs Corporation
ELECTRODATA DIVISION
PASADENA, CALIFORNIA

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

APPENDICES

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Cont)

1

Introduction

This volume in the Burroughs Electronic Data Processing Library is intended to serve as a textbook in introductory coding courses for the Burroughs 220 Electronic Data Processing System. It was written expressly to introduce the novice to a many faceted art, the art of conversing with electronic computing equipment of the so-called stored-program type.

For our purposes it is desirable to distinguish between a coder and a programmer in the following way:

> A *programmer* is the analyst who states a proposed solution to a problem in any language which is convenient.
>
> A *coder* converts this statement of a proposed solution to a problem to a language which is meaningful to the computing system assigned to solve the problem.

Although it often happens that the programmer and the coder are the same person, the analytical aspects of the communications problem will be mentioned only rather briefly in any of the short courses conducted by representatives of the Burroughs Corporation. This is not to deny the importance of programming; rather, it is to emphasize the need for critical and competent analysis: implicit in our definition of analysis is the requirement for defining the problem to be solved. It is clear that *some* problem is defined by the statement of a proposed solution. It is the programmer's responsibility to ensure that it is *the* problem which will be solved.

The preceding paragraph is intended to emphasize what this book is not—what it makes no pretense of being. If the coder is to be analyst as well, textbooks for his education in that specialty must be sought elsewhere. In this book there will be found only a brief discussion to indicate the nature and magnitude of the programmer's task.

Preparing lists of instructions for the computing system which will solve the problem specified by the programmer is the responsibility of the coder. It is the purpose of this book to introduce the novice coder to that aspect of the art which is particularly concerned with instructions to the machine. The emphasis is on "introduction" and "art": this textbook and its related publications—for example, Operational Characteristics of the Burroughs 220 and The Compleat Programmer—are intended to provide the background from which professional skills can be developed. Such skills can be fully developed, however, only by practice and application. Usually this occurs on the job.

The classroom environment provides training in the natural language of the Burroughs 220. At the same time training can also be provided in one or more of the languages which the 220 has been compelled to learn in order to make it easier for human beings to communicate with it. These imposed languages are the so-called automatic coding or automatic programming facilities which have been devised by Burroughs and other users of the equipment.

In this book an introduction to the notions of automatic coding—that is, the preparation of instruction lists with the help of the computer—is to be found in Chapter 13. It should be noted here that there are several different classes of such coding aids. Some of them are designed for use by the occasional coder—an engineer, say, who relatively infrequently wants to prepare problems for solution. Some simplify the regular job of the professional coder.

Finally, something needs to be said about the presentation of information on the Cardatron and Magnetic Tape Systems. Each is relatively extensive and complex, requiring that a substantial amount of information be assimilated before it can be used as it would be in "real life." The quantity of information required is approximately that contained in the relevant sections of Operational Characteristics of the Burroughs 220. Because that book will be used as a supplementary text in introductory courses, it was decided not to reproduce the sections on Cardatron and magnetic tape here. Instead, Chapter 11 describes the Cardatron System briefly and includes some sample instructions. Chapter 12 does the same for magnetic tape.

The authors and the publishers would appreciate constructive criticism of the contents of this book whether it relates to errors in fact or to the manner of presentation. Such remarks or requests for further information should be addressed to:

> Manager, Publications & Training
> ElectroData Division
> Burroughs Corporation
> 460 Sierra Madre Villa Avenue
> Pasadena, California

2 A Digital Computer System

GENERAL

Electronic digital computers are now established as useful —and sometimes indispensable—aids to organizations engaged in a wide range of business and scientific pursuits.

One of the most impressive characteristics of digital computers is their operating speed. In discussing computer speeds, it is necessary to use words for divisions of a second: millisecond, for a thousandth of a second, and microsecond, for a millionth of a second.

It takes a desk-size computer, for example, 50 milliseconds to add two numbers together. But a computer such as the Burroughs 220 takes only about 200 microseconds (or 0.2 milliseconds) to add the same two numbers. During the time it takes to read these two paragraphs, the Burroughs 220 could sum about 150,000 such numbers. This is the order of speed to be considered during a detailed discussion of a computing system.

Although these computing systems are complex—both in construction and operation—the difficulties of understanding them are similar to those encountered when one approaches any unfamiliar subject. Much of their complexity can be reduced to a combination of simple principles. And these principles can be illustrated by familiar things and ideas.

Let us take a look at these automatic, electronic, data-processing systems to see how the elements of such a system and the role played by each element in the system can be illustrated by an analogy.

Suppose that the supervisor of a payroll department asks a new clerk to figure out how much each person in the office has earned for the past week. The clerk is given the time card and earnings record card for each employee, a desk calculator, a pencil, a typewriter, and a list of instructions telling him what to do.

We will discuss this problem in its simplest form: we will consider the preparation of a paycheck for a single employee. The clerk needs to know hours worked, rate of pay, and what deductions—such as withholding tax—are to be subtracted from the employee's gross pay. We will assume that the employee works exactly 40 hours and thus is not entitled to overtime pay. Also, we will concentrate on the computing portion of the job.

Before he starts the job, the clerk copies the date and the hours worked from the employee's time card into specified columns on the employee's earnings record card (Fig. 2-1). He then copies the hourly pay and the rate of deductions in percentage form from the previous week. (For our problem, we assume there have been no recent changes.)

The list of step-by-step instructions which the clerk uses reads as follows:

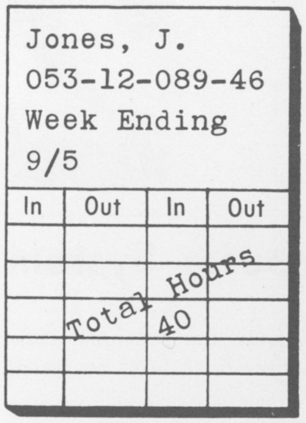
1. Multiply hours by rate to get gross pay.
2. Record gross pay in the column specified.
3. Multiply gross pay by rate of deductions to get the amount.
4. Record the dollar amount of deductions in the column specified.
5. Subtract the dollar amount of deductions from gross pay to get net pay.
6. Record net pay in the column specified.

These instructions could be stated more concisely by taking advantage of the format of the earnings record card used by the clerk as a work sheet. Since the columns are numbered, why not abbreviate the instructions by using column numbers? If this were done, the instructions would look like this:

1. Multiply the number in column 1 by the number in column 2.
2. Record the product in column 4.
3. Multiply the number in column 4 by the number in column 3.
4. Record this product in column 5.
5. Subtract the number in column 5 from the number in column 4.
6. Record the answer in column 6.

These instructions could be stated even more concisely if we used the column numbers to represent the information recorded in the columns. If we do this, however, we must be sure to keep in mind that each column number is only a label that stands for the information recorded in that column. Thus when we write the number 1 we mean the contents of column 1, 2 refers to the contents of column 2, etc., and we mean the numbers in columns 1 or 2 opposite the current date. We can make one more assumption for further simplification: since everyone is familiar with the arithmetic symbols $\times$ and $-$, these symbols can be substituted for the words they represent. If the word "column" is left out, the instruction sheet would now read:

1. $1 \times 2 = 4$
2. $3 \times 4 = 5$
3. $4 - 5 = 6$

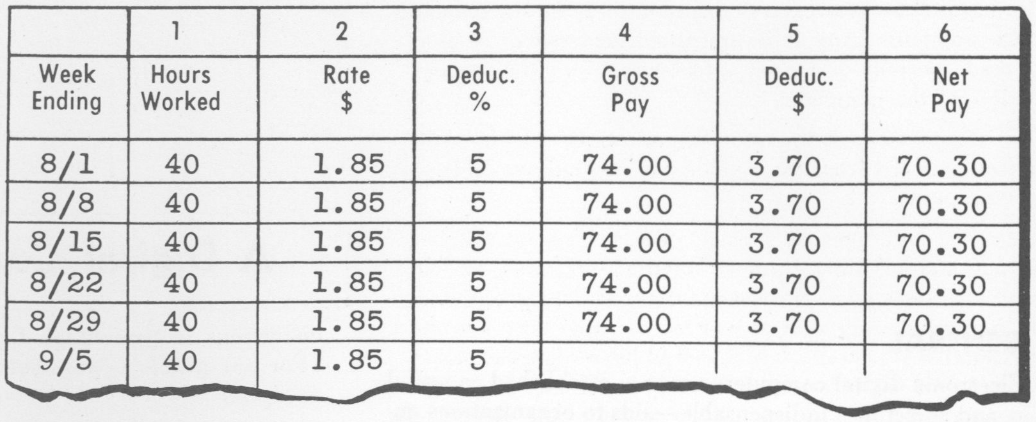


Jones, J.
053-12-089-46
Week Ending
9/5

In	Out	In	Out
		40	

Total Hours

Time Card



	1	2	3	4	5	6
Week Ending	Hours Worked	Rate $	Deduc. %	Gross Pay	Deduc. $	Net Pay
8/1	40	1.85	5	74.00	3.70	70.30
8/8	40	1.85	5	74.00	3.70	70.30
8/15	40	1.85	5	74.00	3.70	70.30
8/22	40	1.85	5	74.00	3.70	70.30
8/29	40	1.85	5	74.00	3.70	70.30
9/5	40	1.85	5			

Earnings Record Card

Figure 2-1. Payroll Job Preparation.

These condensed instructions can now be recorded on the earnings record work sheet (Fig. 2-2).

The clerk is now ready to begin. He is directed to start with the first instruction. He looks at this instruction to determine the two numbers to be used (operands) and the operation to be performed on them.

To execute the instruction, the clerk uses the desk calculator. First, he enters the operand from column 1 into the keyboard of the desk calculator. When the "multiply" motor bar is depressed, the two numbers are multiplied and the product is displayed. He copies the product into column 4, the column specified by the instruction.

The clerk is directed to execute the instructions in sequence. After executing instruction 1, he interprets and executes instructions 2 and 3 in the same way, recording the results on the work sheet.

After executing all three instructions, the clerk types the completed payroll check from the information on the work sheet.

BASIC ELEMENTS OF A COMPUTER SYSTEM

Let us review the elements of our analogy and relate them to the elements of a data processing system.

The outside data from the time card is taken into the calculation process when the clerk reads it and records it on the earnings record card. This process can be called "input."

During the calculation process, the information is held ready and available in the clerk's mind and on the work sheet. This can be called "storage."

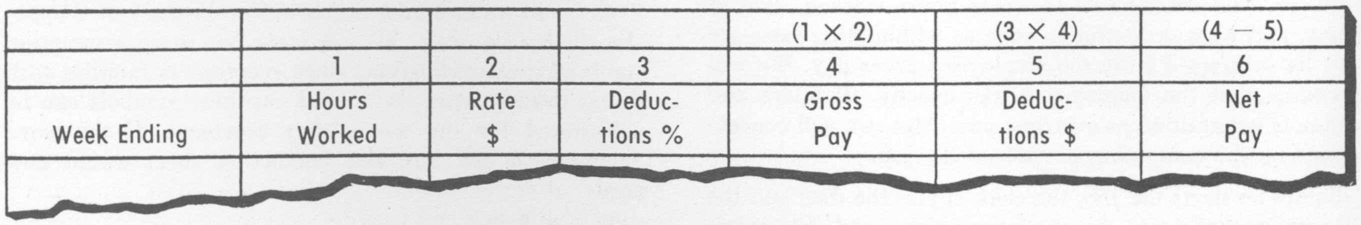
The data from the work sheet is processed on the calculator according to a prescribed procedure. The calculator can be called the arithmetic element.

The flow and processing of the information proceeds in an ordered manner under the direction of the clerk. He can be called the control element.

The processed results are copied from the work sheet onto the paycheck and returned to the person who delegated the clerk to perform the operation. This process can be called "output."

Input, storage, arithmetic, control, and output—these are the five basic elements of a computer system. Let us take a closer look at each of these from the standpoint of their place in a computing system (Fig. 2-3).

The dotted lines enclose the three elements of the system which comprise what is commonly known as the computer. These three elements together form the processing center of the system where the actual compilation, computation, and manipulation of data takes place. The control unit directs the flow of information between storage—where it is held—and the arithmetic unit, where it is processed.



				(1 × 2)	(3 × 4)	(4 − 5)
	1	2	3	4	5	6
Week Ending	Hours Worked	Rate $	Deduc-tions %	Gross Pay	Deduc-tions $	Net Pay

Figure 2-2. Earnings Record Card with Instructions.

The input element transmits information from the outside to the computer. Any of several input devices can be used to read specially prepared information into the storage section of the computer.

The storage element provides the means by which information received from the outside can be held available to the computer.

There are two types of storage: internal—or working—storage and auxiliary storage. Internal storage is used to store all information necessary for immediate processing. An internal storage unit must provide rapid access to the information it contains and it should have a large capacity —to contain all information needed for immediate problem solution.

Because there are technical and economic limits to the size of high-speed internal storage units, information not currently needed is often stored in larger but slower auxiliary devices. Auxiliary storage is almost always included in a large data processing system.

The arithmetic element is the computing portion of the system. It is here that the actual work of problem solving is done. The operations of addition, subtraction, multiplication, and division are performed by this unit. No matter how complicated a mathematical problem may be, it can be broken down to these four basic operations.

This unit also provides means for comparing one number to another to determine whether they are equal or which is the greater. The result of such a comparison allows the computer to choose among several specified series of further operations.

The control element does just what its name implies—it controls the operation of the computer during the complete process of problem solution. This element directs the sequence of operation, interprets the operations to be performed, initiates the action which performs the operations, and activates the input and output devices.

The output element transmits processed results from the computer to the outside. Output devices accept information from computer storage and reproduce it in a convenient form for normal use.

Now let us refer to parts of our analogy to illustrate some of the concepts of computer operation and computer usage.

CODING

A computer, like the clerk in our analogy, must be told what to do. It does only what it is told to do—nothing more, nothing less.

There is, however, a problem of communication with a computer, much like the language barrier between two persons who speak different languages. The language of a person who wishes to communicate with a computer must first be translated into the language of the computer. The language of the problems to be solved is mathematics, or English statements of decisions to be made; the language of computers is simple arithmetic and elementary choices.

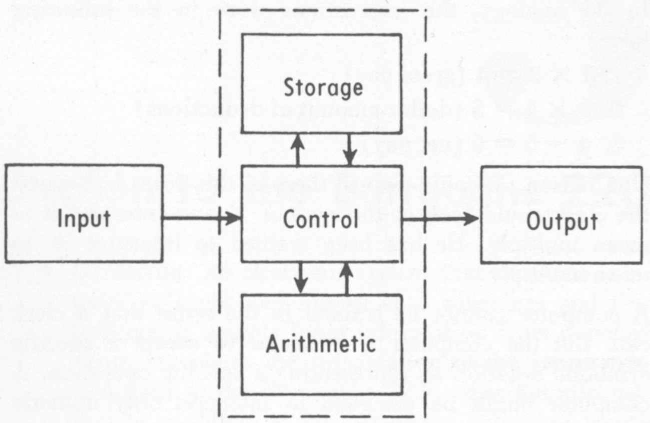


Figure 2-3. Elements of a Computer System.

There are two steps in the process of translating human language to the language of a computer.

The first is illustrated in the three forms assumed by the instructions in our analogy: the problem in written statement form was analyzed in terms of simple arithmetic. Any problem in mathematics or written statement form must be analyzed in terms of the basic operations comprising the computer language.

Secondly, the user must consider how the computer can be told to solve his analyzed problem with the limited vocabulary available: most computers have a vocabulary of from 20 to 70 instructions. This part of the translation —putting the problem to be solved into words the computer can understand—is called coding.

Coding a computer application of some length and detail for problem solution is not a simple matter. Consider, for example, trying to translate an extensive work, such as the Bible, into a language with a vocabulary of only 800 words. It might take several paragraphs to explain one word not included in the vocabulary. A computer vocabulary does not include such phrases as "calculate net pay" or "evaluate function." Therefore, a coder must build such an operation using only the words—called instructions—that have meaning for the computer.

The end result of building the operation is a list of instructions called a code or program—an orderly explanation to the computer of each individual operation it is to perform. A computer code is similar in form and purpose to the step-by-step list of instructions given to the clerk in our analogy.

The list of instructions comprising a computer vocabulary usually includes the four basic arithmetic operations, operations providing the ability to make elementary choices, operations governing the transfer of data between sections of the computer, operations governing input and output equipment, and various other more complex operations which are used frequently enough to justify their inclusion.

FORMS OF INSTRUCTIONS

In the analogy, the instructions were in the following form:

1. 1 × 2 = 4 (gross pay)
2. 3 × 4 = 5 (dollar amount of deductions)
3. 4 − 5 = 6 (net pay)

One reason we could reduce them to this form is because the clerk could look at the symbol × and interpret it to mean multiply. He has been trained to interpret × to mean multiply.

A computer cannot be trained in the sense that a clerk can. But the computer is designed to accept a specific symbolic notation as representing a specific operation. A computer might be designed to interpret only numeric forms. In this case, each instruction in the computer vocabulary would be assigned a numeric notation. These numeric notations would be used when preparing instructions for input to the computer. For example, the operation multiply could be assigned the numeric notation 14. The computer would automatically interpret this notation as meaning to multiply.

WORD CONCEPT

Digital computers handle information in units consisting of a fixed number of digits—a length of 10 to 12 decimal digits is most frequently used. Since these units may represent not only numeric quantities but also coded numeric instructions and alphabetic data, it is convenient to refer to them as "words." This provides a common term for all types of information handled by a computer.

STORED PROGRAM

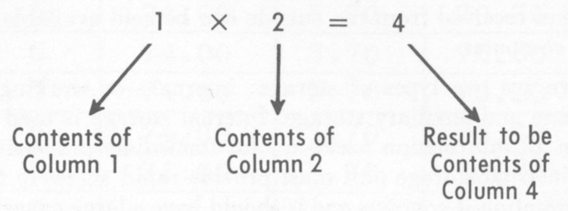
In the analogy, the condensed instructions were recorded on the earnings record work sheet with the data. Although the instructions were recorded in the same manner as the data, they were interpreted as instructions by the clerk who controlled the operation. The earnings record work sheet provided storage for data and instructions alike.

A stored-program computer operates on the same principle. It accepts the data to be processed and the instructions which process the data from the outside, and writes them both in storage in the same manner. To execute the program formed by the instructions, the computer locates each instruction in turn—either sequentially or in an order specified by the code—looks at it, and operates as the instruction directs on the information the instruction specifies. An instruction word is identified as such—that is, as different from a data word—by the control element: the clerk in our analogy, or the control unit in the computer operation.

ADDRESS CONCEPT

Now the question arises of how to keep track of individual words in storage; how to locate an instruction to be executed, an item of data to be processed, or a place to store data or an instruction.

Recall that in our analogy the data was uniquely identified by numbers 1 through 6. These numbers indicated the column in which the number would be found—the number in that column opposite the current date. Thus each piece of data had a specific location on the earnings record card—the storage element of the analogy:



This is the principle by which words of information are identified and located in computer storage. Each word of information—data or instruction—which enters a computer is written in storage in a specific location identified by a unique number, called an address.

Just as the number in the location on the earnings record card which is identified by 1 is completely different from 1, so are the contents of a location in computer storage completely different from the address which identifies the location. The addresses of locations play an inactive part in any operation—that of a directive for placement and location of the pieces of data or instructions. The contents of the location play the active part: they can be either the instructions which process the data or the data to be processed.

When the clerk wants to perform an operation on a number in a specific location, he keys that number into the desk calculator. The number now appears in two places, in the desk calculator and in its original location on the earnings record card. The same thing occurs in computer storage. When a number is taken from storage and sent to the arithmetic unit for some operation on it, a copy of the number remains in the location.

If the clerk chose to write a number into a location on the earnings record card which already contained a number, he would have to erase the number written there first. In computer storage, when a number is written into a location, any number previously written there is automatically erased.

REGISTERS

When the clerk in our analogy performed calculations on the desk calculator, he keyed the operands into the desk calculator before depressing the designated motor bar. Thus the keyboard of the desk calculator provided temporary storage for the operands.

In computers there are certain one-word locations separate from the internal storage unit which provide temporary storage for operands and control words in computer operations. These locations are called registers: a register stores an operand or a control word while or until it is used.

The general description given in this chapter would apply to any stored-program digital computing system. Chapter 3 begins the discussion of the Burroughs 220.

3

An Introduction to the Burroughs 220

GENERAL

The Burroughs 220 electronic data processing system is the kind of computer system described in Chapter 2. It is a general-purpose computing system—one that is suitable for both scientific and business applications and it can be expanded to fit the nature of the job to be done. The purpose of this chapter is to describe this system in more detail—to introduce the parts of the system and explain their function.

As a system, the Burroughs 220 is comprised of several units. These units may be thought of as (1) those included in the data-processing section of the system and (2) those concerned with input and output.

DATA-PROCESSING SECTION

The data-processing section—or the computer section—of the Burroughs 220 consists of the following three units:

1. Data Processor: the arithmetic unit that performs the arithmetic and comparing operations and manipulates words of information.
2. Core Storage: the unit that provides the internal (or working) storage for the system.
3. Memory Control: the unit that controls the transfer of information between working storage and the Data Processor.

The Control Console (Fig. 3-1) is a separate unit that, because of its function, is an integral part of the data-processing section. It is equipped with operation controls (start button, stop button, etc.) and indicators for the whole system. Its main function is to provide for manual monitoring of system operations.

INPUT-OUTPUT SECTION

The Burroughs 220 system is designed to permit the inclusion of the following input and output units.

1. Photoreader: a photo-electric input device that reads into core storage information that has been punched into paper tape (Fig. 3-1).
2. Character-at-a-Time Printer: an output printing device that types information transferred directly from core storage.
3. Paper-Tape Punch: an electromechanical output device that punches into paper tape information transferred directly from core storage.
4. Cardatron: an electronic system that links the computer to input and output card machines and line printers. It accepts input information from punched cards, translates the information to the representation used in the Burroughs 220[1], edits the information to conform to the Burroughs 220 word length and format, and transmits it to core storage[2]. The Cardatron[3] also receives output information from core storage, translates it to card machine representation, edits it to conform to a specified output format, and transmits it to a line printer or card punch.
5. Magnetic Tape Storage Devices: electromechanical devices that read from—and record on—magnetic tape used by the system for auxiliary storage. These devices accept information from core storage and write it onto magnetic tape; they can also read information on magnetic tape and transmit it to core storage.
6. Manual Keyboard: a ten-key manually operated numeric keyboard associated with the Control Console. The keyboard is used to enter a few words at a time into core storage or to alter register contents (Fig. 3-1).

SYMBOLIC NOTATION OF INSTRUCTIONS

The Burroughs 220 is designed to use instructions in numeric form. Each operation in its vocabulary is assigned a numeric code: for example, in the Burroughs 220 the multiply operation is assigned the numeric operation code 14.

WORD CONCEPT

A Burroughs 220 word is 11 decimal digits in length. Ten decimal digits represent data or an instruction; the eleventh is the sign-digit position:

Digit Positions	±	1	2	3	4	5	6	7	8	9	0

Computer Word

To identify individual digits within a word, the digit positions of the word are uniquely numbered from left to right, excluding the sign-digit position; the sign digit is represented by the symbol ±.

[1]Appendix A is a description of how information is represented in the Burroughs 220.

[2]Appendix B is a description of how information is stored in the Burroughs 220.

[3]Trademark of the Burroughs Corporation.

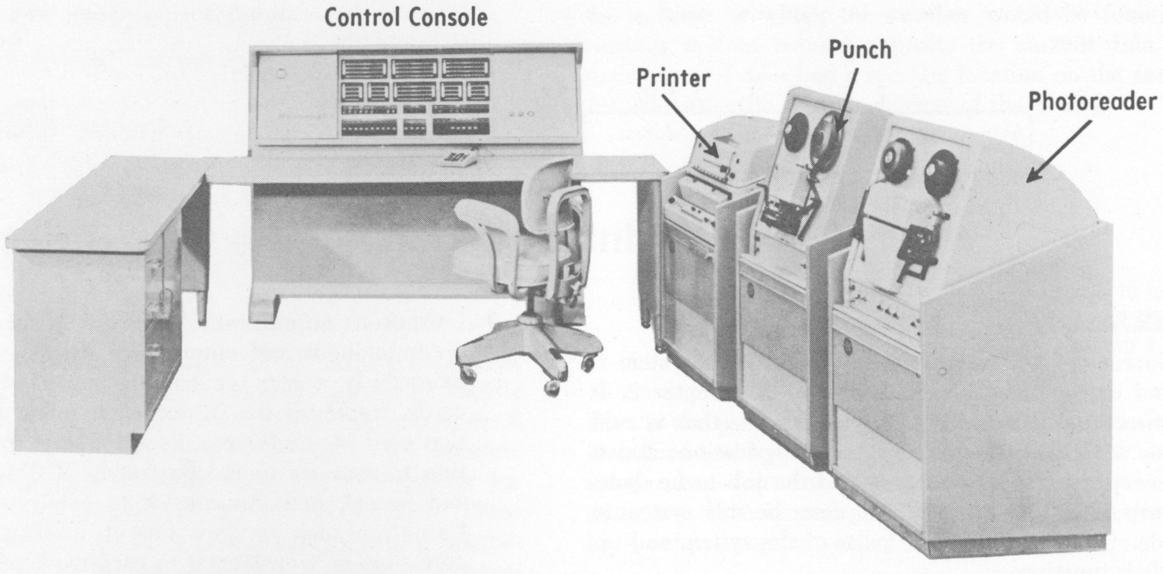


Figure 3-1. Control Console Printer, Punch, and Photoreader

The leftmost digit position is referred to as the sign-digit position; the next digit position is called the first digit position; the next digit position is called the second digit position and so forth. The last or rightmost digit position —although numbered 0—is called the tenth digit position. The first digit position may also be referred to as the high-order digit of the word; the tenth digit position may be referred to as the low-order digit position of the word.

The sign-digit position is used to designate the algebraic sign of numeric words; that is, it tells whether the word is plus or minus. A 0 in the sign-digit position represents a plus sign; a 1 denotes a minus sign.

The sign digit of instruction words has no algebraic significance; it is used for control purposes. (This use will be discussed under the topic of B Register in this chapter and in more detail in Chapter 8.)

The 11 decimal digits comprising a Burroughs 220 word can represent an instruction word or a data word. The control unit in the data-processing section of the system determines how the word is interpreted. For example, the word 0 4259 10 4955 can represent the number + 4,259,104,955 or it can represent the instruction "clear the A register and insert the contents of storage location 4955 into the A register."

A data word may also represent alphanumeric information. In the Burroughs 220, the alphanumeric code is a two-digit code; that is, a single alphanumeric character is represented by a pair of adjacent decimal digits. Thus a single computer word can represent a maximum of five alphanumeric characters.

The alphanumeric code must be a two-digit code because the number of alphabetic, numeric, and special characters to be represented exceeds ten. A single digit position in the Burroughs 220 could represent only ten different characters—each having a code 0 through 9.

In the Burroughs 220 alphanumeric code system, special characters are assigned a code from 00 through 30, alphabetic characters are assigned a code from 40 through 69, and numeric characters are assigned a code from 80 through 89.

To illustrate, the following is a sample section number representing section 7—51 in the Marketing Department:

± 54 87 20 85 81
M 7 — 5 1

INSTRUCTION FORMAT

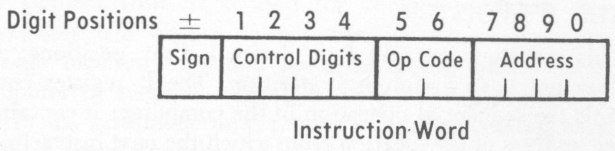
So far we have considered the nature of an instruction, the function it performs in a program, and how it is stored. Now let us consider in detail the instruction format of the Burroughs 220.

The Burroughs 220 uses a single-address instruction code; that is, one instruction per word, one address per instruction. Each instruction word may refer to one and only one storage address.

Since instructions are, in general, executed in the sequence in which they are stored in the Burroughs 220, the address of the next instruction to be executed need not be specified in a 220 instruction. For example, if the instructions of a four-step program were stored in locations 1000, 1001, 1002, and 1003, the instruction in location 1001 would automatically be executed after the instruction in location 1000; the instruction in location 1002 would automatically be executed after the instruction in location 1001, etc.

In the single-address instruction code employed by the 220, the location of only one operand in an arithmetic operation is specified by the instruction; the other operand is always the contents of a specific register. A separate instruction is needed to store the result.

An instruction word is divided into three parts (excluding the sign digit): the address, the operation code, and the control digits.



The four low-order digit positions of an instruction word store the address part of the instruction. In cases where the contents of a specific location are to be used during the execution of an instruction, these four digits specify the address of that location. Otherwise, as in the case of a manipulation instruction where no data from storage is used, the address part is irrelevant; it may sometimes be used to specify some other quantity, such as a specific constant.

Digit positions 5 and 6 store the operation code—the numeric equivalent of the operation specified; for example, 14 is the numeric equivalent of the MULTIPLY instruction.

Digit positions 1 through 4 store what are called control digits. These digits are used to designate special properties or variations of the instruction.

REGISTERS

The Burroughs 220 uses electronic registers for temporary storage of instructions or data words brought from core storage to be used in a Data Processor operation. A word is stored in a register while or until it is used. Registers are also used to store information necessary for control of computer operation. During computer operation, the contents of the individual registers are displayed on the control panel (Fig. 3-2) of the Control Console. (Appendix C is a description of how information is stored in a register and how the register contents are displayed.)

Each of the registers in the Burroughs 220 has a specific name and function; a description of the registers associated with the computing portion of the system follows.

THE A REGISTER

This register contains 11 decimal digit positions: ten digits for the instruction or data word and one digit for the sign. This register is used to store one of the operands in an arithmetic operation and to store an instruction or data word to be manipulated under program control. It also acts as an accumulator; that is, the results of most operations appear in this register.

THE R REGISTER

This register contains 11 decimal digit positions: ten digits plus sign-digit position. This register is primarily an extension of the A register, as shown in Fig. 3-3.

THE D REGISTER

This register contains 11 decimal digit positions: ten digits plus sign-digit position. In operations involving two operands, the operand whose address is specified by the instruction is brought from core storage and placed in the D register before the operation begins. When an instruction word is brought from core storage and sent to the C register for execution, a copy of the instruction as it appeared in core storage is contained in the D register.

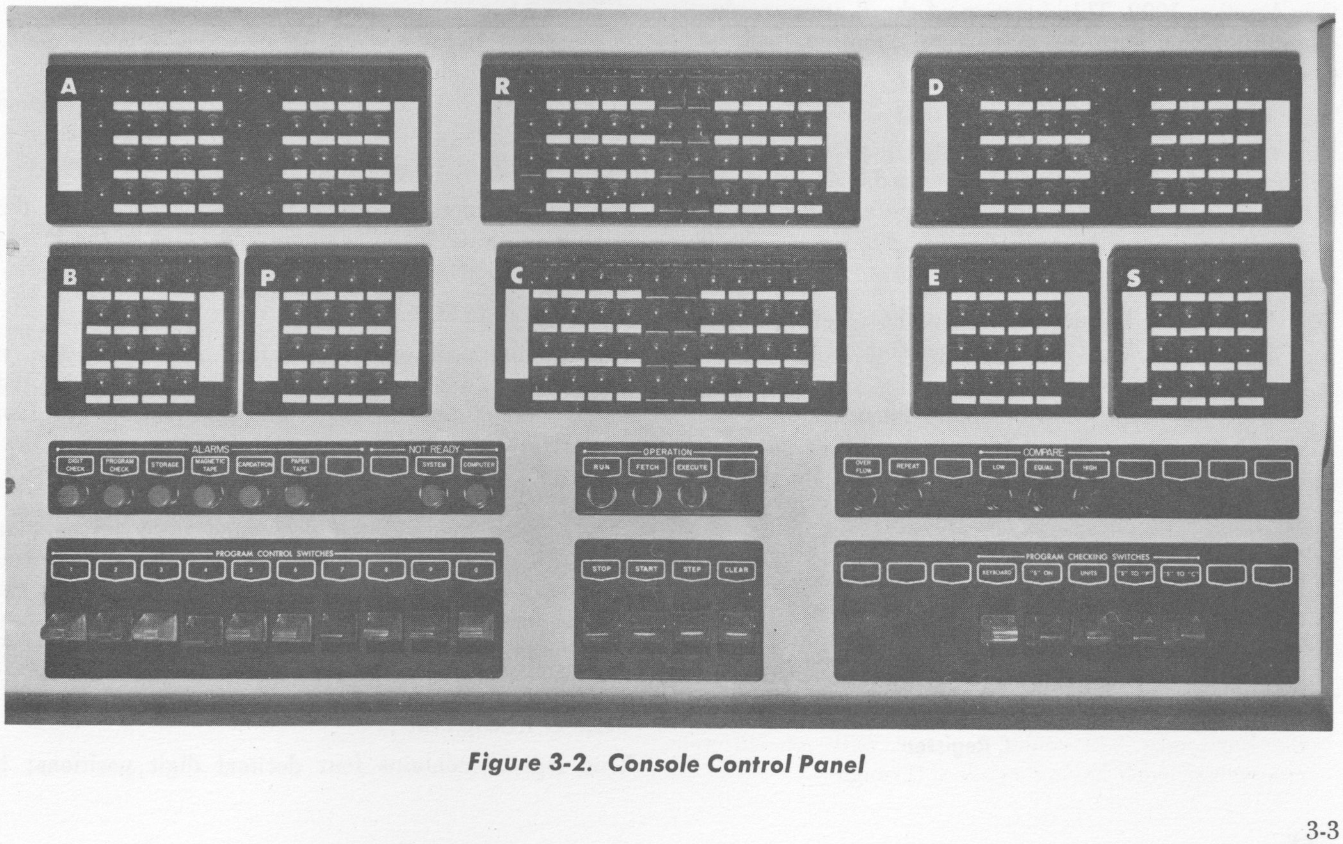


Figure 3-2. Console Control Panel

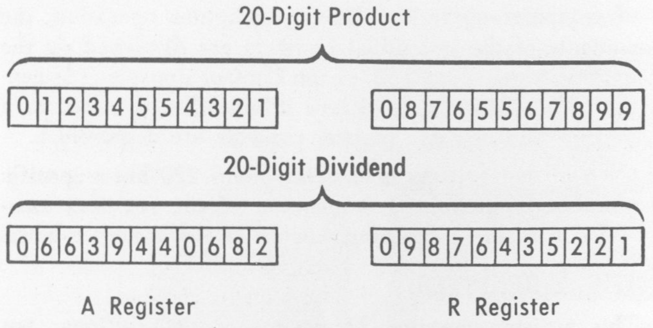


Figure 3-3. A and R Registers

THE B REGISTER

This register contains four decimal digit positions: it does not have a sign-digit position. The primary function of the B register is to provide for address modification; that is, when so specified in the sign-digit position of an instruction brought from core storage for execution, the number contained in the B register is added to the address portion of the same instruction as it is sent to the C register for interpretation and execution. Thus, when executed, the modified copy of the instruction will reference a storage location that is different from the one referenced by the unmodified copy retained in core storage and in the D register.

For example: Suppose an instruction in core storage has 1000 as its address. Let us suppose also that the B register contains 0500. If the instruction brought from storage indicates that B register address modification is to take place, the contents of the B register are added to the address of the instruction (1000 + 0500). When executed, the instruction will reference location 1500 rather than location 1000. This function of the B register, plus other functions, such as tallying or counting, will be discussed in Chapter 8.

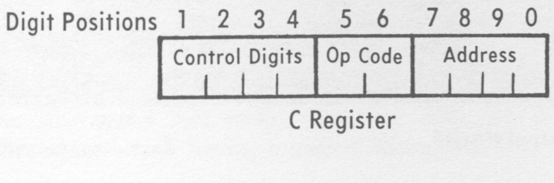
THE C REGISTER

This register contains 10 decimal digit positions; it does not have a sign-digit position. Any word that enters the C register is interpreted as an instruction and executed.

It is convenient to regard the C register as being divided into three parts:

1. The four high-order digit positions (digit positions 1, 2, 3, and 4) of the C register contain the control digits of the instruction.
2. Digit positions 5 and 6 contain the operation code.
3. The four low-order digit positions (digit positions 7, 8, 9, and 10) of the C register contain the address part of the instruction.

The sign digit—used for control purposes in an instruction word—is not contained in the C register. The significance of this digit is checked in the D register where a copy of it remains.



THE P REGISTER

This register contains four decimal digit positions; it does not have a sign-digit position. The P register controls the sequential operation of the computer; it contains the address of the location from which the next instruction will be taken for execution.

For example, if the instruction to be executed next were in location 0500, the P register would contain 0500. After this instruction is taken from location 0500 and sent to the C register for execution, the contents of the P register are increased by one. The next instruction will be taken from location 0501.

This sequential operation mode can be interrupted by a transfer-control instruction. Such an instruction causes the contents of the P register to be replaced by the address portion of the transfer-control instruction. The next instruction would be taken from that address. To illustrate: in the previous example, when the instruction in location 0500 was sent to the C register for execution, the P register contained 0501, the address of the next instruction in sequence to be executed. If, however, the instruction in location 0500 had been a transfer-control instruction, the address specified by that instruction would have been placed in the P register. The next instruction executed would not have been the instruction in location 0501; instead, it would be the instruction in the location whose address was specified by the transfer-control instruction.

Location of Instruction in C Register	*Transfer Control Instruction*	*P Register*
0498	NO	0499
0499	NO	0500
0500	YES	Address specified by transfer-control instruction.

The next instruction to enter the C register would be the instruction in the location whose address is in the P register—that specified by the transfer-control instruction.

THE S REGISTER

This register contains four decimal digit positions; it does not have a sign-digit position. The S register is used by an operator for checking out (debugging) a program on the computer.

THE IB REGISTER

This register contains ten decimal digit positions plus sign-digit position. It is located in the Memory Control Unit and is used as a buffer between core storage and the control and arithmetic units; that is, instructions and data going to the A, R, D, or C registers from core storage first pass through this register.

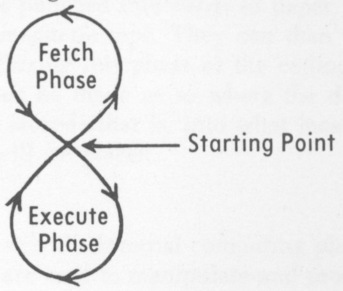
THE E REGISTER

This register contains four decimal digit positions; it does not have a sign-digit position. Located in the Memory Control Unit, it is used for control purposes: the E register will always contain the address of the core storage location to which access is being made under computer or manual control. Although the E register is not in the arithmetic unit, a copy of its contents appears on the Control Console.

OPERATION CYCLE

Whenever the Burroughs 220 seeks and locates an instruction in core storage, transfers it to the C register, and then performs the action specified by the instruction, it has performed the basic cycle of computer operation. This cycle—referred to as the operation cycle—has two phases: the first is called the fetch phase; the second is called the execute phase. During the fetch phase, the instruction is brought from core storage to the C register; during the execute phase, the instruction just fetched is executed.

One might picture the operation cycle as following a pattern like a figure eight:



That is, the computer begins its operation with the fetch phase. As soon as it has completed the fetch phase, it enters the execute phase. When it completes the execute phase, it re-enters the fetch phase, and so forth. Thus, during normal operation, the fetch and execute phases are performed alternately.

A description of what occurs during the execute phase of each instruction appears in subsequent sections of this book.

INPUT-OUTPUT MEDIA

The input-output equipment mentioned at the beginning of this chapter uses various media (punched paper tape, magnetic tape, punched cards) from which information to be transmitted to core storage is obtained and on which information transmitted from core storage is recorded. In addition, the output from core storage may be printed.

During input preparation, data and instructions are recorded on a selected medium and presented to a compatible input device for transmission to core storage. During output, information is presented to an output device which records it on a compatible output medium. (Appendix D is a description of the input-output media employed by the Burroughs 220.)

4 Starting to Code the Burroughs 220

APPROACHING THE PROBLEM

In starting to code the Burroughs 220, three parts of a data-processing problem must be considered: input, processing, and output.

INPUT

An input medium must be decided upon by which data and instructions can be entered into the computer. The numeric and alphanumeric information—representing the data to be processed and the instructions that perform the processing—may be punched into cards or paper tape, or may be stored on magnetic tape. They can then be read into core storage. During this phase of the coding operation, a decision must be made as to where the data and instructions will be stored; that is, into what locations of core storage they will be placed.

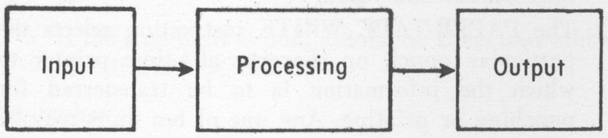
PROCESSING

The processing phase is the internal computing phase; the stored instructions are used to manipulate and process the stored data.

OUTPUT

Some means must be decided upon by which the processed results may be made available for use. This may be accomplished by printing out the processed data, punching it into cards or paper tape, or storing it on magnetic tape.

These three phases may be shown as:



To illustrate, let us take a simple example problem. Assume that we have two quantities, x and y. We want to enter them into the computer, add them together to obtain their sum, z, and print out that sum. We have decided to enter the data—x and y—into locations 0001 and 0002. When the sum is computed it will be temporarily stored in location 0100, prior to being printed out. The instructions that we have written to process the data will be stored in locations 1995 through 1999. Punched paper tape has been selected as the input medium to read the data and instructions into the computer.

For the sample problem, storage layout will look like this:

Storage Location	*Contents of Location*	
0000	Not used	
0001	Quantity x	Data
0002	Quantity y	
. . .	Locations 0003 through 0099 not used for this problem.	
0100	Storage location for z (computed result).	
. . .	Locations 0101 through 1994 not used for this problem.	
1995	Instruction 1	
1996	Instruction 2	
1997	Instruction 3	
1998	Instruction 4	
1999	Instruction 5	

When the data and instructions have been stored, the computer will be directed to execute the instructions starting with the one in location 1995.

The instructions will direct the computer to do the following:

Instruction 1.	Clear the A register. Take quantity x from location 0001 and place it in the A register.
Instruction 2.	Take quantity y from location 0002 and add it to quantity x in A register; the sum z will appear in the A register.
Instruction 3.	Take the sum z from the A register and store it in location 0100.
Instruction 4.	Print the sum.
Instruction 5.	Halt the operation.

These instructions will be explained in detail later.

PAPER-TAPE SYSTEM

The Paper-Tape System and two of its instructions will be discussed as an example of input-output media.[1] The various arithmetic, manipulation, and decision-making instructions will be discussed in this and subsequent chapters.

The Burroughs 220 uses paper tape as one of its input-output media. The paper-tape input equipment for the Burroughs 220 is a photoelectric reader. The paper-tape

[1]For a detailed description of the paper-tape system and the remaining paper-tape instructions, consult Operational Characteristics of the Burroughs 220, Bulletin 5020.

output equipment is a paper-tape punch; a character-at-a-time printer may be substituted for a paper-tape punch. Up to ten paper-tape photoreaders and up to ten high-speed paper-tape punches or character-at-a-time printers may be included in the Paper-Tape System of the Burroughs 220. However, only one input or output unit may be referenced at a time by any one instruction.

On input, the computer can be directed to read either a specified number of words or all the words contained on a length of punched paper tape. Or it can read an unspecified portion of the total words by interspersing the input information on the tape with control words. On output, the coder must specify the number of words to be punched or printed.

CONTROL WORDS

As a general rule, instruction words with a 6 or 7 in the sign-digit position are recognized by the computer as control words. Whether receiving input information from punched paper tape or punched cards, the computer can distinguish these words from other input words if it is directed to do so. It is also possible to ignore 6 and 7 sign digits in punched paper tape and punched-card input operations.

If the computer has been directed to recognize control words, a control word is never sent to storage; instead, it is sent to the C register and executed. Otherwise, a control word is read into storage like any other input word.

Same samples of control words are:

1. A PAPER-TAPE READ instruction with a 6 or a 7 in the sign-digit position might be punched into paper tape preceding the information to be read into core storage. As a control word it would be sent to the C register and executed—thus causing the information following it on the paper tape to be read into core storage starting with the location specified in the address portion of the control word.
2. A control word might be punched into the paper tape in the middle of the input information which, when sent to the C register, would halt the reading operation.
3. A control word might be punched into the paper tape at the end of the input information which, when sent to the C register, would halt the reading operation and transfer control to the first location of a program just read in, so that computation could begin.

The use and purpose of control words will be further clarified for the reader in connection with the descriptions of the paper-tape and Cardatron instructions. (Control blocks perform the same functions with magnetic tape. Refer to Chapter 11. For more complete details, refer to Operational Characteristics of the Burroughs 220.)

Descriptions of the two paper-tape instructions selected as examples follow.

PAPER-TAPE READ (03) ± u nn v PRD aaaa[2]

±	If the sign digit is odd, automatic B register address modification occurs.
u	Designates the unit from which the information is to be read.
nn	Number of words to be read.
v = 0	Read nn words; control words are not recognized as such.
v = 1	Read nn words or read until a control word is encountered; control words are recognized as such.
v = 8 or v = 9	B register address modification of specified input will occur.

1. "Read nn words from unit u into consecutively addressed locations beginning with location aaaa."
2. The PAPER-TAPE READ instruction selects the particular photoreader from which the information is to be read. Any one of 10 photoreaders can be selected by coding a digit from 1 through 0 in the "u" digit position of the instruction; a 1 specifies unit 1, a 0 specifies unit 10.

PAPER-TAPE WRITE (06) ± u nn 0 PWR aaaa

±	If the sign digit is odd, automatic B register address modification occurs.
u	Designates the unit by which the information is to be printed or punched.
nn	Number of words to be printed or punched.
0	Not relevant to the execution of this instruction.

1. "Print or punch nn words on unit u, taking the words from consecutively addressed locations beginning with location aaaa."
2. The PAPER-TAPE WRITE instruction selects the particular punch or character-at-a-time printer to which the information is to be transferred for punching or printing. Any one of ten units may be selected by coding a 1 through 0 in the u digit position of the instruction; a 1 specifies unit 1, a 0 specifies unit 10.

ADDITION AND SUBTRACTION INSTRUCTIONS

CLEAR ADD (10) ± 0000 CAD aaaa

1. "Replace the entire contents of the A register by the contents of storage location aaaa."
2. If the sign digit is odd, automatic B register address modification occurs.

[2]The coder writes the instruction as shown: ± u nn v PRD aaaa, but on input preparation, the mnemonic operation code must be replaced by the numeric: ± u nn v 03 aaaa.

Examples:

A Register Before CAD	*Contents of Location aaaa*	*A Register After CAD*
0 1111 11 1111	0 1234 56 7890	0 1234 56 7890
0 4976 00 3872	0 0000 38 7421	0 0000 38 7421

ADD (12) ± 0000 ADD aaaa

1. "Add the contents of location aaaa to the contents of the A register."
2. The resulting ten-digit sum replaces the contents of the A register. The sign of the A register is set according to the rules for algebraic addition. Exception: If the result of the addition is zero, the sign of the result is the sign of the A register before execution of the ADD instruction.
3. If the sum exceeds the capacity of the A register, overflow occurs and the overflow indicator is turned on. (See discussion on overflow later in this chapter.)
4. If the sign digit is odd, automatic B register address modification occurs.

Examples:

A Register Before ADD	*Contents of Location aaaa*	*A Register After ADD*	*Overflow Indicator*
0 1111 11 1111	0 4444 44 4444	0 5555 55 5555	OFF
0 1111 11 1111	0 1234 56 7890	0 2345 67 9001	OFF
0 4343 33 5757	1 4343 33 5757	0 0000 00 0000	OFF
0 8604 30 0000	0 1703 00 0000	0 0307 30 0000	ON

ADD TO LOCATION (19) ± 0000 ADL aaaa

1. "Add the contents of the A register to the contents of location aaaa."
2. The sum appears in location aaaa.
3. The sign digit of the word in location aaaa is produced according to the rules of algebraic addition.
4. If the sum exceeds the capacity of location aaaa, overflow occurs and the overflow indicator is turned on.
5. If the sign digit is odd, automatic B register address modification occurs.

Examples:

Contents of Location aaaa Before ADL	*Contents of A Register*	*Contents of Location aaaa After ADL*	*Overflow Indicator*
0 4444 44 4444	0 0000 11 1111	0 4444 55 5555	OFF
0 0000 00 0015	1 0000 00 0025	1 0000 00 0010	OFF
0 8910 00 4136	0 1120 00 1221	0 0030 00 5357	ON

CLEAR SUBTRACT (11) ± 0000 CSU aaaa

1. "Replace the entire contents of the A register by the contents of location aaaa."
2. The sign of the word in aaaa is reversed before it enters the A register, that is, if the word has a positive sign, it will appear in the A register with a negative sign. If the sign was negative, it will be positive when it enters the A register.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before CSU	*Contents of Location aaaa*	*Contents of A Register After CSU*
0 7689 00 0322	0 1234 56 7890	1 1234 56 7890
1 0000 00 0000	1 9336 44 7890	0 9336 44 7890
0 0000 00 0044	1 0000 00 0893	0 0000 00 0893

SUBTRACT (13) ± 0000 SUB aaaa

1. "Subtract the contents of aaaa from the contents of the A register."
2. The difference replaces the original contents of the A register.
3. The sign of the result is set according to the rules of algebraic subtraction. Exception: If the result of the subtraction is zero, the sign of the result is the sign of the A register before execution of SUBTRACT.
4. The overflow indicator is turned on if the result of the subtraction exceeds the capacity of the A register.
5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before SUB	*Contents of Location aaaa*	*Contents of A Register After SUB*	*Overflow Indicator*
0 0000 00 0333	0 0000 00 0011	0 0000 00 0322	OFF
1 0000 00 0333	0 0000 00 0011	1 0000 00 0344	OFF
1 0000 00 0333	1 0000 00 0011	1 0000 00 0322	OFF
1 4244 58 7890	0 8000 00 0000	1 2244 58 7890	ON
0 3333 33 3333	1 6666 66 6667	0 0000 00 0000	ON

CLEAR ADD ABSOLUTE (10) ± 0001 CAA aaaa

1. "Replace the entire contents of the A register by the absolute value of the contents of location aaaa."
2. The sign of the word from storage is treated as though it were positive regardless of its actual sign.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before CAA	*Contents of Location aaaa*	*Contents of A Register After CAA*
0 1234 56 7890	1 4320 00 9001	0 4320 00 9001
1 8000 08 8000	0 1111 11 1111	0 1111 11 1111

ADD ABSOLUTE (12) ± 0001 ADA aaaa

1. "Add the absolute value of the contents of aaaa to the contents of the A register."
2. The resultant ten-digit sum replaces the contents of the A register. The sign of the A register is set according to the rules for algebraic addition. Exception: If the result of the addition is zero, the sign of the result is the sign of the A register before execution of the ADA instruction.

3. If the sum exceeds the capacity of the A register, overflow occurs and the overflow indicator is turned on.
4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before ADA	*Contents of Location aaaa*	*Contents of A Register After ADA*	*Overflow Indicator*
0 0000 00 0088	0 0000 00 0012	0 0000 00 0100	OFF
1 0000 00 0099	1 0000 00 0012	1 0000 00 0087	OFF
0 8000 00 2345	1 3220 00 0249	0 1220 00 2594	ON
0 6231 11 4890	1 3900 00 0000	0 0131 11 4890	ON
1 0006 66 0004	0 0006 66 0004	1 0000 00 0000	OFF

CLEAR SUBTRACT ABSOLUTE (11)
± 0001 CSA aaaa

1. "Replace the contents of the A register with the contents of location aaaa; the sign of the A register will be negative."
2. The sign of the word in aaaa is disregarded.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before CSA	*Contents of Location aaaa*	*Contents of A Register After CSA*
0 0000 45 6300	0 0084 93 7000	1 0084 93 7000
1 7777 77 7777	1 5678 12 3450	1 5678 12 3450

SUBTRACT ABSOLUTE (13) ± 0001 SUA aaaa

1. "Subtract the absolute value of the word in location aaaa from the contents of the A register."
2. The sign of the word in location aaaa is treated as positive regardless of its actual value.
3. The ten-digit difference replaces the contents of the A register.
4. The sign of the result is set according to the rules for algebraic subtraction. Exception: If the result of the subtraction is zero, the sign of the result is the sign of the A register before execution of SUA.
5. If the result of subtraction exceeds the capacity of the A register, the overflow indicator is turned on.
6. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register Before SUA	*Contents of Location aaaa*	*Contents of A Register After SUA*	*Overflow Indicator*
0 0000 00 0000	0 0000 00 8809	1 0000 00 8809	OFF
1 0000 00 0888	1 0000 00 0008	1 0000 00 0896	OFF
1 8070 38 0000	0 0007 66 0049	1 8078 04 0049	OFF
1 9999 99 9999	0 0000 00 0001	1 0000 00 0000	ON
1 8004 48 0000	1 8700 00 0000	1 6704 48 0000	ON

HALT INSTRUCTION

Even the shortest and simplest programs must include a HALT instruction.

HALT (00) ± 0000 HLT 0000

1. "Stop operation of the computer."
2. If the operator depresses the START button, after the HALT instruction has been executed, the computer will fetch and execute the contents of the location immediately following the HALT.
3. When the HALT instruction is executed, the contents of the A, R, C and B registers are undisturbed.
4. A program usually contains several HALT instructions. Since the address portion of this instruction is not used to reference a storage location, it may be used to assist the programmer in identifying HALT instructions. For example, he can identify the following halts by examining the contents of the C register displayed on the Console control panel:

0000 00 0001	=	Halt No. 1
0000 00 0002	=	Halt No. 2
0000 00 0003	=	Halt No. 3

5. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits occurs.

PARTIAL-WORD OPERATION

Instructions normally refer to the entire word in a register or in storage. Some operations, however, allow the coder to specify either an entire word or any digit or set of adjacent digits (field) in a word.

The digit or digits referenced by this type of instruction are called partial-word fields, and this type of machine operation is called a partial-word operation.

The four high-order or control digits of partial-word instructions are used to specify the following:

1. Total-word or partial-word operation, that is, whether the entire word or only a part of the word will be operated upon.
2. The size and location of the partial-word field within the word, if partial-word operation is indicated.

   A partial-word field is defined in the Burroughs 220 by the letters sL: s specifies the rightmost—low-order—digit of the partial-word field; L specifies the number of digits in the field. In other words, the field is determined by starting at digit-position s and counting left L digit positions.
3. The variation of the instruction to be executed in cases where two instructions have the same operation code. (For example, the STORE A and STORE R instructions both have the numeric operation code 40. STORE A is distinguished from STORE R by having a 0 in digit-position 4, the variation-digit position, while STORE R has a 1 in this position.)

SAMPLE PARTIAL-WORD INSTRUCTIONS

STORE A (40) ± s L f 0 STA aaaa

If f = 1 Partial-word operation.

If f = 0 Total-word operation.

If f = 0, sL not relevant.

If f = 1, s specifies rightmost digit position of the partial-word field, L specifies number of adjacent digits in the partial-word field.

If the sign digit is odd, automatic B register address modification occurs.

1. "Store the specified field of the A register in the corresponding field in location aaaa."
2. If f = 0, the STORE A instruction replaces the entire contents of location aaaa by the entire contents of the A register.
3. If f = 1, the sL digits specify the partial-word field. Since partial-word operations reference only a specified field, the remaining digits of the word in location aaaa are unaltered.
4. Execution of STORE A does not alter the contents of the A register.
5. Any individual digit except the sign digit can be treated as a separate field. The sign digit must be referenced in conjunction with at least one adjoining digit.

Examples:

Contents of location aaaa before STA is executed: 0 9999 99 9999

Instruction	*Contents of A Register Before and After STA Is Executed*	*Contents of Location aaaa After Execution of STA*
0 0000 STA aaaa	0 2222 22 2222	0 2222 22 2222
0 0300 STA aaaa	0 0000 12 3456	0 0000 12 3456
0 0310 STA aaaa	0 0000 12 3456	0 9999 99 9456
0 6210 STA aaaa	0 3333 33 3333	0 9999 33 9999
0 0010 STA aaaa	0 4400 44 0044	0 4400 44 0044
0 3410 STA aaaa	1 5432 15 8765	1 5439 99 9999

STORE R (40) ± s L f 1 STR aaaa

If f = 1, partial-word operation.

If f = 0, total-word operation.

If f = 0, sL not relevant.

If f = 1, s designates rightmost digit position of the partial-word field, L specifies the number of adjacent digits in the partial-word field.

If the sign digit is odd, automatic B register address modification occurs.

1. "Store the specified field of the R register in the corresponding field of location aaaa."
2. If f = 0, the STORE R instruction replaces the entire contents of location aaaa by the entire contents of the R register.
3. If f = 1, the sL digits specify the partial-word field. Since partial-word operations reference only a specified field, the remaining digits of the word in location aaaa are unaltered.
4. Execution of STORE R does not alter the contents of the R register.
5. Any individual digit except the sign digit can be treated as a separate field. The sign digit must be referenced in conjunction with at least one adjoining digit.

Examples:

Contents of location aaaa before STR is executed: 0 1111 11 1111

Instruction	*Contents of R Register Before and After STR is Executed*	*Contents of Location aaaa After Execution of STR*
0 0001 STR aaaa	0 9999 99 9999	0 9999 99 9999
0 3211 STR aaaa	0 5346 70 6434	0 1341 11 1111

Consider the following short program as an example of some of the instructions that have been described.[3]

Given: *Year to Date*	*This Week*
Gross in location 0100	Gross in location 0200
Tax in location 0101	Tax in location 0201
FICA in location 0102	FICA in location 0202
Insurance in location 0103	Insurance in location 0203
Net in location 0104	

Find net pay (net = gross − tax − FICA − insurance) for this week; store in location 0204. Update year to date. Start program in location 0000.

```
0000   CAD   0200 }
0001   SUB   0201 }  Calculate net pay this week.
0002   SUB   0202 }
0003   SUB   0203 }
0004   STA   0204    Store net pay this week.
0005   ADL   0104    Update net pay.
0006   CAD   0200 }
0007   ADD   0100 }  Update gross.
0008   STA   0100 }
0009   CAD   0201 }
0010   ADD   0101 }  Update tax.
0011   STA   0101 }
0012   CAD   0202 }
0013   ADD   0102 }  Update FICA.
0014   STA   0102 }
0015   CAD   0203 }
0016   ADD   0103 }  Update insurance.
0017   STA   0103 }
0018   HLT   0000    Halt operation.
```

[3]The reader should be aware that the sample problems are not always realistic; in many of them, more instructions appear than would be used by an experienced coder. The examples were chosen for simplicity and for the purpose of introducing new instructions gradually.

CONCEPT OF OVERFLOW

A Burroughs 220 word is always the same size: it must contain 10 decimal digits plus a sign digit to be acceptable to the computer.

Although two stored numbers meet this word-size requirement, their sum may exceed this size, just as in everyday arithmetic the sum of two 10-digit numbers may produce an 11-digit result. For example:

```
   8432 16 7431
+  1843 16 7142
---------------
  10275 33 4573
```

Such a result is too large to be inserted in the A register. The 11th or high-order digit, generated by the "carry one," creates a number that exceeds the register capacity. Because the A register has room for only the 10 low-order digits of the result, the high-order digit is lost. The largest number that can be stored in the computer is 9999 99 9999; the sum of two words of this magnitude will equal 1 8888 88 8888. Thus the lost digit is always a 1.

Such an occurrence—creation of a number too large to be contained in the A register—is referred to as overflow.

Examples:

Arithmetic Operation	*Appearance of A Register*	*Overflow*
+ 8000 00 0000		
(+) + 1000 00 0000		
+ 9000 00 0000	0 9000 00 0000	NO
+ 9000 00 0000		
(+) + 3000 00 0000		
+ 1 2000 00 0000	0 2000 00 0000	YES
+ 4841 99 9978		
(+) + 5168 00 9122		
+ 1 0010 00 9100	0 0010 00 9100	YES
+ 9841 99 9978		
(−) − 0168 00 9122		
+ 1 0010 00 9100	0 0010 00 9100	YES

The computer indicates overflow by turning on the overflow indicator. The computer will stop when overflow occurs, unless the overflow indicator is turned off by the program.

When the possibility of overflow is anticipated, a BRANCH ON OVERFLOW instruction must be inserted in the program immediately following the instruction that may cause the overflow. If overflow occurs, the BRANCH ON OVERFLOW instruction will turn off the overflow indicator, preventing the computer from stopping, and causing a transfer of control to an alternate location. If overflow does not occur, the BRANCH ON OVERFLOW instruction is executed but no branch occurs; program control continues in sequence.

Recognition of an overflow condition when it occurs is a valuable programming aid. Use of this technique is discussed in Chapter 7 under Tallying and Address Modification.

BRANCHING (TRANSFER OF CONTROL)

Normal Burroughs 220 operation calls for sequential execution of program instructions from successively addressed storage locations. Situations arise, however, where interruptions of this sequence are necessary. Branching instructions are then used.

A branching instruction can cause program control to be transferred to the instruction in the location specified by its address. That is, the next instruction to be fetched after a branch is not the one following the branching instruction, but the one in the location specified by the address part of the branching instruction. Sequential operation is resumed at the point to which control is transferred. For example:

BRANCH UNCONDITIONALLY (30)
± 0000 BUN aaaa

1. "Transfer control to the instruction in location aaaa."
2. Sequential operation resumes after the branch.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Example:

Location	*Instruction*
1000	0 0000 CAD 2225
1001	0 0000 ADD 1500
1002	0 0000 STA 1550
1003	0 0000 BUN 3000
1004	0 1000 PRB 1000

On the BUN instruction, control is transferred to location 3000, and the next instruction is taken from that location.

BRANCH ON OVERFLOW (31) ± 0000 BOF aaaa

1. "If the overflow indicator is on, turn it off and transfer control to location aaaa; take the next instruction from location aaaa." (If the overflow indicator is off, control continues in sequence.)
2. Sequential operation resumes after the branch.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Example:

Location	*Instruction*
1000	0 0000 CAD 2225
1001	0 0000 ADD 1500
1002	0 0000 BOF 3000
1003	0 0000 STA 1550
1004	0 0000 HLT 1111

On the BOF instruction, if adding the contents of location 1500 to the A register produces overflow, transfer control to location 3000. If overflow does not occur, continue in sequence; take-the next instruction from location 1003.

SAMPLE PROBLEM

INPUT

Using punched paper tape as the input medium, read the data words A, B, C, and D into locations 0250 through 0253. Read the instructions into locations 0500 through 0510.

PROCESSING

Add A to B; check for overflow. If overflow occurs, halt the processing operation. If no overflow occurs, store the result, X, in location 0254. Then add C and D to the sum in the A register. Store the final result, Y, in location 0255.

OUTPUT

Use the character-at-a-time printer to print out the final result. Refer to Table 4-1 for solution to problem.

Table 4-1.
Solution to Sample Problem.

Storage Location	Program on Paper Tape	Remarks
	6 1000 PRB 0250	Control instruction: read data into core storage.
0250	0 2000 76 5431	Quantity A.
0251	1 1200 80 0000	Quantity B.
0252	0 0000 00 0008	Quantity C.
0253	0 0088 00 9999	Quantity D.
0254	0 0000 00 0000	Sum X.
0255	0 0000 00 0000	Sum Y.
	6 1000 PRB 0500	Control instruction: read instructions into core storage.
0500	0 0000 CAD 0250	A rA (Quantity A sent to A Register)
0501	0 0000 ADD 0251	A + B
0502	0 0000 BOF 0505	If overflow, halt.
0503	0 0000 STA 0254	No overflow. Store X.
0504	0 0000 BUN 0506	To C and D routine.
0505	0 0000 HLT 0001	Overflow halt.
0506	0 0000 ADD 0252	(A + B) + C
0507	0 0000 ADD 0253	(A + B + C) + D
0508	0 0000 STA 0255	Store Y.
0509	0 1010 PWR 0255	Print out final result Y.
0510	0 0000 HLT 0002	Program complete; halt operation.
	6 0000 BUN 0500	Control instruction: data and instructions read in; transfer control to first instruction to be executed.

5

Rearranging Information for Computation

SHIFTING

The coder will often find it useful to rearrange the contents of the A register and the R register, or both. The Burroughs 220 provides shifting instructions for this purpose, which allow the coder to move the digits contained in these registers to the left or to the right of their initial register positions.

SHIFT RIGHT A (48) ± 0000 SRA 00nn

1. "Shift the contents of the A register, excluding the contents of the sign-digit position, nn digit positions to the right."
2. Digits shifted out of the low-order end of the A register are lost; as each digit leaves the A register, a 0 enters digit position 1 of the A register, until the register is filled from the left with nn 0's.
3. The R register is not affected by this instruction.
4. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SRA Instruction

A Register	*R Register*
1 7133 41 9821	1 4792 06 4910

Contents of A and R Registers
After Execution of SRA Instruction

Instruction	*A Register*	*R Register*
0 0000 SRA 0006	1 0000 00 7133	1 4792 06 4910
0 0000 SRA 0010	1 0000 00 0000	1 4792 06 4910

SHIFT LEFT A (49) ± 0000 SLA 00nn

1. "Shift the contents of the A register, excluding the contents of the sign-digit position, nn digit positions to the left."
2. This is a circulating shift; as each digit is shifted out of digit position 1 of the A register, it enters the low-order digit position of the A register.
3. The R register is not affected by this instruction.
4. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SLA Instruction

A Register	*R Register*
1 7144 51 9821	1 4692 06 4910

Contents of A and R Registers
After Execution of SLA Instruction

Instruction	*A Register*	*R Register*
0 0000 SLA 0006	1 9821 71 4451	1 4792 06 4910
0 0000 SLA 0010	1 7144 51 9821	1 4792 06 4910

SHIFT RIGHT A AND R (48) ± 0001 SRT 00nn

1. "Shift the contents of the A register and the R register, excluding the contents of the sign-digit positions of both registers, nn digit positions to the right."
2. When using this instruction, the A and R registers are considered as one register holding a 20-digit number.
3. This is not a circulating shift; digits shifted out of the low-order digit position of the R register are lost. As each digit leaves the right end of the R register, a 0 enters digit position 1 of the A register, until the registers are filled from the left with nn 0's. The number of digits to be shifted (nn) must always be 19 or less.
4. The contents of the sign-digit positions of the A and R registers are not shifted with the other digits during the execution of this instruction. The sign digit of the R register is replaced by the sign digit of the A register; the sign digit of the A register remains the same.
5. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SRT Instruction

A Register	*R Register*
1 8376 63 7429	0 4125 78 6439

Contents of A and R Registers
After Execution of SRT Instruction

Instruction	*A Register*	*R Register*
0 0001 SRT 0004	1 0000 83 7663	1 7429 41 2578
0 0001 SRT 0010	1 0000 00 0000	1 8376 63 7429

SHIFT LEFT A AND R (49) ± 0001 SLT 00nn

1. "Shift the contents of the A register and the R register, excluding the contents of the sign-digit positions of both registers, nn digit positions to the left."
2. When using this instruction, the A and R registers are considered as one register holding a 20-digit number.

3. This is a circulating shift; as each digit is shifted out of digit position 1 in the A register, it enters the low-order digit position of the R register.
4. The contents of the sign-digit positions of the A and R registers are not shifted with the other digits. The sign digit of the A register is replaced by the sign digit of the R register; the sign of the R register remains the same.
5. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SLT Instruction

A Register	*R Register*
1 8376 63 7429	0 4125 78 6439

Contents of A and R Registers
After Execution of SLT Instruction

Instruction	*A Register*	*R Register*
0 0001 SLT 0006	0 7429 41 2578	0 6439 83 7663
0 0001 SLT 0010	0 4125 78 6439	0 8376 63 7429

SHIFT RIGHT A WITH SIGN (48)
± 0002 SRS 00nn

1. "Shift the contents of the A register, including the contents of the sign-digit position, nn digit positions to the right."
2. The execution of this instruction does not result in the recirculating of digits. Instead, digits shifted out of the low-order digit position of the A register are lost; as each digit leaves the A register, a 0 enters the A register through the sign-digit position.
3. The R register is not affected by this instruction.
4. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SRS Instruction

A Register	*R Register*
0 9823 86 7149	1 4976 20 4193

Contents of A and R Registers
After Execution of SRS Instruction

Instruction	*A Register*	*R Register*
0 0002 SRS 9005	0 0000 98 2386	1 4976 20 4193
0 0002 SRS 0010	0 0000 00 0000	1 4976 20 4193

SHIFT LEFT A WITH SIGN (49)
± 0002 SLS 00nn

1. "Shift the contents of the A register, including the contents of the sign-digit position, nn digit positions to the left."
2. This is a circulating shift; as each digit is shifted out of the sign-digit position in the A register, it enters the low-order digit position of the A register.
3. The R register is not affected by this instruction.
4. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Contents of A and R Registers
Before Execution of SLS Instruction

A Register	*R Register*
0 9823 86 7149	1 4976 20 4193

Contents of A and R Registers
After Execution of SLS Instruction

Instruction	*A Register*	*R Register*
0 0002 SLS 0005	6 7149 09 8238	1 4976 20 4193
0 0002 SLS 0010	9 0982 38 6714	1 4976 20 4193

UNPACKING

A data word may be so constructed by the coder that it contains two or more kinds of information. This technique of storing different kinds of items in a single data word is referred to as "packing" a word. Suppose that location 1520 contains 0 4736 00 1756 where: 4736 is an employee number and 1756 is the employee's hourly rate of pay ($1.756).

When the packed word is brought out of core storage, it is necessary to "unpack" or separate the various types of information before each item may be used individually.

The shifting instructions provide one means by which the coder may unpack a data word. By shifting the information not immediately needed into the R register, for example, he is able to isolate the specific data needed for immediate calculations. For example: location 3500 contains 0 0450 01 6000 where: 4500 denotes year-to-date charity deductions ($45.00) and 16000 the year-to-date insurance deductions ($160.00). To unpack this information, the contents of the A and R registers could be shifted right so that the A register contains only the year-to-date charity deductions (0001 SRT 0005). The R register now contains the year-to-date insurance deductions:

A Register	*R Register*
0 0000 00 4500	0 1600 00 0000

When the information in the A register—charity deductions—has been used, the A register can be cleared (set to zero). (See CLEAR A instruction later in this chapter.) The contents of the R register are shifted back into the A register:

A Register	*R Register*
0 0000 01 6000	0 0000 00 0000

Now the year-to-date insurance information is available for use by the program.

LOADING THE R REGISTER

The LOAD R instruction, generally used for temporarily storing information when the A register contains information that cannot be destroyed, can be used in unpacking a data word. With this instruction the packed word can be brought directly into the R register. Then the various types of information can be separated. Those digits representing a specific kind of information can be shifted as a unit into the A register. The remaining segments of the data word can be retained in the R register until needed.

LOAD R (41) ± 0000 LDR aaaa

1. "Replace the contents of the R register by the contents of location aaaa."
2. Execution of this instruction does not alter the contents of location aaaa.
3. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of R Register Before Execution of LDR Instruction: 0 0943 20 0055

Instruction	*Contents of Location aaaa*	*Contents of R Register After Execution of LDR Instruction*
0 0000 LDR aaaa	0 0000 01 0001	0 0000 01 0001
0 0000 LDR aaaa	1 4321 70 8963	1 4321 70 8963

EXTRACTING

Another instruction used to unpack or select specific digits of information contained in a Burroughs 220 word is the EXTRACT instruction. The characteristics of this instruction are as follows:

EXTRACT (17) ± 0000 EXT aaaa

1. "Extract specified digits from the word in the A register by changing each digit, including the sign digit, to zero, if the corresponding digit position of the word in location aaaa contains an even digit."
2. A digit in the A register remains unchanged if the digit in the corresponding digit position of the word in location aaaa is odd.
3. In most programs, when the EXTRACT instruction is used, a constant has been stored in location aaaa containing a predetermined combination of 0's and 1's. This number is referred to as an extract constant.
4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Example:

Location 1002 could contain the word 0 7463 24 9128. If the coder wants to retain only the four middle digits of the word, he could store in location 1025 the extract constant 0 0001 11 1000. The following steps accomplish the desired extraction:

Instruction	*Contents of A Register*	*Remarks*
0 0000 CAD 1002	0 7463 24 9128	Only desired digits of word remain in A register.
0 0000 EXT 1025	0 0003 24 9000	

SIGN-DIGIT MANIPULATION

There are occasions when the coder may wish to change the sign digit of a word in the A register. The following instruction provides this facility.

LOAD SIGN A (43) ± 000n LSA 0000

1. "Replace the contents of the sign-digit position of the A register by n."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Examples:

Instruction	*Contents of A Register Before Execution of Instruction*	*Contents of A Register After Execution of Instruction*
0 0001 LSA 0000	0 4321 56 0789	1 4321 56 0789
0 0000 LSA 0000	1 4321 56 0789	0 4321 56 0789

MULTIPLICATION

MULTIPLY (14) ± 0000 MUL aaaa

1. "Multiply the contents of location aaaa by the contents of the A register. Insert the 20-digit product in the A and R registers."
2. The ten low-order digits of the product replace the contents of the R register. (Note that this will destroy any information that has previously been stored in the R register.) The ten high-order digits of the product replace the contents of the A register. The sign of the product is inserted in the sign-digit positions of both the A and R registers.
3. The sign of the product is the algebraic result of the operation.
4. If the multiplier and multiplicand are not positioned properly, the high-order digits of the product could appear in the R register. (Refer to discussion of scaling.) If this occurs, the product must be shifted into the A register before it can be used in further calculations.
5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

The following examples illustrate the multiply operation, showing positioning of the product in the A register, the R register, or both.

Example 1.

Problem: Multiply the contents of location aaaa by the contents of the A register: 0 0000 03 3333

Multiplicand in Location aaaa	*Contents of the A and R Registers After the Execution of the Multiply Instruction*	
0 0000 00 0002	0 0000 00 0000	0 0000 06 6666
0 0000 02 0000	0 0000 00 0000	0 0666 66 0000
1 0200 00 0000	1 0000 06 6666	1 0000 00 0000

Example 2.

Problem: Multiply the contents of location aaaa by the contents of the A register: 1 0440 00 0000

Multiplicand in Location aaaa	*Contents of the A and R Registers After the Execution of the Multiply Instruction*	
0 0033 30 0000	0 0001 45 6200	0 0000 00 0000
1 0000 01 1111	1 0000 00 0488	1 8840 00 0000
0 1111 11 1111	0 0048 88 8888	0 8840 00 0000

ROUND (16) ± 0000 RND 0000

1. "Increase the absolute value of the number in the A register by + 0000 00 0001, if the high-order digit of the R register is 5 or greater, and clear the R register. If the high-order digit of the R register is less than 5, leave the A register unaltered and clear the R register."
2. Overflow is possible if the A register contains all 9's before the execution of the instruction and the high-order digit of the R register is 5 or greater.
3. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Example:

Multiply 1.432 by 43. Round the product to two decimal places.
Assume: location 2001 contains 0 0000 00 1432
location 2010 contains 0 0000 00 0043

Instructions	*A Register*	*R Register*
0 0000 CAD 2010	0 0000 00 0043	0 0000 00 0000
0 0000 MUL 2001	0 0000 00 0000	0 0000 06 1576
0 0001 SLT 0009	0 0000 00 6157	0 6000 00 0000
0 0000 RND 0000	0 0000 00 6158	0 0000 00 0000
0 0000 HLT 0000	0 0000 00 6158	0 0000 00 0000

Product: 61.58

CLEARING REGISTERS AND LOCATIONS

There are many occasions when the coder wants to clear (set to zero) the A or the R register (or both), or a storage location. The following instructions perform the clearing function.

CLEAR A (45) ± 0001 CLA 0000

1. "Replace every digit, including the sign digit, in the A register with 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Example:

Contents of A and R Registers
Before Execution of CLA Instruction

A Register	*R Register*
1 7432 89 6577	1 4571 22 1665

Contents of A and R Registers
After Execution of CLA Instruction

Instruction	*A Register*	*R Register*
0 0001 CLA 0000	0 0000 00 0000	1 4571 22 1665

CLEAR R (45) ± 0002 CLR 0000

1. "Replace every digit, including the sign digit, in the R register with 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Example:

Contents of A and R Registers
Before Execution of CLR Instruction

A Register	*R Register*
1 7432 89 6577	1 4571 22 1665

Contents of A and R Registers
After Execution of CLR Instruction

Instruction	*A Register*	*R Register*
0 0002 CLR 0000	1 7432 89 6577	0 0000 00 0000

CLEAR A AND R (45) ± 0003 CAR 0000

1. "Replace every digit, including the sign digit, of the A and R registers with 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

Example:

Contents of A and R Registers
Before Execution of CAR Instruction

A Register	*R Register*
1 7432 89 6577	1 4571 22 1665

Contents of A and R Registers
After Execution of CAR Instruction

Instruction	*A Register*	*R Register*
0 0003 CAR 0000	0 0000 00 0000	0 0000 00 0000

CLEAR LOCATION (46) ± 0000 CLL aaaa

1. "Replace every digit, including the sign digit, of location aaaa with 0."
2. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Example:

Contents of location 2250 before execution of CLL instruction: 1 4376 52 8997

Contents of Location 2250
After Execution of CLL Instruction

Instruction	*Location 2250*
0 0000 CLL 2250	0 0000 00 0000

DIVISION

DIVIDE (15) ± 0000 DIV aaaa

1. "Divide the 20-digit contents of the combined A and R registers by the contents of location aaaa."
2. If the absolute value of the divisor (the number in location aaaa) is greater than the absolute value of the portion of the dividend in the A register, the division is performed. Upon completion of the divide operation, the A register will contain the 10-digit quotient and the R register will contain the true (undivided) remainder. The sign of the quotient will be the algebraic result of the operation. The sign of the remainder will be the same as the sign of the dividend.

3. If the absolute value of the divisor is less than or equal to the absolute value of the portion of the dividend in the A register, division will not occur because the result would exceed the capacity of the A register. Instead, the overflow indicator will be turned on, and the execution of the instruction will terminate, leaving the contents of the A and R registers unaltered.
4. Since the dividend is considered by the computer to be a 20-digit number contained in the A and R registers, the R register must be cleared before a divide operation, if the 10 low-order digits of the problem dividend are zeros.
5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Example 1.

Location 1000 contains 0 0000 00 0300
Location 1001 contains 1 0000 00 0064

Instructions	*A Register*	*R Register*
0 0000 CAD 1001	1 0000 00 0064	0 0000 00 0000
0 0000 DIV 1000	1 2133 33 3333	1 0000 00 0100

The answer will be—.2133 33 3333, with an undivided remainder of 100.

Example 2.

Location 0500 contains 1 0000 00 0002
Location 0501 contains 0 0000 00 0004

Instructions	*A Register*	*R Register*
0 0000 CAD 0501	0 0000 00 0004	0 0000 00 0000
0 0000 DIV 0500	0 0000 00 0004	0 0000 00 0000

Dividing 4 by 2 produces the number 2, which exceeds the capacity of the A register (see following section on scaling). Therefore: the overflow indicator is turned on. Execution of the instruction is terminated.

SCALING

In fixed-point arithmetic[1], the operations of addition, subtraction, division, and multiplication are performed as if the numbers in the Burroughs 220 had a decimal point—the so-called machine decimal point—in a fixed location in a word. The machine decimal point is located between the sign-digit position and the high-order digit position of each word.

Because the machine decimal point and the problem decimal point seldom coincide, the coder must keep track of the problem decimal point during arithmetic operations. As an example of this, a word will be processed by the computer as 0 0014 63 4210 even though to the coder it may represent the value 1463.4210. This problem of accounting for the decimal point in an arithmetic result is referred to as bookkeeping the decimal point, or scaling.

To help the coder determine the location of the problem decimal point with respect to the machine decimal point, the concept of scale factor has been developed. This concept states that:

1. If the problem decimal point is located to the right of the machine decimal point, the number is said to have a negative scale factor.
2. If the problem decimal point is located to the left of the machine decimal point, the number is said to have a positive scale factor.
3. The value of the scale factor is equal to the number of digit positions between the machine decimal point and the problem decimal point.

Examples (where the machine decimal point is represented ‸):

+ ‸ 123.450000[2] has a scale factor of −3.
− ‸ 000123.4500 has a scale factor of −6.
+1234.5 ‸ 0000 00 0000 has a scale factor of +1.
.12345 ‸ 0000 00 0000 has a scale factor of +5.

In keeping with the scale factor concept, the following rules are suggested to help the coder keep track of the decimal point.

RULE 1

In addition and subtraction, the number in the A register (first operand) and the number in core storage (second operand) must have the same scale factor. This is a familiar practice in everyday arithmetic: lining up the decimal points before adding two numbers or subtracting one number from another number. For example:

First Operand	*Second Operand*	*Remarks*
0‸0000 14 3.291	0‸0000 00 5.371	Each has a scale factor of −7.
0‸0045.00 0000	0‸0017.00 0000	Each has a scale factor of −4.
0‸46.13 00 0000	0‸15.54 00 0000	Each has a scale factor of −2.

RULE 2

In multiplication, the scale factor of the product is the sum of the scale factor of the multiplier and the scale factor of the multiplicand. For example:

Multiplier	*Multiplicand*
0‸12.00 00 0000	0‸12.00 00 0000
0‸13.00 00 0000	0‸0000 00 013.0

Product	
0‸0144.00 0000	0 0000 00 0000
0‸0000 00 0016	0 9.000 00 0000

[1]The Burroughs 220 also provides for floating-point arithmetic to keep track of the problem decimal point automatically. See Chapter 10.

[2]The machine decimal point will be denoted by the symbol ‸ ; the problem decimal point will be denoted by the conventional period.

RULE 3

In division, the scale factor of the divisor is subtracted from the scale factor of the dividend to obtain the scale factor of the quotient. For example:

```
                    Dividend
0‸0000 16.9000              0 0000 00 0000
0‸0000 00 0016              0 9.000 00 0000

      Divisor                     Quotient
0‸0013.00  0000             0‸01.30 00 0000
0‸013.0 00 0000             0‸0000 00 13.00
```

As a final example, suppose that it is desired to calculate an employee's gross earnings during a pay period. It is known that his rate is in the form r.rr, and that hours worked are in the form hh.h. It is desired to calculate gross earnings rounded to the nearest cent. The problem is to store "rate" and "hours worked" to produce the product in the desired form. Thus:

pay = rate × hours
= + 000r.rr 0000 × 00hh. h0 0000
= + 0000 00 pp.pp p000 00 0000
(a scale factor of −8)

The desired result (obtaining a scale factor of −8) could also be achieved in other ways; for example:

+ 0000 r.r r000 × + 0hh.h 00 0000

as well as

+ 0r.rr 00 0000 × + 0000 hh. h000

yield the same result.

SAMPLE PROBLEMS

Refer to Tables 5-1 and 5-2 for sample scaling problems.

Table 5-1

Sample Scaling Problem No. 1

Location	Contents	Remarks
2001	0 000X X.X X000	Rate.
2002	0 000X X.X X000	Hours worked.

Given information above, find gross earnings to nearest cent. Store result in location 2003. Begin program in location 1000.

Instruction			Scale Factor
First Method			
1000	CAD	2001	−5
1001	MUL	2002	−10
1002	SLT	0002	−8
1003	RND	0000	−8
1004	STA	2003	−8
Second Method			
1000	CAD	2001	−5
1001	SLA	0002	−3
1002	MUL	2002	−8
1003	RND	0000	−8
1004	STA	2003	−8

Result: Location 2003 contains 0 0000 0(X) XX.XX, gross earnings.

Table 5-2

Sample Scaling Problem No. 2

Location	Contents	Scale Factor
1000	90	−10
1001	82	−10
1002	93	−10
1003	65	−10
1004	75	−10
1005	80	−10
3000	6	−2

Given the information above, find the average, correct to one decimal, of the six numbers stored in locations 1000 through 1005. Store the answer in location 1006. Start the program in location 2000.

Instruction			Remarks
2000	CAR	0000	0→A and R registers.
2001	CAD	1000	First number→rA.
2002	ADD	1001	Add remaining five numbers.
2003	ADD	1002	
2004	ADD	1003	
2005	ADD	1004	
2006	ADD	1005	
2007	DIV	3000	Find average.
2008	SRT	0001	Scale factor of −9.
2009	RND	0000	Round to one decimal.
2010	STA	1006	Store result.
2011	HLT	0000	Halt operation.

6

Using the Burroughs 220 to Make Decisions

STEPS IN MAKING DECISIONS

One of the frequent clerical functions necessary in the manual processing of data is to sort information by some characteristic before further processing.

This is the case when an accounts payable clerk looks at each of a stack of vendor invoices to find those that must be paid on a specific date. He examines the date on each invoice, putting those with the specified date aside. When all of the invoices have been examined, those put aside may be processed further and checks prepared for the vendors represented by the invoices.

This task, like other sorting tasks, can be broken down into a sequence of steps. Let us consider the example of a clerk who must examine three numbers to determine which is the smallest.

Assuming that no two of the three numbers are equal, the steps for this job might be as follows:

1. He looks at the first number.
2. He then looks at the second number.
3. He remembers the first number and decides which of the numbers is the smaller. He disregards the larger.
4. He looks at the third number.
5. He remembers the smaller number from the first comparison, and decides which of the numbers now being compared is the smaller. He disregards the larger.
6. He is then able to post the smallest number to a special report.

This process is graphically portrayed in the flow chart shown in Fig. 6-1.

The reader will note from the sequence of steps illustrated that the basic operation is twofold:

1. Examining data for specific properties;
2. Deciding what action to take as the result of the examination; i.e., performing an operation if the specific properties are present, or rejecting the original data and examining another piece of data if the specific properties are absent.

The Burroughs 220 can:

1. Examine data;
2. Make a "decision" as to the course of action to be taken (as the result of the examination).

SETTING UP COMPUTER DECISIONS

The decision-making operations of the computer are predetermined by the coder. In his program, he must provide the possible alternate courses of action and the criteria to determine which alternative is to be taken. Decision-making situations can usually be reduced by the coder to a set of comparisons: one number is compared to another number and a "larger than," "equal to," or "smaller than" situation is set up within the computer depending on the results of the comparison. This provides the basis for the selection of a course of action to be taken. The coder may so code his program that the computer takes the proper course of action for any of the possible results of the comparison.

If the problem of examining three numbers to determine the smallest were applied to the computer, the coder might write his program as follows:

Step in Program	*Instruction To Computer*
Step 1	Compare first number with second number.
Step 2	Interrogate computer. If first number smaller, go to next step. If second number smaller, go to step 9.
Step 3	Compare first number with third number.
Step 4	If first number smaller, go to next step. If third number smaller, go to step 7.
Step 5	Store first number as smallest number.
Step 6	Halt computer operation.
Step 7	Store third number as smallest number.
Step 8	Halt computer operation.
Step 9	Compare second number with third number.
Step 10	If second number smaller, go to next step. If third number smaller, go back to step 7.
Step 11	Store second number as smallest number.
Step 12	Halt computer operation.

The reader will note that the computer may or may not go through all the steps listed. And it may or may not take the steps in the order listed. This illustrates the principle that the coder must always foresee every alternative that may be required. See Fig. 6-2 for a flow chart of this program.

The coder may use the decision-making ability of the computer to select particular items from extremely large volumes of data in order to perform specific operations on

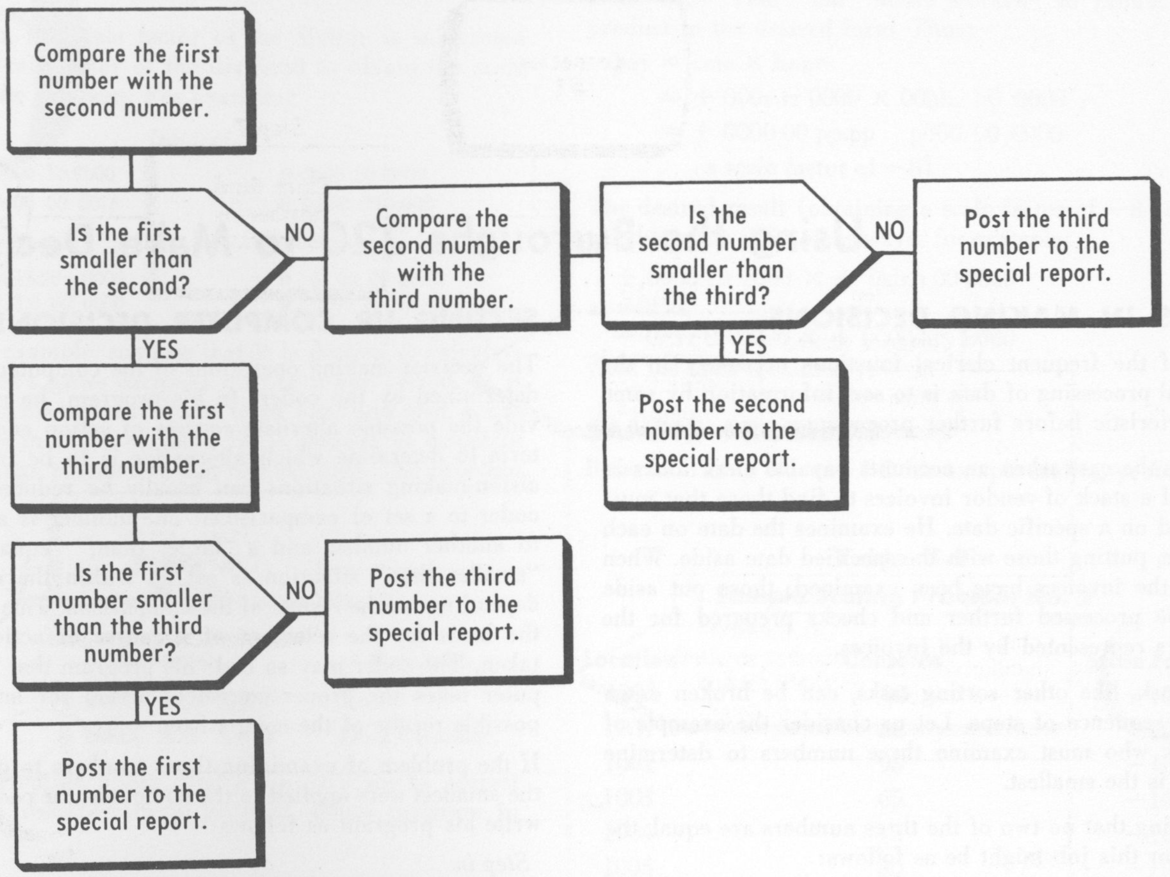

Compare the first number with the second number.
Is the first smaller than the second?
NO
Compare the second number with the third number.
Is the second number smaller than the third?
NO
Post the third number to the special report.
YES
Post the second number to the special report.
YES
Compare the first number with the third number.
Is the first number smaller than the third number?
NO
Post the third number to the special report.
YES
Post the first number to the special report.

Figure 6-1. Clerk's Decision-Making Steps

the selected items. For example, inventory balances for 2000 items could be examined by the computer to determine if any of the 2000 items has been depleted to its reorder point of 250. Those items that had reached the reorder point could be reordered automatically. This task might be accomplished by writing a program to have the computer do the following:

1. Compare a constant of 250 to each inventory balance.
2. If an *equal-to* or *greater-than* condition exists as a result of the comparison, transfer control to a reorder routine; if a *smaller-than* condition exists, examine the next inventory balance.

Thus the computer can automatically select one of several program paths without stopping or without the necessity of additional instructions.

DECISION-MAKING OPERATIONS OF THE BURROUGHS 220

When a decision-making operation is required in a program, the coder must first think of setting up a decision-making condition within the computer. This usually consists of a comparison of two words, two fields, or two digits. Based upon the result of such a comparison, a transfer of program control may occur. For example: if the comparison shows the two quantities compared to be equal, a transfer may occur; or if the comparison shows one quantity to be larger than the other, a transfer may occur. A separate branching instruction or a special transfer-of-control feature built into the comparison instruction determines which sequence of instructions is to be followed in accordance with the outcome of the particular comparison in the program.

Such is the case when the contents of the A or R register are compared with a word in storage. (Either the entire words are compared or specified partial-word fields.) A transfer of control may be effected, based upon the result of the comparison.[1] For this operation, two instructions are needed: one to make the comparison and one to transfer control. One or more branching instructions following the comparison in the program are specified to cause a transfer of control on certain results of the comparison. On any result other than the ones specified, the branching instruction causes program control to continue in sequence.

There are other methods of incorporating a decision-making operation into a program. One of these methods requires the use of a tally. This method can be used when the number of times a loop must be repeated is known in advance. (A loop is a repetition of part of a program.)

[1]The results of the comparison may be observed on the comparison indicator, on the Console.

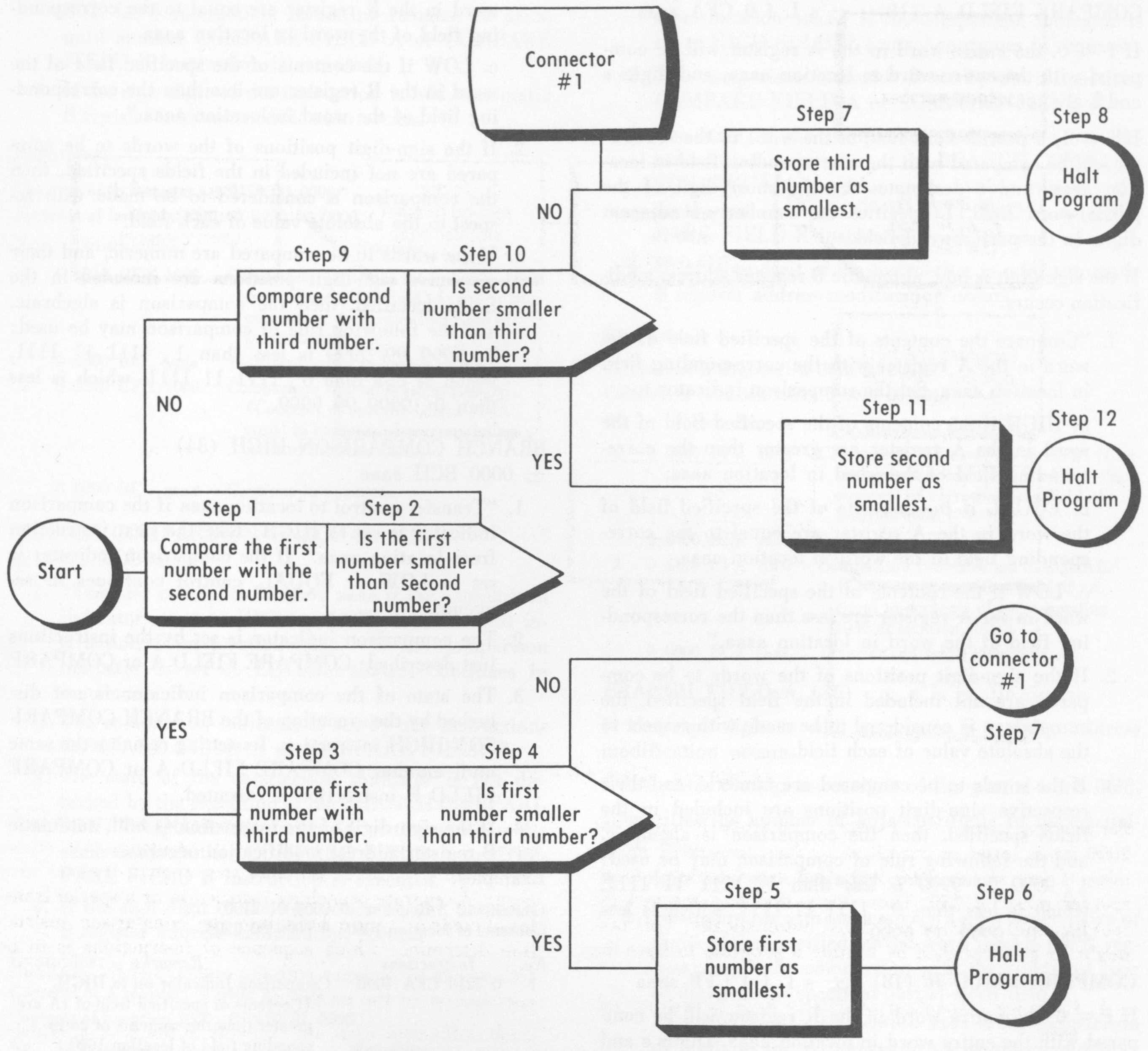

Connector #1
Step 7
Store third number as smallest.
Step 8
Halt Program
NO
Step 9
Compare second number with third number.
Step 10
Is second number smaller than third number?
NO
Step 11
Store second number as smallest.
Step 12
Halt Program
YES
Step 1
Compare the first number with the second number.
Step 2
Is the first number smaller than second number?
Start
Go to connector #1
Step 7
NO
YES
Step 3
Compare first number with third number.
Step 4
Is first number smaller than third number?
Step 5
Store first number as smallest.
Step 6
Halt Program
YES

Figure 6-2. Computer Decision-Making Steps

At the beginning of each repetition, a tally (number) in storage can be reduced or increased by 1, and then tested by a conditional-transfer instruction. The tally is selected so that when the final iteration has been performed, a predetermined condition will exist in the computer—as a result of the final increase or decrease of the tally—causing the conditional-transfer instruction to transfer program control. (An example of the predetermined condition mentioned above is overflow; an example of a conditional-transfer instruction is BRANCH ON OVERFLOW. Overflow and BRANCH ON OVERFLOW are described in Chapter 4.)

Two other methods require only a single instruction for the decision-making operation. These methods are examples of the special transfer-of-control feature built into a comparison instruction.

The sign digit of a word in the A register may be compared with a specified digit of the instruction being executed. When the digits compared are equal, control is transferred to a program path designated by the address portion of the instruction.

A transfer of control may also occur with a single instruction when one or more adjacent digits in a specified field of the A or R register are compared with one or two digits specified by the instruction being executed. As with the previous comparison, if the digits compared are equal, program control is transferred to a program path designated by the address portion of the instruction.

In either case, when the numbers compared are not equal, the program continues in sequence. The following Burroughs 220 instructions perform the operations described.

COMPARE FIELD A (18) ± s L f 0 CFA aaaa

If f = 0, the entire word in the A register will be compared with the entire word in location aaaa, and digits s and L are not relevant.

If f = 1, a partial-word field of the word in the A register will be compared with the corresponding field in location aaaa, and s designates the rightmost digit of the partial-word field. L specifies the number of adjacent digits in the partial-word field.

If the sign digit is odd, automatic B register address modification occurs.

1. "Compare the contents of the specified field of the word in the A register with the corresponding field in location aaaa. Set the comparison indicator to:

   a. HIGH if the contents of the specified field of the word in the A register are greater than the corresponding field of the word in location aaaa.

   b. EQUAL if the contents of the specified field of the word in the A register are equal to the corresponding field of the word in location aaaa.

   c. LOW if the contents of the specified field of the word in the A register are less than the corresponding field of the word in location aaaa."
2. If the sign-digit positions of the words to be compared are not included in the field specified, the comparison is considered to be made with respect to the absolute value of each field.
3. If the words to be compared are numeric, and their respective sign-digit positions are included in the fields specified, then the comparison is algebraic, and the following rule of comparison may be used: 1 ‸ 9999 99 9999 is less than 1 ‸ 1111 11 1111, which is less than 0 ‸ 1111 11 1111, which is less than 0 ‸ 9999 99 9999.

COMPARE FIELD R (18) ± s L f 1 CFR aaaa

If f = 0, the entire word in the R register will be compared with the entire word in location aaaa. Digits s and L are not relevant.

If f = 1, a partial-word field of the word in the R register will be compared with the corresponding field in location aaaa, and s designates the rightmost digit of the partial-word field. L specifies the number of adjacent digits in the partial-word field.

If the sign digit is odd, automatic B register address modification occurs.

1. "Compare the contents of the specified field of the word in the R register with the corresponding field of the word in location aaaa. Set the comparison indicator to:

   a. HIGH if the contents of the specified field of the word in the R register are greater than the corresponding field of the word in location aaaa.

   b. EQUAL if the contents of the specified field of the word in the R register are equal to the corresponding field of the word in location aaaa.

   c. LOW if the contents of the specified field of the word in the R register are less than the corresponding field of the word in location aaaa."
2. If the sign-digit positions of the words to be compared are not included in the fields specified, then the comparison is considered to be made with respect to the absolute value of each field.
3. If the words to be compared are numeric, and their respective sign-digit positions are included in the fields specified, then the comparison is algebraic, and the following rule of comparison may be used: 1 ‸ 9999 99 9999 is less than 1 ‸ 1111 11 1111, which is less than 0 ‸ 1111 11 1111, which is less than 0 ‸ 9999 99 9999.

BRANCH COMPARISON HIGH (34)
± 0000 BCH aaaa

1. "Transfer control to location aaaa if the comparison indicator is set to HIGH; take the next instruction from location aaaa." If the comparison indicator is set to LOW or EQUAL, control continues in sequence.
2. The comparison indicator is set by the instructions just described: COMPARE FIELD A or COMPARE
3. The state of the comparison indicator is not disturbed by the execution of the BRANCH COMPARISON HIGH instruction. Its setting remains the same until another COMPARE FIELD A or COMPARE FIELD R instruction is executed.
4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register: 0 0000 01 2100
Contents of location 1000: 0 0000 00 3200

No.	*Instructions*	*Remarks*
1	0 7210 CFA 1000	Comparison Indicator set to HIGH. (Contents of specified field of rA are greater than the contents of corresponding field of location 1000.)
	0 0000 BCH aaaa	Transfer control to location aaaa.
2	0 8210 CFA 1000	Comparison Indicator set to LOW. (Contents of specified field of rA less than contents of corresponding field of location 1000.)
	0 0000 BCH aaaa	Continue in sequence.

BRANCH COMPARISON LOW (34)
± 0001 BCL aaaa

1. "Transfer control to location aaaa if the comparison indicator is set to LOW; take the next instruction from location aaaa." If the comparison indicator is set to HIGH or EQUAL, control continues in sequence.
2. The comparison indicator is set by COMPARE FIELD A or COMPARE FIELD R.
3. The state of the comparison indicator is not disturbed by the execution of the BRANCH COMPARI-

SON LOW instruction. Its setting remains the same until another COMPARE FIELD A or COMPARE FIELD R instruction is executed.

4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register: 0 0150 00 0000
Contents of Location 1000: 0 0250 00 0000

No.	*Instructions*	*Remarks*
1	0 4310 CFA 1000	Comparison Indicator set to LOW. (Contents of specified field of rA less than contents of corresponding field of location 1000.)
	0 0000 BCL aaaa	Transfer control to location aaaa.
2	0 4210 CFA 1000	Comparison Indicator set to EQUAL. (Contents of specified field of rA equal to contents of corresponding field of location 1000.)
	0 0000 BCL aaaa	Continue in sequence.

BRANCH COMPARISON UNEQUAL (35)
± 0001 BCU aaaa

1. "Transfer control to location aaaa if the comparison indicator is set to HIGH or LOW: take the next instruction from location aaaa." If the comparison indicator is set to EQUAL, control continues in sequence.
2. The comparison indicator is set by the instructions COMPARE FIELD A or COMPARE FIELD R.
3. The state of the comparison indicator is not disturbed by the execution of the BRANCH COMPARISON UNEQUAL instruction. Its setting remains the same until another COMPARE FIELD A or COMPARE FIELD R instruction is executed.
4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A register: 0 0402 31 0800
Contents of Location 1000: 0 0180 31 0200

No.	*Instructions*	*Remarks*
1	0 2210 CFA 1000	Comparison Indicator set to HIGH. (Contents of specified field of rA greater than contents of corresponding field of location 1000.)
	0 0000 BCU aaaa	Transfer control to location aaaa.
2	0 6210 CFA 1000	Comparison Indicator set to EQUAL. (Contents of specified field of rA equal to contents of corresponding field of location 1000.)
	0 0000 BCU aaaa	Continue in sequence.
3	0 4210 CFA 1000	Comparison Indicator set to LOW. (Contents of specified field of rA less than contents of corresponding field of location 1000.)
	0 0000 BCU aaaa	Transfer control to location aaaa.

BRANCH COMPARISON EQUAL (35)
± 0000 BCE aaaa

1. "Transfer control to location aaaa if the comparison indicator is set to EQUAL: take the next instruction from location aaaa." If the comparison indicator is set to HIGH or LOW, control continues in sequence.
2. The comparison indicator is set by the instructions COMPARE FIELD A or COMPARE FIELD R.
3. The state of the comparison indicator is not disturbed by the execution of the BRANCH COMPARISON EQUAL instruction. Its setting remains the same until another COMPARE FIELD A or COMPARE FIELD R instruction is executed.
4. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Contents of A Register: 0 0010 00 0001
Contents of Location 1000: 0 0010 00 0010

No.	*Instruction*	*Remarks*
1	0 0410 CFA 1000	Comparison Indicator set to LOW. (Contents of specified field of rA less than contents of corresponding field of location 1000.)
	0 0000 BCE aaaa	Continue in sequence.
2	0 4410 CFA 1000	Comparison Indicator set to EQUAL. (Contents of specified field of rA equal to contents of corresponding field of location 1000.)
	0 0000 BCE aaaa	Transfer control to location aaaa.

BRANCH FIELD A (36) ± s L nn BFA aaaa

If the sign digit is odd, automatic B register address modification occurs.

s designates the rightmost digit of the partial-word field.

L specifies the number of adjacent digits in the partial-word field.

nn: digits used as a basis for comparison.

1. "Compare nn with successively higher-order pairs of digits of the specified partial-word field in the A register; begin the comparison with the rightmost pair of digits of the specified partial-word field.

   "Transfer control to location aaaa if the comparison of digits in the A register to nn produces equality; take the next instruction from location aaaa." Control continues in sequence if any digit comparison produces inequality.
2. The initial comparison concerns the digit in the A register specified by s. Subsequent comparisons concern digits to the left of s. The order of comparison is as follows:

   a. The low-order digit of nn is compared with the low-order digit of the partial-word field.

   b. The high-order digit of nn is compared with the next higher-order digit of the partial-word field.

   c. The low-order digit of nn is compared with the next higher-order digit of the partial-word field, and so forth.
3. If s and L both equal 0, nn will be compared with the five adjacent pairs of digits of the A register; the sign digit will not be included in the comparison.

4. If sL specifies a one-digit field, the digit specified will be compared with the low-order digit of nn.
5. If sL specifies an odd field length, the odd digit—the high-order digit—of the field specified will be compared with the low-order digit of nn.

Examples:

Instruction	*Contents of A Register*	*Remarks*
0 0000 BFA aaaa	1 0000 00 0000	Transfer control to location aaaa.
0 0000 BFA aaaa	1 0000 00 1000	Continue in sequence.
0 0499 BFA aaaa	0 0000 00 9999	Transfer control to location aaaa.
0 1210 BFA aaaa	1 0000 00 0000	Transfer control to location aaaa.

BRANCH FIELD R (37) ± s L nn BFR aaaa

If the sign digit is odd, automatic B register address modification occurs.

s designates the rightmost digit of the partial-word field.

L specifies the number of adjacent digits in the partial-word field.

nn: digits used as a basis for comparison.

1. "Compare nn with successively higher-order pairs of digits of the specified partial-word field in the R register; begin the comparison with the rightmost pair of digits of the specified partial-word field.
   "Transfer control to location aaaa if the alternate comparison of digits in the R register to nn produces equality; take the next instruction from location aaaa." Control continues in sequence if any digit comparison produces inequality.
2. The initial comparison concerns the digit in the R register specified by s. Subsequent comparisons concern digits to the left of s. The order of comparison is as follows:
   a. The low-order digit of nn is compared with the low-order digit of the partial-word field.
   b. The high-order digit of nn is compared with the next higher-order digit of the partial-word field.
   c. The low-order digit of nn is compared with the next higher-order digit of the partial-word field, and so forth.
3. If s and L both equal 0, nn will be compared with the five adjacent pairs of digits of the R register; the sign digit will not be included in the comparison.
4. If sL specifies a one-digit field, the digit specified will be compared with the low-order digit of nn.
5. If sL specifies an odd field length, the odd digit—the high-order digit—of the field specified will be compared with the low-order digit of nn.

Examples:

Instruction	*Contents of R Register*	*Remarks*
0 0400 BFR aaaa	0 1234 56 1000	Continue in sequence.
0 6264 BFR aaaa	0 0000 64 2050	Transfer control to location aaaa.
0 3301 BFR aaaa	0 1010 47 1051	Transfer control to location aaaa.

BRANCH SIGN A (33) ± 000n BSA aaaa

1. "Compare the sign digit of the word in the A register with n. Transfer control to location aaaa if the comparison produces equality; take the next instruction from location aaaa." If the comparison produces inequality, control continues in sequence.
2. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Instruction	*Contents of A Register*	*Remarks*
0 0001 BSA aaaa	1 1234 56 7890	Transfer control to location aaaa.
0 0000 BSA aaaa	1 1234 56 7890	Continue in sequence.

SAMPLE PROBLEMS

1. Compare the automobile part description found in location 2000 and the six high-order digit positions of location 2001 with the corresponding digit positions of the words in locations 3000 and 3001. If any digits compared are unequal, halt the operation; if all are equal, transfer program control to a routine beginning in location 1000. Start the program in location 3050.

Location	*Instruction*
3050	0 0000 CAD 2000
3051	0 0000 CFA 3000
3052	0 0001 BCU 3057
3053	0 0000 CAD 2001
3054	0 6610 CFA 3001
3055	0 0001 BCU 3057
3056	0 0000 BUN 1000
3057	0 0000 HLT 0000

2. Compare the two numbers in locations 1000 and 1050.
   If 1000 is greater than 1050, transfer control to 3000.
   If 1000 is less than 1050, transfer control to 3050.
   If 1000 is equal to 1050, and they are equal to zero, transfer control to 3060.
   If 1000 is equal to 1050, and they are not equal to zero, transfer control to 3010.
   Start the program in location 2000.

Location	*Instruction*
2000	0 0000 CAD 1000
2001	0 0000 CFA 1050
2002	0 0000 BCH 3000
2003	0 0001 BCL 3050
2004	0 0000 BFA 3060
2005	0 0000 BUN 3010

7 Automatic Repetition of Program Segments

LOOPING

Suppose the coder wants to sum 12 numbers stored in 12 consecutive storage locations. One way to do this would be to bring the first number into the A register and follow this with 11 ADD instructions—one for each of the remaining numbers to be added. For example, assuming that the 12 numbers to be added are stored in locations 1000 through 1011, the program would appear as follows:

```
2050   0 0000 CAD 1000    First number→rA.
2051   0 0000 ADD 1001  ┐
2052   0 0000 ADD 1002  │
2053   0 0000 ADD 1003  │
2054   0 0000 ADD 1004  │
2055   0 0000 ADD 1005  │
2056   0 0000 ADD 1006  ├ Sum remaining numbers.
2057   0 0000 ADD 1007  │
2058   0 0000 ADD 1008  │
2059   0 0000 ADD 1009  │
2060   0 0000 ADD 1010  │
2061   0 0000 ADD 1011  ┘
2062   0 0000 HLT 0000    Halt operation.
```

The reader will notice that the 11 ADD instructions differ in one respect only: the low-order digit of each address increases by one. Therefore, by using the first ADD instruction 11 times, and increasing the address by one each time, the same result may be obtained.

This way of summing the 12 numbers is called looping. A loop is a segment of a program which is repeated several times: often the address portions of some of the instructions are altered between repetitions. Using a loop for the example above, we would bring the first of the 12 numbers into the A register and follow this by a single ADD instruction, the address of which is increased by one before each repetition. For example:

```
2050   0 0000 CAD 1000  ┐
2051   0 0000 ADD 1001  ├ Add 12 numbers.
2052   0 0000 STA 1000  ┘
2053   0 0000 CAD 2051  ┐
2054   0 0000 ADD 2057  ├ Increase address by 1.
2055   0 0000 STA 2051  ┘
2056   0 0000 BUN 2050    Return for next iteration.
2057   0 0000  00 0001    Constant for increasing
                          address.
```

ADDRESS MODIFICATION

Alteration of the address part of an instruction is called address modification. The address part of an instruction is altered so that the instruction will reference a different location in storage each time it is executed. This is possible because a stored-program computer can perform arithmetic operations on instructions as well as data.

As explained in Chapter 3, data and instructions are stored in the same way. Instructions are recognized as instructions only when selected and interpreted as such by the control unit. A word interpreted as an instruction at one point in a program may be processed by other instructions—just as if it were a data word—at another point in the same program. Thus the computer can be coded to alter its own instructions.

LOOP TESTING

The example of adding 12 numbers using address modification within a loop makes it apparent that some way must be devised to tell the machine when to stop. The program as outlined would not stop after the contents of the specified 12 locations had been summed; the computer would continue to sum the contents of all the remaining locations in core storage.

Therefore, an exit must be provided for every loop. A record must be kept of the number of times the loop has been executed, so that the loop can be terminated after the last desired iteration. This can be done in any of several ways.

INCREASING A TALLY

Add 1 to the contents of a location reserved for tallying each time the loop is executed; compare the tally each time through the loop to a constant that is equal to the desired number of iterations. When the comparison is equal, a branch instruction transfers control from the loop. For example, suppose a coder wishes a particular loop to be repeated 20 times. Following is a sample loop tally and exit routine:

```
1000   0 0000 CAD 2050  ┐
1001                    │
. . .                   │
. . .                   ├ Main part of loop.
. . .                   │
. . .                   │
1010                    ┘
```

```
1011   0 0000 CAD 1018 )
1012   0 0000 ADD 1020 }  Increase tally.
1013   0 0000 STA 1018 )
1014   0 0000 CFA 1019    Test for last iteration.
1015   0 0000 BCE 1017    Exit after last iteration.
1016   0 0000 BUN 1000    Return for next iteration.
1017   0 0000 HLT 0000    Halt operation.
1018   0 0000  00 0000    Tally.
1019   0 0000  00 0020    Constant.
1020   0 0000  00 0001    Constant.
```

DECREASING A TALLY

Subtract 1 each time through the loop from a tally that is initially equal to the desired number of iterations; test for zero condition. When the tally equals zero, the iterations are completed and a branch instruction transfers control from the loop. For example, suppose a coder wishes to repeat a loop 20 times. Following is another sample loop tally and exit routine.

```
1000   0 0000 CAD 2050 )
1001                   |
. . .                  |
. . .                  |
. . .                  |
. . .                  }  Main part of loop.
. . .                  |
. . .                  |
. . .                  |
. . .                  |
1010                   )
1011   0 0000 CAD 1017 )
1012   0 0000 SUB 1018 }  Decrease tally.
1013   0 0000 STA 1017 )
1014   0 0000 BFA 1016    Exit after last iteration.
1015   0 0000 BUN 1000    Return for next iteration.
1016   0 0000 HLT 0000    Halt operation.
1017   0 0000  00 0020    Tally.
1018   0 0000  00 0001    Constant.
```

INCREASING A TALLY TO CAUSE OVERFLOW

Each time through a loop, add 1 to a word reserved for tallying. The word is chosen so that overflow occurs on the last iteration. The exit instruction, BRANCH ON OVERFLOW, immediately follows the instruction which performs the tallying. For example, suppose a coder wishes a loop to be repeated 20 times. Following is a third sample loop tally and exit routine.

```
1000   0 0000 CAD 2050 )
1001                   |
. . .                  }  Main part of loop.
. . .                  |
. . .                  |
1010                   )
1011   0 0000 CAD 1017 }  Increase tally.
1012   0 0000 ADL 1016 }
1013   0 0000 BOF 1015    Exit after last iteration.
1014   0 0000 BUN 1000    Return for next iteration.
1015   0 0000 HLT 0000    Halt operation.
1016   0 8000  00 0000    Tally.
1017   0 0100  00 0000    Constant.
```

The INCREASE FIELD LOCATION instruction is very useful for this method of loop tallying and exiting.

DECREASING A TALLY TO CAUSE FIELD UNDERFLOW

A fourth method of tallying and loop exiting will be discussed with the DECREASE FIELD LOCATION instruction in the next section of this chapter.

TIME FOR LOOPING

It should be noted that although the use of a loop decreases the number of steps in a code, and therefore the number of storage locations required, it also increases the execution time of the code. The time is increased because the computer must execute one or several extra instructions (the loop tally and exit instructions) each time through a loop. On the other hand, if the instructions of the loop were coded in straight sequence, as our 11 ADD instructions at the beginning of this chapter, there would be no need for the extra loop tally and exit instructions.

INCREASING AND DECREASING FIELD CONTENTS

INCREASE FIELD LOCATION (26)
± sLnn IFL aaaa

If the sign digit is odd, automatic B register address modification occurs.

s designates the rightmost digit of the partial-word field.

L specifies the number of adjacent digits in the partial-word field.

nn: digits used to increase the specified field.

1. "Increase the specified partial-word field of location aaaa by nn. If overflow occurs, that is, if the result exceeds the capacity of the specified partial-word field, the overflow indicator is turned on."
2. If the sign-digit position of the word in location aaaa is included in the specified partial-word field, it does not have sign significance; instead, it has numeric significance, and is treated in the same manner as the other ten digits of the word.

Examples:

Instruction	*Contents of Location aaaa Before Execution of Instruction*	*Contents of Location aaaa After Execution of Instruction*	*Overflow Indicator*
0 0202 IFL aaaa	0 0000 00 0012	0 0000 00 0014	OFF
0 6314 IFL aaaa	0 2973 43 9216	0 2973 57 9216	OFF
0 6220 IFL aaaa	0 0002 90 2400	0 0002 10 2400	ON

In tallying a loop, it is sometimes useful to set up a tally that can be counted down. To do this, the coder must use a tally equal to the number of repetitions wanted. This tally can be counted down 1 each time through the loop, leading to an automatic exit from the loop when the count is completed. With this method, the coder can check the tally at any time and see exactly how many iterations remain to be performed.

The following instruction is very useful in this method of tallying for a loop exit, as well as in other operations such as modifying instruction and/or data words in storage.

DECREASE FIELD LOCATION (27)
± sLnn DFL aaaa

If the sign digit is odd, automatic B register address modification occurs.

s designates the rightmost digit of the partial-word field of location aaaa.

L specifies the number of adjacent digits in the partial-word field.

nn: digits used to decreased specified field.

1. "Decrease the specified partial-word field of the word in location aaaa by nn." For example:

Instruction	*Contents of Location aaaa Before Execution of Instruction*	*Contents of Location aaaa After Execution of Instruction*
0 0202 DFL aaaa	0 0000 00 0012	0 0000 00 0010
0 6240 DFL aaaa	0 2093 40 0912	0 2093 00 0912
0 0001 DFL aaaa	0 4000 00 0001	0 4000 00 0000

2. If a field of the word in location aaaa is decreased through zero, the tens complement of the true algebraic result is obtained. For example:

	Instruction	*Contents of Location aaaa Before Execution of Instruction*	*Contents of Location aaaa After Execution of Instruction*
A.	0 4405 DFL aaaa	0 0000 10 6002	0 9995 10 6002
B.	0 6212 DFL aaaa	0 0000 10 6002	0 0000 98 6002
C.	0 1203 DFL aaaa	0 1013 04 6002	9 8013 04 6002

In example A, the algebraic result of subtracting 5 from 0000 would be −5; in example B, the algebraic result of subtracting 12 from 10 would be −2. However, a special kind of arithmetic is used with the DECREASE FIELD LOCATION instruction—one without algebraic sign significance. Thus the result in example A was 9995, the tens complement of 5, and the result in example B was 98, the tens complement of 2.

In example C, the sign digit of the word in location aaaa was included in the partial-word field which was decreased through zero. With the DECREASE FIELD LOCATION instruction the sign digit does not have algebraic significance; instead it has numeric significance. For example, the digit 1 in the sign-digit position of a word would be treated as the number 1, not as specifying a negative quantity. Thus the result of subtracting 03 from 01 in example C was 98, the tens complement of 2.

3. Decreasing a specified field through zero creates a condition in the Burroughs 220 called field underflow[1].

4. If field underflow occurs, the repeat indicator[2] is turned off. If field underflow does not occur, the repeat indicator is turned on.

Examples:

Instruction	*Contents of Location aaaa Before Execution of Instruction*	*Contents of Location aaaa After Execution of Instruction*	*Repeat Indicator*
0 0202 DFL aaaa	0 0000 00 0002	0 0000 00 0000	ON
0 0201 DFL aaaa	0 0000 00 0000	0 0000 00 0099	OFF
0 4101 DFL aaaa	0 1234 56 7890	0 1233 56 7890	ON
0 6214 DFL aaaa	0 2973 43 9216	0 2973 29 9216	ON
0 6220 DFL aaaa	0 0002 10 2400	0 0002 90 2400	OFF
0 1232 DFL aaaa	0 5236 47 8888	7 3236 47 8888	OFF

BRANCH, REPEAT (32) ± 0000 BRP aaaa

1. "Transfer control to location aaaa if the repeat indicator is on; take the next instruction from location aaaa." If the repeat indicator is off, control continues in sequence.

2. The setting of the repeat indicator is not disturbed by the execution of a BRANCH, REPEAT instruction. If it is on, it stays on until a field underflow condition occurs. If it is off, it remains off until it is turned on by a DECREASE FIELD LOCATION or a DECREASE FIELD LOCATION, LOAD B instruction.

3. If the sign digit is odd, automatic B register address modification occurs.

Returning to the example given at the beginning of this chapter, the following is a sample loop tally and exit routine using the DECREASE FIELD LOCATION and BRANCH, REPEAT instructions. Assume that the 12 numbers are stored in locations 1000 through 1011, and that the program begins in location 2050.

```
2050   0 0000 CAD 1000 ⎫
2051   0 0000 ADD 1001 ⎬  Sum 12 numbers.
2052   0 0000 STA 1000 ⎭
2053   0 0000 CAD 2058 ⎫
2054   0 0000 ADL 2051 ⎭  Modify address.
2055   0 2201 DFL 2056    Tally.
```

[1]This kind of underflow is not to be confused with the exponent underflow discussed in Chapter 10 (floating point).

[2]The repeat indicator is an electronic device which can be interrogated by a BRANCH REPEAT instruction to determine whether an iteration will be repeated. It is turned on or off by either of two instructions:
DECREASE FIELD LOCATION
DECREASE FIELD LOCATION, LOAD B

```
2056   0 1000 BRP 2050    Test for exit.
2057   0 0000 HLT 0000    Halt operation.
2058   0 0000  00 0001    Constant.
```

PRESETTING

After all or part of a program has been executed by the computer, various instruction or data words in storage have been modified. For example, tally locations may have been referenced by the program, in which case they will contain a portion of, or a completed, tally.

At this point, it may be found necessary to trace a part of the executed program to locate a coding error, check specific program operations, verify subtotals, etc. To do this, it is first necessary to restore each of the modified instruction or data words to its original status.

This may be accomplished by appending a list of instruction and data words to the program to restore the program, thus avoiding the inconvenience of reloading it. The data words are the original contents of the locations which will later contain modified contents. The instructions provide for replacing the modified words with the original ones. For example: Assume that the instructions in locations 1000 and 1010 of the program have been modified during the initial running of the program, and location 1035 has been used for a tally. The program could be written as follows:

```
Location     Instruction            Remarks

  0900    0 0000 CAD 0907  }   Entry point.
  0901    0 0000 STA 1000  }
  0902    0 0000 CAD 0908  }
  0903    0 0000 STA 1010  }   Prestore instructions.
  0904    0 0000 CAD 0909  }
  0905    0 0000 STA 1035  }
  0906    0 0000 BUN 1000      To main program.
  0907    0 9950  10 3000  }
  0908    0 0000  13 2060  }   Prestored words.
  0909    0 0000  00 0000  }
  . . .          . . .     }   Locations 0910 through
  . . .          . . .     }   0999 not used.
  . . .          . . .     }
  1000    0 9950 CAD 3000      Start main program.
```

Whenever it is necessary to repeat the section of the program beginning with location 1000, control is transferred to location 0900. This restores the contents of locations 1000, 1010, and 1035 before the section of the program beginning with location 1000 is carried out.

Location 0900 serves as an entry point to the presetting portion of the program. Many coders place these entry or "restart" points at several locations within their programs. When a specific portion of the program must be re-executed, program control can be transferred to the restart point of that particular segment from any place in the program.

SAMPLE PROBLEM

Sum the 20 numbers stored in locations 0130 through 0149. Store the result in location 0150. Start the program in location 0010. The solution is shown in Table 7-1.

Table 7-1
Solution to Sample Problem

Location	Instruction	Remarks
	First Method	
0010	0 0001 CLA 0000	Clear A register.
0011	0 8000 ADD 0130	Sum numbers.
0012	0 0101 IFL 0011	Modify address.
0013	0 2201 IFL 0011	Tally for loop exit.
0014	0 0000 BOF 0016	Loop exit on overflow.
0015	0 0000 BUN 0011	Return for next iteration.
0016	0 0000 STA 0150	Store sum.
0017	0 0000 HLT 0000	Halt operation.
	Second Method	
0010	0 0001 CLA 0000	Clear A register.
0011	0 1900 ADD 0149	Sum numbers.
0012	0 0101 DFL 0011	Modify address.
0013	0 2201 DFL 0011	Tally for loop exit.
0014	0 0000 BRP 0011	Return for next iteration, if no field underflow.
0015	0 0000 STA 0150	If field underflow, store sum.
0016	0 0000 HLT 0000	Halt operation.

8 Using the B Register

FUNCTION

The Burroughs 220 provides a means for automatic address modification—the B register. The basic function of the B register is to allow the automatic modification of instructions without actually changing their form in core storage. This is done by adding the contents of the B register to the address of an instruction, during the fetch phase, as the copy of the instruction is transferred to the C register. This is simpler and faster than modifying instructions in the A register or in storage; the program is speeded up and the number of programming steps reduced.

Because of its accessibility, the B register is also very valuable in operations involving tallying or counting, and in operations where it is necessary to store an address for later referencing in the program.

GENERAL DESCRIPTION

The B register is a four-digit-position register with no sign-digit position. Its primary purpose is to provide for automatic address modification.

As each instruction is received in the IB register from core storage, it is checked to determine if B register address modification is to take place. The sign-digit position of the instruction is the key in this determination. If the sign digit of the instruction is odd (1 is generally used), the address part of the instruction is increased by the contents of the B register, as it passes through the adder[1] from the IB register to the C register. The instruction in the C register, as modified by the contents of the B register, is then executed. (See Fig. 8-1, examples 1 and 2.)

If a carry-one condition occurs when the address part of the instruction is increased by the contents of the B register, the 1 is ignored. Thus the operation code of the instruction cannot be altered by this occurrence. (See Fig. 8-1, example 3.)

If the sign digit of the instruction in the IB register is even, B register address modification will not occur. It is important to note that instructions with odd digits in the sign position remain unaltered in storage: they are only temporarily modified by the B register immediately before execution. Thus the same instruction may be executed many times in a program, being temporarily modified each time by a different number in the B register.

Simultaneously with the transfer of the modified instruction to the C register, the original instruction is transferred to the D register. The word in the D register—an exact copy of the instruction as it appears in storage—is used for checking purposes.

CODING WITH THE B REGISTER

The B register must be set to a specific value before its use is required by the program. The Burroughs 220 provides loading instructions to set the B register to the value of specified digits of a word in memory.

The coder may increase or decrease a number in the B register—a number which has just been loaded or a number from a previous setting—by a specified amount. This ability to count the B register up or down is provided by two instructions: INCREASE B, BRANCH and DECREASE B, BRANCH. Each of these instructions is actually two instructions in one. They increase or decrease the contents of the B register and they also act as transfer-control instructions. If overflow does not occur as the result of an INCREASE B, BRANCH instruction or field underflow does not occur as a result of a DECREASE B, BRANCH instruction, program control is transferred to the location specified by the address portion of the instruction. If, however, either overflow or field underflow occurs, program control continues in sequence.

TALLYING

The coder often finds it necessary to keep a tally: for example, in loop testing and exiting as described in Chapter 6. He may load the B register with any four digits, then increase or decrease the contents of the register in specified increments each time through the loop. The program will automatically branch back to the beginning of the loop each time it passes the INCREASE B, BRANCH or DECREASE B, BRANCH instruction until an overflow or field underflow condition occurs.

When either condition occurs, the program will resume sequential operation and therefore will exit from the loop.

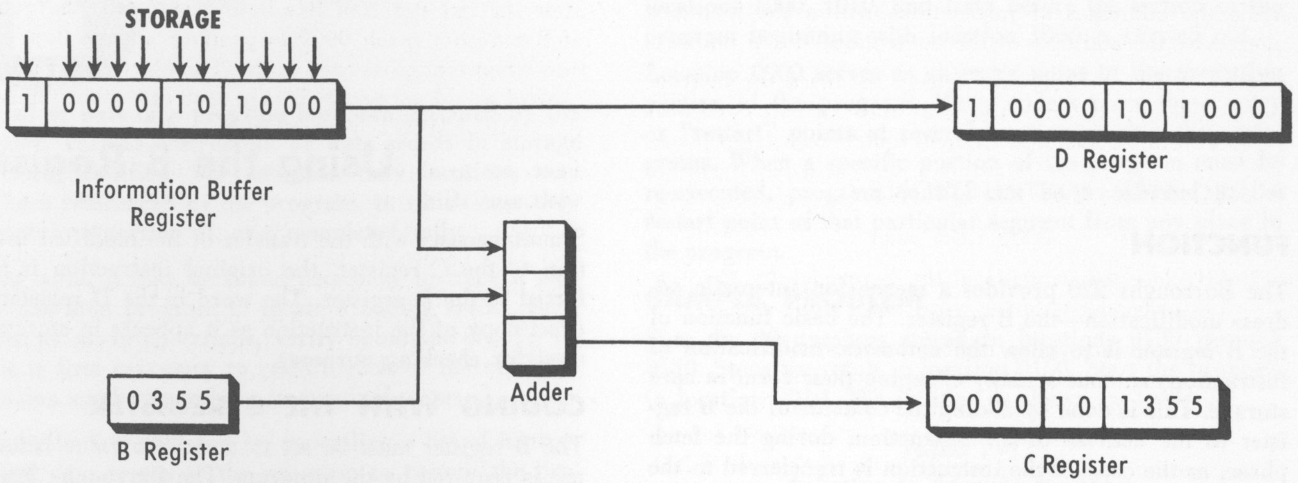
ADDRESS MODIFICATION

The address portion of an instruction may be modified by the contents of the B register as the instruction is fetched from core storage and brought to the C register for execution.

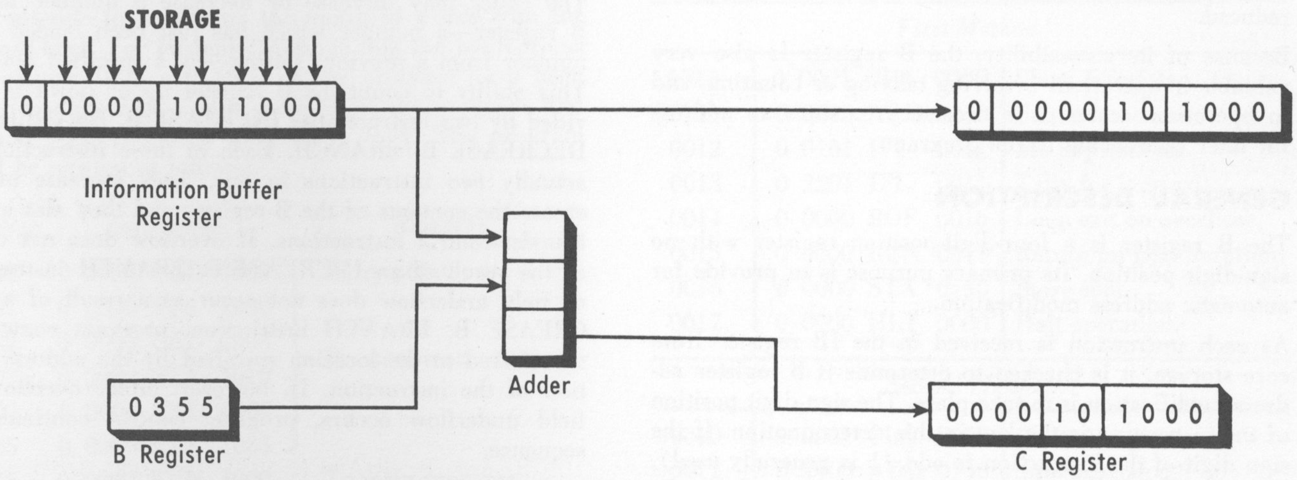
[1]The adder is an electronic device that can form the sum of two decimal digits. Thus, the digits of a sum are formed one at a time in the adder and shifted serially into the specified register.

Using the B Register

Example 1
B Register Address Modification



Example 2
No B Register Address Modification



Example 3
B Register Address Modification
With A "Carry One" Condition

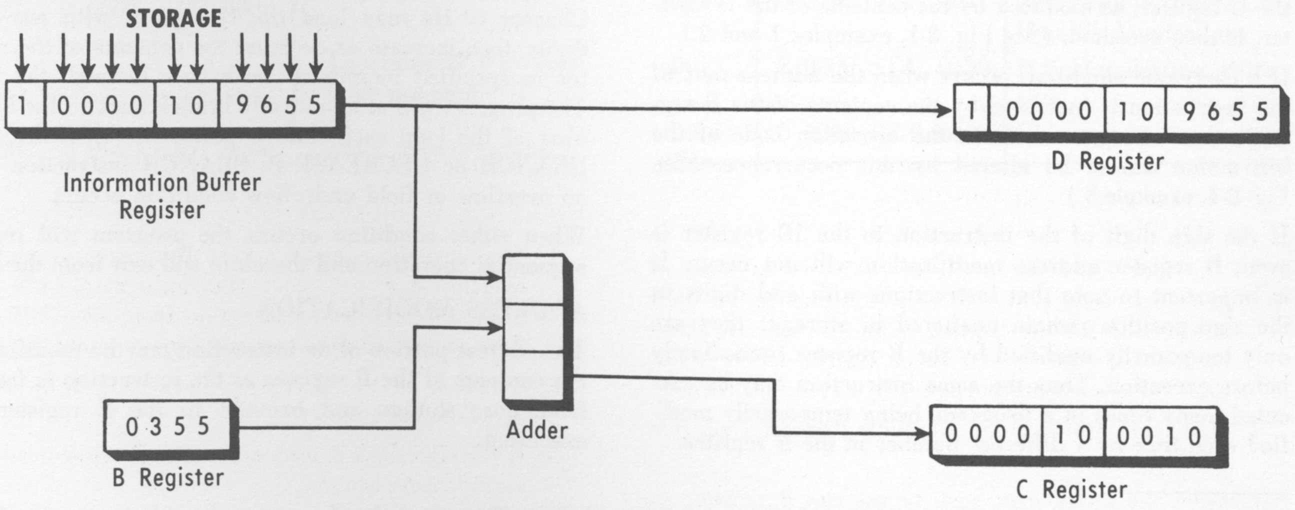


Figure 8-1. Examples of B Register Address Modification

There is often a need to modify several instructions in a loop each time the loop is executed. Since the B register can be counted up or down, tallying a loop may take place while the addresses of instructions within the loop are altered by the addition of the contents of the B register.

Thus a different B register setting modifies the instructions each time through the loop until an exit condition (as the result of overflow or field underflow) takes place.

B REGISTER INSTRUCTIONS

LOAD B (42) ± 0000 LDB aaaa

1. "Replace the contents of the B register by the four low-order digits of the word in location aaaa."
2. The four low-order digits of the word in location aaaa are usually stored as a constant to be used specifically for the B register. However, if the address portion of an instruction is not used for addressing purposes, it may be utilized (e.g., ROUND, HALT, CLEAR A, etc.).
3. If the sign digit is odd, automatic B register address modification occurs.

Examples:

Contents of B register before execution of LDB: 5555

Contents of Location aaaa	*Instruction*	*Contents of B Register After Execution of LDB*
0 0000 00 0250	0 0000 LDB aaaa	0250
0 0086 43 6000	0 0000 LDB aaaa	6000
1 4323 03 3002	0 0000 LDB aaaa	3002

LOAD B COMPLEMENT (42) ± 0001 LBC aaaa

1. "Replace the contents of the B register by the tens complement of the number that is stored in the four low-order digit positions of the word in location aaaa."
2. To obtain the tens complement of a number for use in the B register, four digits are subtracted from 10,000. The execution of this instruction causes the four low-order digits of the word in location aaaa to be automatically subtracted from 10,000. For example:

Contents of Location aaaa	*Complement Operation*	*Contents of B Register After Execution of LBC*
0 0000 00 2310	10,000 − 2310 =	7690
0 0000 16 0004	10,000 − 0004 =	9996
0 9510 10 0000	10,000 − 0000 =	0000
0 0000 00 9982	10,000 − 9982 =	0018
0 0000 00 0150	10,000 − 0150 =	9850

3. The four low-order digits of the word in location aaaa are usually stored as a constant to be used specifically for the loading of the B register. However, if the address portion of an instruction is not used for addressing purposes, it may be utilized.
4. If the sign digit is odd, automatic B register address modification occurs.

Sometimes the coder may wish to use the B register to count up instead of down. This may be necessary to reference locations in ascending order. Loading the B register with the number of iterations to be made and increasing B each time through the loop may not suffice because after the last iteration has been completed, the contents of B may not reach 9999 to produce an overflow condition when increased once more. Therefore the transfer of control would not occur. It is in such cases that the LOAD B COMPLEMENT instruction is useful.

Take, for example, a program in which it is necessary to add a specified quantity to each of the 54 numbers stored in locations 0500 through 0553, in that order. The B register may be used to tally for a loop exit. The coder can load the B register with the tens complement of 54 (the number of locations affected), or 9946. He will then write his first CLEAR ADD instruction with an address of 0554, representing the address of the first location plus the number of locations. This instruction would have a 1 in the sign-digit position so that its address portion would be B modified.

The first time through the loop, the contents of the B register are added to the address portion of the instruction, producing a five-digit number (0554 + 9946 = 10500). The leftmost 1 is deleted by the computer so that the instruction brought to the C register is CLEAR ADD 0500.

After the execution of the CLEAR ADD instruction, the B register is increased by 1 by an INCREASE B, BRANCH instruction. Thus, the second time through the loop, the modified instruction will be CLEAR ADD 0501 (0554 + 9947 = 10501). The contents of the next-to-last location (0553) will be brought to the A register when the B register contains 9999 (0554 + 9999 = 10553). The last INCREASE B, BRANCH instruction, when the register contains 9999, will take the register to zero causing an overflow in the B register and therefore an exit from the loop.

DECREASE B, BRANCH (21) ± nnnn DBB aaaa

1. "Decrease the contents of the B register by nnnn. If field underflow does not occur, transfer control to location aaaa; take the next instruction from aaaa." If field underflow occurs, control continues in sequence.
2. If the sign digit is odd, automatic B register address modification occurs.
3. For examples of instruction, refer to Table 8-1.

INCREASE B, BRANCH (20) ± nnnn IBB aaaa

1. "Increase the contents of the B register by nnnn. If overflow does not occur, transfer control to location aaaa; take the next instruction from aaaa." If overflow occurs, control continues in sequence.
2. Overflow in the B register does NOT turn on the overflow indicator.
3. If the sign digit is odd, automatic B register address modification occurs.
4. Refer to Table 8-1 for examples of instruction.

Table 8-1. Examples of Decrease B, Branch and Increase B, Branch Instructions

Location	Instruction	Contents of B Register Before Execution of Instruction	Contents of B Register After Execution of Instruction	Remarks
		Decrease B, Branch		
0412	0002 DBB 0396	0006	0004	No field underflow; transfer control to location 0396.
0412	0005 DBB 0396	0002	9997	Field underflow occurred; control continues in sequence.
0100	0001 DBB 0400	0001	0000	No field underflow; transfer control to location 0400.
0100	0001 DBB 0400	0000	9999	Field underflow occurred; control continues in sequence.
		Increase B, Branch		
0100	0001 IBB 0396	9984	9985	No overflow; transfer control to location 0396.
0100	0001 IBB 2330	9999	0000	Overflow occurred; control continues in sequence.
0412	0005 IBB 0396	9997	0002	Overflow occurred; control continues in sequence.

STORE B (40) ± sLf2 STB aaaa

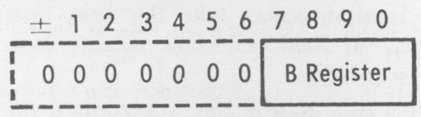
If the sign digit is odd, automatic B register address modification occurs.

If f = 1, partial-word operation will take place; s designates the rightmost digit of the partial-word field. L specifies the number of digits in the partial-word field.

If f = 0, total-word operation will take place; sL not relevant.

1. "Store the specified field of the B register in the corresponding field of location aaaa."

   The B register is treated as if it were 11 digits long:



   If any of the first six digit positions are referenced by sL, 0's will be stored in the corresponding digit positions of the word in location aaaa.

2. If f = 0

   The STORE B instruction replaces the four low-order digits of the word in location aaaa by the contents of the B register. The four high-order digits of the word in location aaaa are replaced by zeros.

3. If f = 1

   The sL digits specify the partial-word field in the B register which replaces the corresponding field of the word in location aaaa. Those digits not included in the field specified by sL are unaltered.

4. Execution of the STORE B instruction does not alter the contents of the B register.

5. Any individual digit except the sign digit can be treated as a separate field. The sign digit must be referenced in conjunction with at least one adjoining digit.

Examples:

Contents of location aaaa before STB is executed: 0 3333 33 3333

Instruction	*Contents of B Register*	*Contents of Location aaaa After STB Is Executed*
0 0002 STB aaaa	9810	0 0000 00 9810
0 4412 STB aaaa	9810	0 0000 33 3333
0 0212 STB aaaa	9810	0 3333 33 3310

DECREASE FIELD LOCATION, LOAD B (28)
± sLnn DLB aaaa

If the sign digit is odd, automatic B register address modification occurs.

s designates the rightmost digit of the partial-word field.

L specifies the number of adjacent digits in the partial-word field.

nn: digits used to decrease the specified partial-word field.

1. "Decrease the specified partial-word field of location aaaa by nn and load the B register with the modified partial-word field. If field underflow occurs, turn off the repeat indicator. If field underflow does not occur, turn on the repeat indicator.

Table 8-2. Examples of Decrease Field Location, Load B Instruction

Instruction	Contents of Location aaaa Before Execution of Instruction	Contents of B Register Before Execution of Instruction	Contents of Location aaaa After Execution of Instruction	Contents of B Register After Execution of Instruction	Repeat Indicator
0 0402 DLB aaaa	0 1223 49 0148	0010	0 1223 49 0146	0146	ON
0 8205 DLB aaaa	0 3946 25 2014	9999	0 3946 25 1514	1500	ON
0 3310 DLB aaaa	0 0050 40 2222	0150	0 9950 40 2222	9950	OFF
0 6650 DLB aaaa	0 1255 00 9753	0000	0 1254 50 9753	1254	ON

2. Note that this is the only case where the B register can be loaded with the contents of any partial-word field in a word—not just the four low-order digits.
3. If the sign-digit position of location aaaa is included in the specified partial-word field, it does not have sign significance; instead it has numeric significance.
4. It is not necessary to follow this instruction immediately with a BRANCH REPEAT instruction. Several other instructions may separate the two instructions, since the repeat indicator, when turned on, remains on until a field underflow condition occurs.
5. If the L digit of the instruction is less than 4, the digits of the specified partial-word field replace the high-order digits of the B register. If the L digit of the instruction is greater than 4, the four high-order digits of the partial-word field replace the entire contents of the B register.
6. For examples of instruction, refer to Table 8-2.

CLEAR B (45) ± 0004 CLB 0000

1. "Replace every digit in the B register by 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits occurs.

Examples:

B Register Before Execution of Instruction	*B Register After Execution of Instruction*
0000	0000
0005	0000
8989	0000

CLEAR A AND B (45) ± 0005 CAB 0000

1. "Replace every digit in the A and B registers by 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits occurs.

Examples:

A and B Registers Before Execution of Instruction	*A and B Registers After Execution of Instruction*
1 1234 56 7890 8989	0 0000 00 0000 0000
0 4000 00 0001 5050	0 0000 00 0000 0000

CLEAR R AND B (45) ± 0006 CRB 0000

1. "Replace every digit in the R and B registers by 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits occurs.

Examples:

R and B Registers Before Execution of Instruction	*R and B Registers After Execution of Instruction*
1 1234 56 7890 8989	0 0000 00 0000 0000
0 3204 00 0000 6001	0 0000 00 0000 0000

CLEAR A, R, AND B (45) ± 0007 CLT 0000

1. "Replace every digit in the A, R, and B registers by 0."
2. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits occurs.

Examples:

A, R, and B Registers Before Execution of Instruction

A	R	B
1 1234 56 7890	1 3214 67 1321	9914
0 4321 98 1001	0 9786 33 4122	4003

A, R, and B Registers After Execution of Instruction

A	R	B
0 0000 00 0000	0 0000 00 0000	0000
0 0000 00 0000	0 0000 00 0000	0000

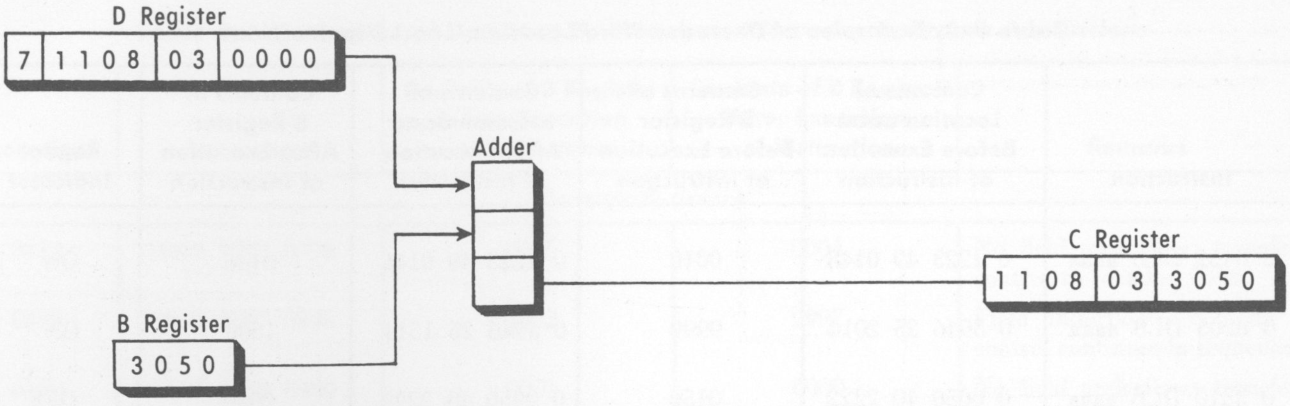


Figure 8-2. B Register Address Modification of Control Word

FLOATING IN A PROGRAM

A coder often wants to make use of standard, existing routines to perform auxiliary functions. For example, after running his main program on the computer, he may need a routine to simplify checking for program errors or to print or punch specified sections of the program and/or data generated by it. Or he may want to use routines with his program for evaluating standard mathematical functions, such as square root or logarithms.

Since these routines are used only occasionally and with programs of different sizes, it is desirable to be able to read such a routine into any core storage locations not being used by the main program. For example, if 4000 words of storage were available and locations 0000 through 2999 were used for the main program and constants, the coder would have locations 3000 through 3999 available for storing and executing any auxiliary routine. A routine designed to use any set of the available locations is one that can be "floated in." Such a program always makes use of "relative addressing."

Relative addressing consists of writing a routine on the assumption that it may be stored in any group of storage locations; the specific locations are assigned by means of B register address modification as the routine is loaded. Because specific storage locations will be assigned on input, the coder can use any consecutive locations when he writes the routine. However, for ease of referencing and for uniformity, the coder will usually begin a relatively coded routine as though it would start in location 0000.

The assignment of specific locations is made by use of the B register for address modification on input. The use of a 1 in the sign-digit position of a word to cause B register address modification during the fetch phase, just prior to execution of the instruction, has been described. For B register address modification of an instruction during input, other digits[2] are used in the sign-digit position. These digits fall into two categories:

1. A sign digit which specifies B register address modification of a control word in which this digit appears.
2. A sign digit in an instruction which specifies B register address modification of the instruction as it is loaded.

FIRST CATEGORY

This digit which calls for B register address modification of the word in which it appears performs the dual function of notifying the computer to send the control word to the C register for execution, first adding the contents of the B register to the address portion. For the purpose of floating in a program, the control words of interest are the input instructions punched in paper tape or cards. For example: A 7 in the sign-digit position of a PAPER TAPE READ instruction notifies the computer to add the contents of the B register to the address portion of the instruction and send it to the C register; the words following the instruction on paper tape are loaded into core storage beginning with the location specified by the modified address. See Fig. 8-2.

The ten-word routine (indicated by digit positions 2 and 3 in the D and C registers) would be loaded into locations 3050 through 3059. The 8 in the variation digit position of the instruction notifies the computer that designated input is to be B register address modified as it is read into storage.

SECOND CATEGORY

In this case, the digits notify the computer to add the contents of the B register to the address portion of the instruction as it passes from the D register through the adder on its way to storage. These digits also tell the computer whether this instruction should be stored with a 0 or a 1 in the sign digit position. For example: if previously specified by an 8 or a 9 in the variation-digit position of the PAPER TAPE READ instruction, an 8 or a 9 in the sign-digit position of a word being loaded from paper tape notifies the computer that the address portion of the word is to be B modified as it goes into storage. If an 8 is used, it is changed to a 0 before the

[2]Although paper-tape instructions are used as examples in the discussion following, the digits described are the same for Cardatron instructions.

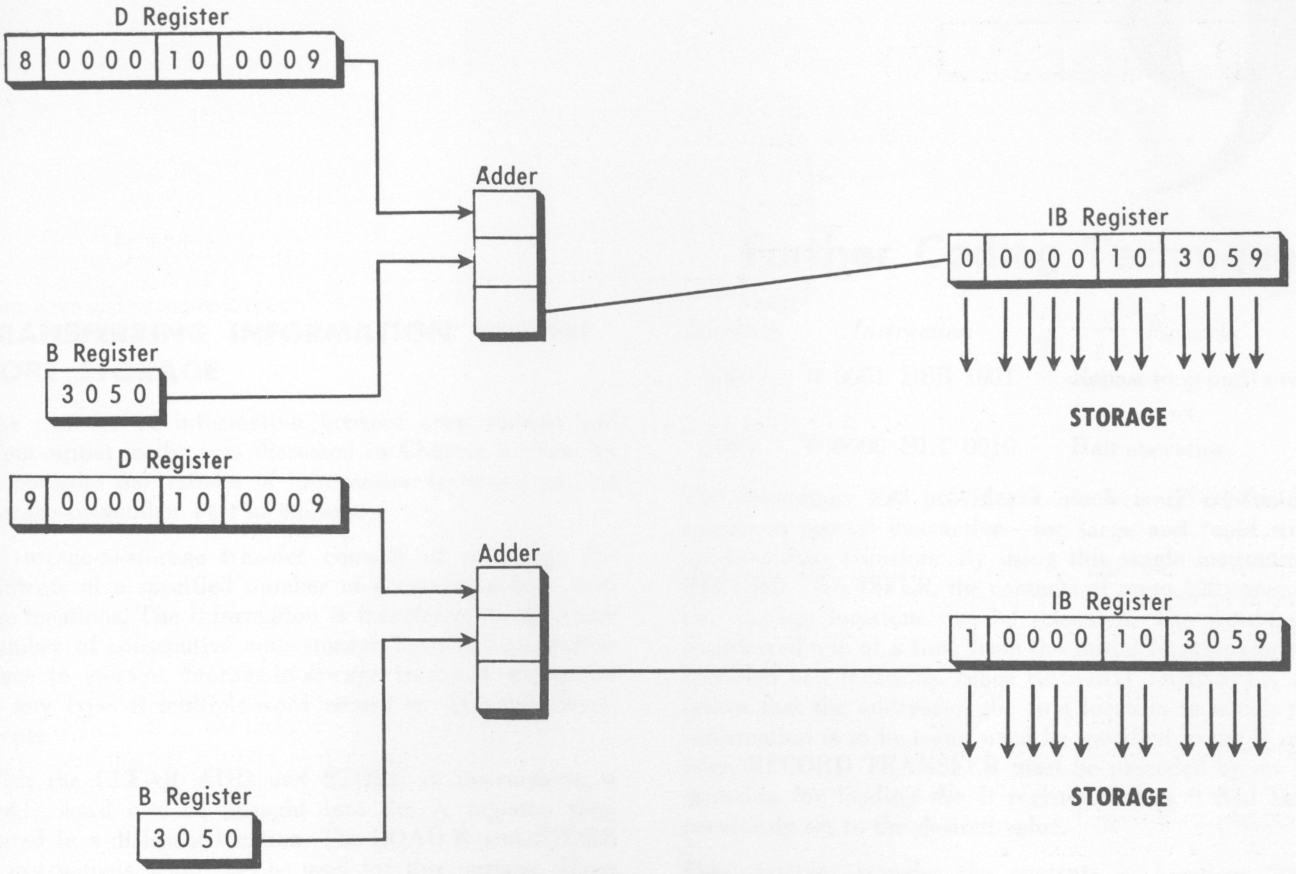


Figure 8-3. B Register Address Modification of Instruction

word is stored; if a 9 is used, it is changed to a 1 before the word is stored (Fig. 8-3).

SAMPLE PROBLEMS

Assume that the coder wants to use the first method shown in Table 8-3 as an auxiliary routine; he would like to read it into core storage beginning in location 1010. He would first set the B register to 1000. The program would appear on paper tape as follows:

```
7 1069 PRD 0010
0 0003 CAR 0020
8 0001 LBC 0010
1 0000 ADD 0150
8 0001 IBB 0012
0 0000 STA 0150
0 0000 HLT 0000
```

The program in storage would appear as follows:

```
1010  0 0003 CAR  0020
1011  0 0001 LBC  1010
1012  1 0000 ADD  0150
1013  0 0001 IBB  1012
1014  0 0000 STA  0150
1015  0 0000 HLT  0000
```

Table 8-3. Sample Use of Overflow and Underflow

Clear the A and R registers. Sum the 20 numbers stored in locations 0130 through 0149. Store the result in location 0150. Start the program in location 0010.

Instruction	Remarks
First Method	
0010 0 0003 CAR 0020	Clear A and R registers.
0011 0 0000 LBC 0010	9980 $\longrightarrow$ rb.
0012 1 0000 ADD 0150	Sum the 20 numbers.
0013 0 0001 IBB 0012	If no overflow in B register return for next iteration.
0014 0 0000 STA 0150	If overflow, store sum.
0015 0 0000 HLT 0000	Halt operation.
Second Method	
0010 0 0003 CAR 0019	Clear A and R registers.
0011 0 0000 LDB 0010	0019 $\longrightarrow$ rb.
0012 1 0000 ADD 0130	Sum the 20 numbers.
0013 0 0001 DBB 0012	If no field underflow in B register, return for next iteration.
0014 0 0000 STA 0150	If field underflow, store sum.
0015 0 0000 HLT 0000	Halt operation.

9 Further Coding Techniques

TRANSFERRING INFORMATION WITHIN CORE STORAGE

The transfer of information between core storage and input-output media was discussed in Chapter 3. Now let us consider the transfer of information from one part of storage to another part of storage.

A storage-to-storage transfer consists of relocating the contents of a specified number of consecutive core storage locations. The information is transferred to the same number of consecutive core storage locations at another place in storage. Storage-to-storage transfers are useful in any type of multiple-word record or data rearrangements.

With the CLEAR ADD and STORE A instructions, a single word can be brought into the A register, then stored in a different location. The LOAD R and STORE R instructions could also be used for this purpose. However, using this method for storage-to-storage transfers of large volumes of information would be a slow and tedious process.

This method could be simplified by using address modification within a loop, but it would be even slower. For example, by means of an address-modified CLEAR ADD instruction followed by an address-modified STORE A instruction, the contents of consecutive storage locations can be brought into the A register one by one and stored in turn in consecutive locations elsewhere in storage.

The address portions of the CLEAR ADD and STORE A instructions may be modified either by an arithmetic operation or by the B register, depending on the particular program or the preference of the coder. In either case, the operation would be set up in a loop, thus requiring some tallying and testing arrangement by which program control would exit from the loop after the contents of the specified number of locations had been relocated.

For example: transfer the contents of locations 2000 through 2010 to locations 2040 through 2050. Use the B register for address modification. Start the program in location 1000. Halt the operation after the transfer.

Location	*Instruction*	*Remarks*
1000	0 0000 LDB 1004	0010→rB.
1001	1 0000 CAD 2000	Contents of 2000-2010 are transferred to 2040-2050.
1002	1 0000 STA 2040	
1003	0 0001 DBB 1001	Repeat loop until overflow occurs.
1004	0 0000 HLT 0010	Halt operation.

The Burroughs 220 provides a much more convenient means—a special instruction—for large and rapid storage-to-storage transfers. By using this single instruction, RECORD TRANSFER, the contents of up to 100 consecutive storage locations can be relocated. The words are transferred one at a time from the initial locations to the specified new locations. Since RECORD TRANSFER requires that the address of the first location to which the information is to be transferred be specified in the B register, RECORD TRANSFER must be preceded by an instruction for loading the B register, unless it had been previously set to the desired value.

For example: transfer the contents of locations 2000 through 2010 to locations 2040 through 2050. Use the RECORD TRANSFER instruction. Start the program in location 1000. Halt the operation after the transfer.

Location	*Instruction*	*Remarks*
1000	0 0000 LDB 1002	2040→rB.
1001	0 0110 RTF 2000	Transfer information.
1002	0 0000 HLT 2040	Halt operation.

This program does the following:

$$\text{Contents of}\left\{\begin{array}{l} 2000 \rightarrow 2040 \\ 2001 \rightarrow 2041 \\ 2002 \rightarrow 2042 \\ 2003 \rightarrow 2043 \\ 2004 \rightarrow 2044 \\ 2005 \rightarrow 2045 \\ 2006 \rightarrow 2046 \\ 2007 \rightarrow 2047 \\ 2008 \rightarrow 2048 \\ 2009 \rightarrow 2049 \\ 2010 \rightarrow 2050 \end{array}\right.$$

RECORD TRANSFER (29) ± 0 nn 0 RTF aaaa

If the sign digit is odd, automatic B register address modification occurs.

0, not relevant to the execution of this instruction.

nn: number of words to be relocated.

0, not relevant to the execution of this instruction.

1. "Relocate the contents of nn consecutive storage locations, beginning with location aaaa."
2. The specified words are transferred one at a time to the nn consecutive storage locations beginning with the location whose address is in the B register.
3. If nn = 00, 100 consecutive words will be transferred; if nn = 01, one word will be transferred.
4. After the execution of a RECORD TRANSFER instruction:

   a. The B register will contain the address of the last location filled plus 1, that is, the address of the next location to be filled.

   b. The address part of the instruction in the C register will be equal to the address of the last location from which a word was transferred plus 1, that is, the address of the next location from which a word will be transferred.

SUBROUTINES

When writing a large program, the coder often finds that a certain group or sequence of instructions must be repeated several times at different points in the program. This group of instructions will perform a single well-defined function. For example, there may be many points in the program where a group of numbers must be sorted, the square root of a number found, or a FICA tax determined. It is usually not practical to write the necessary instructions in the main program every time the operation is needed. Instead, the required sequence of instructions may be written once—as a subprogram to the main program. Then control may be transferred to the subprogram, or subroutine, each time it is needed. The desired standard sequence of operations is performed and control returned to the main program.

Typical programs may use dozens of subroutines. In fact, some subroutines are used so frequently that they have been written as separate programs, i.e., coded independently from any specific main program. Such subroutines can be filed in a reference library.

Since subroutines are generally not placed in the body of the program, but are stored separately from the main program and entered by a transfer-control instruction, there must be an instruction in the subroutine to transfer control back to the main program. Such an exit instruction from a subroutine must transfer to different locations at different times. Thus, the exit instruction cannot be a fixed instruction in the subroutine; instead it must be altered depending on the circumstances of the exit.

This process of transferring control between the main program and a subroutine is referred to as linkage, or linking the subroutine to the program.

Most frequently, control must be transferred from a subroutine back to the point in the program at which the normal program sequence was interrupted. The following instruction is very useful in this type of program-subroutine linkage:

STORE P (44) ± 0000 STP aaaa

1. "Replace the address portion of location aaaa by the contents of the P register, increased by one."
2. Normally, the P register contains the address of the location from which the next instruction will be taken (i.e., the contents of P is 1 greater than the address of the instruction being executed). For example:

Location	*Instruction*	*Contents of P Register*
1000	0 0000 CAD 2000	1001
2025	0 0000 STA 0100	2026
4073	0 0000 SUB 0335	4074

3. When a STORE P instruction is executed, the contents of the P register plus 1, or the address of the location two addresses beyond that of the STORE P instruction, is stored. The STORE P instruction rarely stands by itself; it is usually followed by an instruction that transfers control to a subroutine. Because STORE P is followed by a transfer control instruction, the re-entry point into the main program is two addresses beyond the STORE P address. For example:

Location	*Instruction*	*Remarks*
1000	0 0000 STP 0200	Location 0200 = 0000 00 1002.
1001	0 0000 BUN 3000	Transfer to subroutine beginning in location 3000.
1002	0 0000 CAD 3500	Re-entry point into main routine.

4. Although the address stored is two addresses beyond the location of the STORE P instruction, the contents of the P register remain 1 greater than the address of the instruction being executed. For example:

Location	*Instruction*	*Contents of P Register*	*Address Stored by Store P Instruction*
1000	0 0000 STP 2000	1001	1002
2025	0 0000 STP 0100	2026	2027
4073	0 0000 STP 0335	4074	4075

A way to set up a subroutine exit is to store the contents of the P register in the address portion of the word in the first location of the subroutine. This location will have a 30 already stored in its fifth and sixth digit positions (0 0000 30 0000). When the contents of the P register are thus stored, the contents of the location become an instruction (BUN) that transfers control to the re-entry location of the main program.

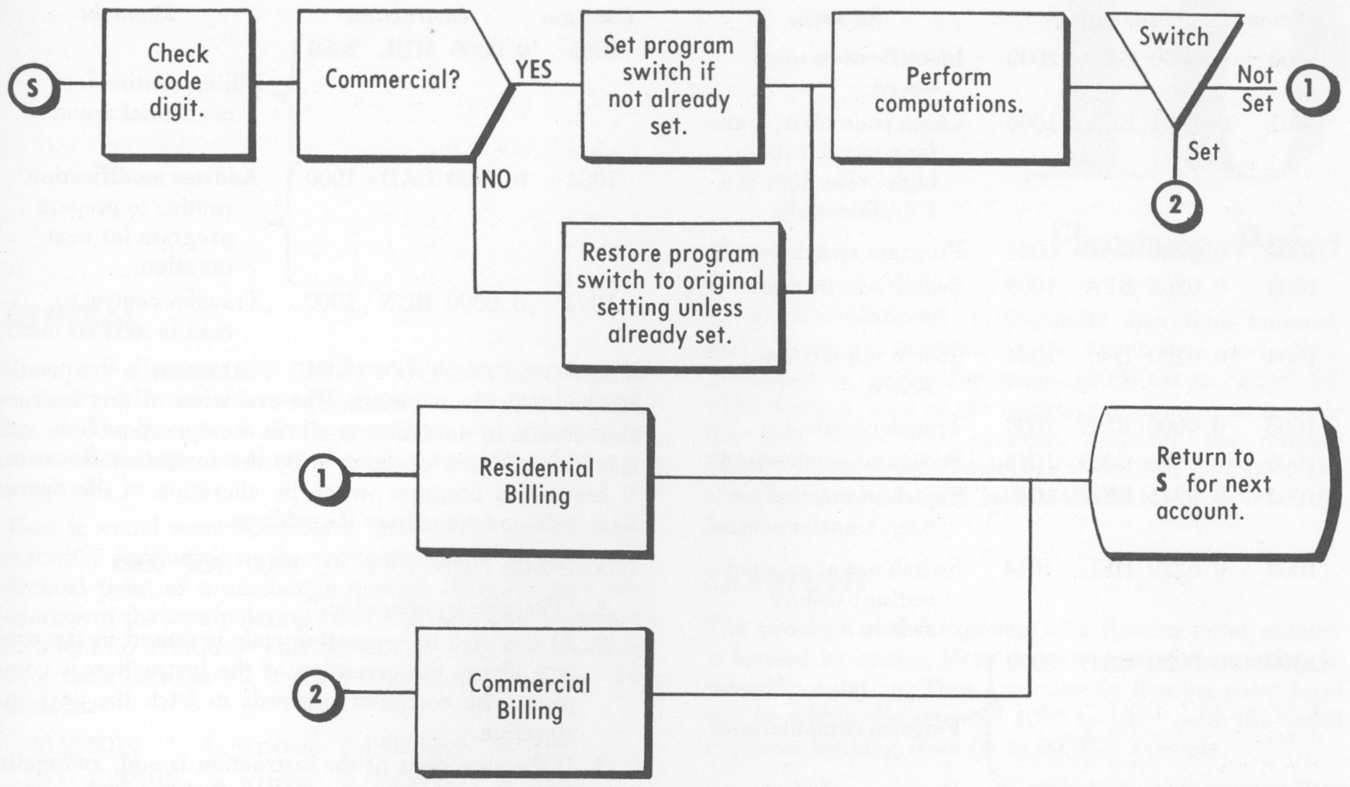

S
Check code digit.
Commercial?
YES
NO
Set program switch if not already set.
Perform computations.
Switch
Not Set
Set
1
2
Restore program switch to original setting unless already set.
1
Residential Billing
Return to S for next account.
2
Commercial Billing

Figure 9-1. Utility Billing Flow Chart

Example:

Location	*Instruction*	*Remarks*
0500	0 0000 ADD 1000	Part of main program.
0501	0 0000 STA 1500	
0502	0 0000 STP 2000	Store P + 1 in first location of subroutine.
0503	0 0000 BUN 2001	Transfer control to subroutine.
0504	0 0000 CAD 1001	Re-entry point into main program.
. . .	. . .	Main program.
. . .	. . .	
2000	0 0000 30 (0000)	Instruction for re-entry to main program. Becomes 0 0000 30 0504 when STP instruction executed.
2001	0 0000 CAD 0550	First instruction of subroutine.
. . .	. . .	Subroutine.
. . .	. . .	
. . .	. . .	
2025	0 0000 30 2000	Last step of subroutine. Transfer control to re-entry instruction.

PROGRAM SWITCHES

A program switch, as discussed in this chapter, is an instruction within the program that can be altered by the program to cause the computer to take one of several alternate courses of action.

Assume that as the result of a test within the computer, a decision is made aș to a future course of action. At this time, the instruction being used for a switch would be altered by the program so that when this instruction is reached it will cause control to be transferred to the desired location. If in the course of the program this instruction is altered, the program switch is said to be set. To illustrate this concept, take the following example.

Consider the customer accounting problem of a public utility, where accounts must be identified as either residential or commercial. Both types of accounts are processed in the same manner, but different billing routines are used once the initial processing is completed. To determine the type of account, an identification code must be used. This code is a specific digit in the high-order digit position of one word in the account record. For example, a 1 may be used to designate a residential account and a 2 for a commercial account. When the identification is made, a program switch is set (an instruction altered) so that when the computations for the account have been completed, the proper billing routine is carried out. The flow chart in Figure 9-1 and the following coding illustrate this example.

This program was written assuming an account record is stored beginning with location 2000.

Location	*Instruction*	*Remarks*
1000	0 0000 CAD 2000	Identification word $\rightarrow$ rA.
1001	0 1101 BFA 1006	Check code digit; transfer control if the high-order digit is a 1 (residential).
1002	0 0000 CAD 1044	Program switch $\rightarrow$ rA.
1003	0 0265 BFA 1009	Switch set; transfer control.
1004	0 6220 IFL 1044	Switch not set; set switch.
1005	0 0000 BUN 1009	Transfer control.
1006	0 0000 CAD 1044	Program switch $\rightarrow$ rA.
1007	0 0245 BFA 1009	Switch at original setting; transfer control.
1008	0 6220 DFL 1044	Switch not at original setting; restore switch.
1009	0 0000 CAD 2000	
. . .	. . .	
. . .	. . .	Perform computations.
. . .	. . .	
1044	0 0000 BUN (1065)	Program switch.
1045	0 0000 MUL 2040	
. . .	. . .	Billing routine for commercial account.
. . .	. . .	
. . .	. . .	
1064	0 0000 BUN 1084	Transfer control to address modification routine.
1065	0 0000 MUL 2050	
. . .	. . .	Billing routine for residential account.
. . .	. . .	
. . .	. . .	
1084	0 0000 CAD 1000	Address modification routine to prepare program for next iteration.
. . .	. . .	
. . .	. . .	
. . .	. . .	
1094	0 0000 BUN 1000	Transfer control to read in next account.

The NO OPERATION (NOP) instruction is frequently used as a program switch. The execution of this instruction results in no action at all, as the operation code and specified address are ignored by the computer. However, it becomes a program switch by alteration of the operation code and sometimes the address.

NO OPERATION (01) $\pm$ 0000 NOP 0000

1. "Perform no operation."
2. At the time the operation code is sensed in the execute phase, the execution of the instruction is complete. The computer proceeds to fetch the next instruction.
3. If the sign digit of the instruction is odd, automatic B register modification of the four low-order digits of the instruction occurs.

If used as a program switch, the NO OPERATION instruction is often modified to become a branch unconditionally instruction. In such a situation, the coder stores a NOP instruction with an address that will be used when the operation code is changed from 01 (NOP) to 30 (BUN).

10 Floating Point

GENERAL

Chapter 5 discussed the problem of decimal scaling of numbers to be entered into the computer and manipulated by it. The process is time-consuming and painstaking work when many numbers of widely varying values are involved.

Thus it would seem desirable to have some way of automatically indicating to the computer the location of the decimal point of a number in storage. Why not take advantage of the manipulating facility provided by a variant of scientific notation? This consists of writing a number as a value between 0.1 and 1.0 times a power of ten. To illustrate:

4332.8019	is written	0.43328019×10^4
.00043328019	is written	$0.43328019 \times 10^{-3}$
12.3456789	is written	0.123456789×10^2
12345678	is written	0.12345678×10^8

With scientific notation as a basis, there is a way of automatically indicating to the computer the location of the decimal point of a number in storage, and instructing the computer to take account of the placement of this decimal point in all arithmetic operations. Such a system is said to operate in floating-point arithmetic as distinguished from fixed-point arithmetic.

There are two methods of handling floating-point operations: this facility may be designed into the circuitry of the computer or special subroutines may be devised. The Burroughs 220 has the automatic floating-point feature built in.

In floating-point operation, the power of ten, called the exponent, of a number in scientific notation is stored with an eight-digit number, called the mantissa. This combination of exponent and so-called mantissa forms a "floating-point number." Although other arrangements are used in some other computers, in the Burroughs 220 the coded exponent of a floating-point number is stored in the two most significant (high-order) digit positions of a word and the mantissa is stored in the eight least significant (low-order) digit positions of the same word. The sign of the word is the sign of the mantissa.

MANTISSA

In general, the mantissa of a number in standard floating-point form must be normalized—that is, the most significant digit must be a digit other than zero. The one exception to this is the number zero itself, ± 00 0000 0000. All floating-point operations automatically leave the mantissa of the result in a normalized form, that is, within the range .10000000 to .99999999. (The decimal point of the mantissa is assumed to precede the first digit. For example, consider the number 12345678: its so-called mantissa would be .12345678. The complete number would be $.12345678 \times 10^8$ in scientific notation.)

EXPONENT

The two-digit coded exponent of a floating-point number is formed by adding 50 to the exponent of the number in scientific notation. Thus a number in floating-point form will be within the range of 10^{+49} to 10^{-51} with the coded exponent ranging from 00 to 99. For example:

Number	*Scientific Notation*	*Floating-Point Representation*
+.0932456601	$+.932456601 \times 10^{-1}$	+49 9324 5660
+ 83067.45911	$+.8306745911 \times 10^5$	+55 8306 7459
+.00004832101	$+.4832101 \times 10^{-4}$	+46 4832 1010
−.600003298	$-.600003298 \times 10^0$	−50 6000 0330
− 123456789	$-.123456789 \times 10^9$	−59 1234 5679
−.0000000000091	$-.91 \times 10^{-11}$	−39 9100 0000
+ 0000000000	+ 0000000000	+00 0000 0000

Note that as the result of a floating-point arithmetic operation, overflow can occur. In addition and subtraction, *arithmetic* overflow occurs when the operation causes overflow from the high-order digit of the mantissa into a 99 exponent, thus creating an exponent greater than 99. In multiplication and division, *exponent* overflow occurs when an exponent greater than 99 is generated by the operation (during normalization of the result). In both cases, the overflow indicator is turned on.

If an exponent less than 00 should be generated by an arithmetic operation, exponent underflow will occur and the arithmetic registers (A and R) will be cleared.

FLOATING-POINT INSTRUCTIONS

The Burroughs 220 provides the following instructions to handle floating-point arithmetic operations.

FLOATING ADD (22) ± 0000 FAD aaaa

1. "Add the contents of location aaaa to the contents of the A register."
2. Both the contents of the A register and the contents of location aaaa are treated like floating-point numbers.
3. The sum is in the A register in floating-point form

—the mantissa normalized and the exponent properly adjusted.

4. If, as a result of the execution of this instruction:

   a. The coded exponent would exceed 99, arithmetic overflow occurs and the Overflow Indicator is turned on.

   b. The coded exponent would be smaller than 00, exponent underflow occurs and the A and R registers are cleared; no other indication is given.

5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CAD	2050	Clear the A register and load it with the first number to be added.
1001	FAD	3000	Add in floating-point arithmetic the contents of location 3000.
1002	STA	4000	Store the sum in location 4000.

Contents of A Register After Execution of Instruction in 1000	*Contents of Location 3000*	*Sum in A Register*	*Overflow Indicator*
0 5022 00 0000	0 5044 00 0000	0 50 6600 0000	OFF
0 7090 00 0000	0 5230 00 0000	0 70 9000 0000	OFF
1 3081 00 0000	0 3081 00 0000	1 00 0000 0000	OFF
0 9990 00 0000	0 9910 00 0000	0 01 0000 0000	ON
0 5390 00 0000	0 5310 00 0000	0 54 1000 0000	OFF
1 5120 00 0000	0 4920 00 0000	1 51 1980 0000	OFF
1 9990 00 0000	1 9920 00 0000	1 01 1000 0000	ON

FLOATING ADD ABSOLUTE (22) ± 0001 FAA aaaa

1. "Add the absolute value of the contents of location aaaa to the contents of the A register."
2. Both the contents of the A register and the contents of location aaaa are treated like floating-point numbers.
3. The sum is in the A register in floating-point form —the mantissa normalized and the exponent properly adjusted.
4. If as a result of the execution of this instruction:

   a. The coded exponent would exceed 99, arithmetic overflow occurs and the Overflow Indicator is turned on.

   b. The coded exponent would be smaller than 00, exponent underflow occurs and the A and R registers are cleared; no other indication is given.

5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CAD	2050	Clear the A register and load it with the first number to be added.
1001	FAA	3000	Add in floating-point arithmetic the absolute value of the contents of location 3000.
1002	STA	4000	Store the sum in location 4000.

Contents of A Register After Execution of Instruction in 1000	*Contents of Location 3000*	*Sum in A Register*	*Overflow Indicator*
0 5086 00 0000	0 5014 00 0000	0 51 1000 0000	OFF
0 9980 00 0000	1 9920 00 0000	0 01 0000 0000	ON
0 6244 00 0000	1 6022 00 0000	0 62 4422 0000	OFF
0 4860 00 0009	1 5020 00 0000	0 50 2060 0000	OFF
1 4030 00 0000	1 4120 00 0000	0 41 1700 0000	OFF
0 9980 00 0000	1 9940 00 0000	0 01 2000 0000	OFF

FLOATING SUBTRACT (23) ± 0000 FSU aaaa

1. "Subtract the contents of location aaaa from the contents of the A register."
2. Both the contents of location aaaa and the contents of the A register are treated like floating-point numbers.
3. The difference is in the A register in floating-point form—the mantissa normalized and the exponent properly adjusted.
4. If, as a result of the execution of this instruction:

   a. The coded exponent would exceed 99, arithmetic overflow occurs and the Overflow Indicator is turned on.

   b. The coded exponent would be smaller than 00, exponent underflow occurs and the A and R registers are cleared; no other indication is given.

5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CAD	2050	Clear the A register and load it with the minuend.
1001	FSU	3000	Subtract in floating-point arithmetic the contents of location 3000.
1002	STA	4000	Store the difference in location 4000.

Contents of A Register After Execution of Instruction in 1000	*Contents of Location 3000*	*Difference in A Register*	*Overflow Indicator*
0 5039 00 0000	0 4940 00 0000	0 50 3500 0000	OFF
0 5240 00 0000	1 5040 00 0000	0 50 4040 0000	OFF
1 9990 00 0000	0 9920 00 0000	1 01 1000 0000	ON
1 4020 00 0000	1 3030 00 0000	1 40 2000 0000	OFF
1 7060 00 0000	1 7060 00 0000	1 00 0000 0000	OFF

FLOATING SUBTRACT ABSOLUTE (23) ± 0001 FSA aaaa

1. "Subtract the absolute value of the contents of location aaaa from the contents of the A register."

2. Both the contents of location aaaa and the contents of the A register are treated like floating-point numbers.
3. The difference is in the A register in floating-point form—the mantissa normalized and the exponent properly adjusted.
4. If, as a result of the execution of this instruction:
   a. The coded exponent would exceed 99, arithmetic overflow occurs and the Overflow Indicator is turned on.
   b. The coded exponent would be smaller than 00, exponent underflow occurs and the A and R registers are cleared; no other indication is given.
5. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CAD	2050	Clear the A register and load it with the minuend.
1001	FSA	3000	Subtract in floating-point arithmetic the absolute value of the contents of location 3000.
1002	STA	4000	Store the difference in location 4000.

Contents of A Register After Execution of Instruction in 1000	*Contents of Location 3000*	*Difference in A Register*	*Overflow Indicator*
0 5060 00 0000	1 5020 00 0000	0 50 4000 0000	OFF
0 6230 00 0000	1 6040 00 0000	0 62 2960 0000	OFF
1 3912 34 5678	1 4012 34 5678	1 40 1358 0245	OFF
0 9990 00 0000	0 9990 00 0000	0 00 0000 0000	OFF
1 9980 00 0000	1 9920 00 0000	1 01 0000 0000	ON

FLOATING MULTIPLY (24) ± 0000 FMU aaaa

1. "Multiply the contents of location aaaa by the contents of the A register."
2. Both the contents of the A register and the contents of location aaaa are treated like floating-point numbers.
3. The product is in floating-point form. The two-digit coded exponent and the eight high-order digits of the mantissa are in the A register. The remaining seven or eight digits of the mantissa are in the high-order digit positions of the R register—the last two or three digit positions of the R register being cleared to zero. The sign of the product appears in the sign-digit positions of both the A and the R registers.
4. If, as a result of the execution of this instruction:
   a. Exponent overflow occurs, the Overflow Indicator is turned on.
   b. Exponent underflow occurs, the A and R registers are cleared; no other indication is given.
5. If the mantissa of the operand in the A register, or the mantissa of the operand in location aaaa, is not normalized—that is, if either mantissa has a high-order digit of 0—the operation is terminated and the A and R registers are cleared: the product is assumed to be zero.
6. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples: Refer to table 10-1.

Table 10-1. Examples of Floating Multiply

Given:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CAD	2050	Clear the A register and load it with the multiplier.
1001	FMU	3000	Multiply in floating-point arithmetic by the contents of location 3000.
1002	STA	4000	Store the product in location 4000.

Contents of A Register After Execution of Instruction in 1000	**Contents of Location 3000**	**Product in the A and R Registers**		**Overflow Indicator**
0 7580 00 0000	0 8020 00 0000	0 0080 00 0000	0 0000 00 0000	ON
0 5060 00 0000	0 5030 00 0000	0 5018 00 0000	0 0000 00 0000	OFF
0 3020 00 0000	0 6020 00 0000	0 3940 00 0000	0 0000 00 0000	OFF
1 5112 34 5678	0 5120 00 0000	1 5124 69 1356	1 0000 00 0000	OFF
0 5211 11 1111	0 5222 22 2222	0 5324 69 1357	0 5308 64 2000	OFF
0 6002 00 0000	0 5520 00 0000	0 0000 00 0000	0 0000 00 0000	OFF but operation terminated

FLOATING DIVIDE (25) ± 0000 FDV aaaa

1. "Divide the contents of the combined A and R registers by the contents of location aaaa." The exponent and the eight high-order digits of the mantissa of the dividend are in the A register. The eight low-order digits of the mantissa of the dividend are in the high-order digit positions of the R register.
2. Both the dividend in the A and R registers and the divisor in location aaaa are treated like floating-point numbers.
3. The quotient is in floating-point form. The two-digit coded exponent and the eight high-order digits of the mantissa are placed in the A register. The one or two low-order digits of the mantissa are in the high-order digit positions of the R register.
4. The remainder is in the low-order digit positions of the R register.
5. If, as a result of the execution of this instruction:
   a. Exponent overflow would occur, the overflow indicator is turned on.
   b. Exponent underflow would occur, the A and R registers are cleared; no other indication is given.
6. If the mantissa of the dividend is not normalized—that is, if the high-order digit is 0—but the mantissa of the divisor is normalized, the operation is terminated and the A and R registers are cleared: the dividend is assumed to be zero.
7. If the mantissa of the divisor is not normalized, it is assumed that the divisor is zero. The operation is terminated and the overflow indicator turned on.
8. If the sign digit of the instruction is odd, automatic B register address modification occurs.

Examples: Refer to table 10-2.

Table 10-2. Examples of Floating Divide

Given:

Location	*Operation*	*Operand Address*	*Remarks*
1000	CLR	0000	Clear the R register.
1001	CAD	2050	Clear the A register and load it with the dividend.
1002	FDV	3000	Divide in floating-point arithmetic by the contents of location 3000.
1003	STA	4000	Store the quotient in location 4000.

Contents of A Register After Execution of Instruction in 1001	Contents of Location 3000	Quotient and Remainder in the A and R Registers		Overflow Indicator
0 5280 00 0000	0 5040 00 0000	0 5320 00 0000	0 0000 00 0000	OFF
0 2060 00 0000	0 5030 00 0000	0 2120 00 0000	0 0000 00 0000	OFF
0 5040 00 0000	0 5030 00 0000	0 5113 33 3333	0 3300 10 0000	OFF
0 2004 00 0000	0 5030 00 0000	0 0000 00 0000	0 0000 00 0000	OFF
0 5040 00 0000	0 5003 00 0000	0 5040 00 0000	0 0000 00 0000	ON
0 9020 00 0000	0 2010 00 0000	0 0020 00 0000	0 0000 00 0000	ON

As shown above, the R register should be cleared before FLOATING DIVIDE unless a 16-digit dividend is to be used. To illustrate:

Dividend in A and R Registers		*Divisor*	*Contents of A and R Registers After FDV*		*Overflow Indicator*
0 5288 88 8888	0 8888 88 8888	0 5640 00 0000	0 4722 22 2222	0 2200 08 8888	OFF
0 5033 33 3333	0 3333 33 3333	0 5060 00 0000	0 5055 55 5555	0 5003 33 3333	OFF

11

The Cardatron System

GENERAL

Punched cards are an important input-output medium of the Burroughs 220 Data Processing System. When they are used, standard electro-mechanical punched card equipment is required to read information on the cards, to punch processed or computed information into cards, and to print processed information. Certain difficulties would be encountered if the punched-card equipment had to communicate directly with the computer. These difficulties arise from the differences in the mode of operation of the computer and the punched-card and printing devices and involve speeds, codes, and word length.

COMMUNICATION PROBLEMS

SPEEDS

The computer is much faster in operation than the card-handling machines and line printers. If information were accepted by the computer directly from a card device, the computer would be forced to wait while the information was being read by the card machine, thereby reducing the over-all operating speed of the computer.

CODES

The code used with punched cards allows both numeric and alphanumeric characters to be represented by a single card column. For straight numeric words there is no problem of code difference. However, any character—alphabetic, numeric, or special—in an alphanumeric word must be represented in the Burroughs 220 by a two-digit code. Thus information in punched card format may be in a form that is not acceptable to the Burroughs 220.

WORD LENGTH

The computer is limited to receiving information in fixed words of 10 digits plus sign digit, but information recorded in punched cards may be in fields of any length. Therefore, information from punched cards must somehow be grouped into words of a specific length in order to be compatible with either the computer fixed-word length or a specified partial-word length.

The Cardatron System[1] was designed to eliminate such difficulties; it enables the Burroughs 220 to handle applications which deal with masses of punched-card input and printed-report and punched-card output. This system provides a flexible means of linking standard punched-card machines and line printers to the computer.

The Cardatron System resolves the differences between punched-card equipment and the computer as follows:

THE PROBLEM OF SPEED DIFFERENCE

The Cardatron System relieves the computer of the necessity of waiting while punched cards move through the card devices. To permit the computer to operate independently of the card machines, the Cardatron System uses a small magnetic drum—called the buffer—as an intermediate storage device. Each card machine is attached to a Cardatron Input Unit or a Cardatron Output Unit which contains one of these magnetic drums. The card machine communicates not with the computer but with the buffer of the associated Input or Output Unit. The Input Unit receives the information from the card-reading device and holds it until the computer calls for it. The Output Unit accepts information from the computer and holds it until a line printer is ready to print a line of information or until a card punch is ready to punch information into a card.

The action of the Input and Output Units is governed by the Cardatron Control Unit. The Control Unit synchronizes a single Input or Output Unit with the computer until the transfer of information to or from the computer is made. It then releases that unit from the computer and synchronizes it or another unit with the associated card device. An Input Unit will accept new information from the associated card-reading device; an Output Unit will transmit information to the associated punching or printing device. Thus the computer has no direct contact with the card machines, and, since the buffer drums in the Input and Output Units operate at computer speed, the time that the computer is tied up with these units, during transfer of information, is minimized. This arrangement—isolating the computer from direct association with a card device—is called buffering: the computer is buffered from the card machine by an intermediate unit (Fig. 11-1).

PROBLEM OF CODE DIFFERENCE

The translation of an alphanumeric character into its two-digit code on input, and the translation of a two-digit coded alphanumeric character on output, is handled by a special translator in the Cardatron Control Unit. The

[1]For a detailed description of the Cardatron System and a complete list of the Cardatron instructions, refer to Operational Characteristics of the Burroughs 220, Bulletin 5020.

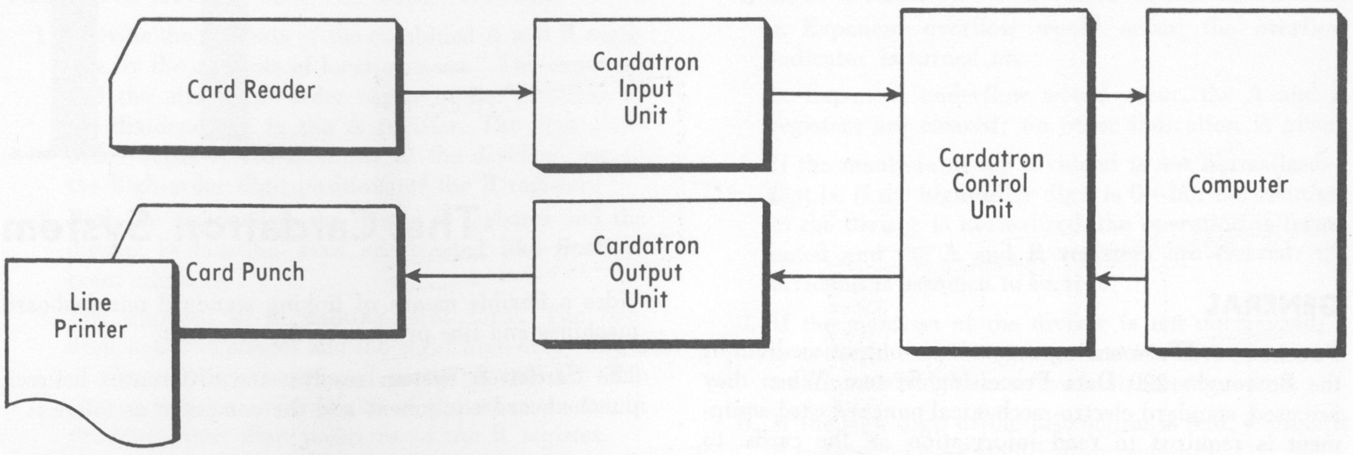


Figure 11-1. Buffering Operation

Cardatron Control Unit is notified to make the necessary translations by certain format digits in the format band selected by the incoming card or in the instruction causing the printing or punching of outgoing information.

PROBLEM OF WORD LENGTH

Compression or expansion of variable-length punched-card fields into fixed-length computer words is accomplished by the Cardatron System. Each Cardatron Input Unit stores format digits which control the grouping of information into computer words; each Cardatron Output Unit stores format digits which control the regrouping of information from the computer into variable-length card fields.

A set of format digits—sometimes called an editing control stream—is assembled for each type of card to be read or punched, or page to be printed. These sets of format digits need not be the same for input and output; information can be read from cards with one type of format and information can be punched or printed with entirely different formats.

Each Input Unit and each Output Unit can store five sets of format digits. On input, a sixth editing control stream can be selected by means of a toggle. This editing control stream is for straight numeric format.

Because of the translating and format-control functions of the Cardatron System, information on punched cards need not be entered in a special form for computer input, nor is the computer restricted to handling numeric information. Also, the buffering function increases the speed of punched-card data handling.

The Cardatron System is built around the Cardatron Control Unit. The Control Unit controls any combination of Input and Output Units up to a maximum of seven. For example, a full Cardatron System might consist of a Cardatron Control Unit, three Input Units connected to three card reading devices, and four Output Units connected to three line printers and a card punch (Fig. 11-2).

In addition, the Cardatron Control Unit provides the information pathway between the Cardatron Input and Output Units and the computer.

THE BUFFER DRUM

Each Input and Output Unit has a magnetic drum—called a buffer drum—to store information. Each drum has an oxide coating on its surface similar to that on magnetic tape; on this surface spots are magnetized to represent information. The surface of the drum is divided into several channels or bands which run around it: one information band and five format bands (Fig. 11-3).

INFORMATION BAND

The information band stores the contents of one punched card or line of edited information. Information on the information band is ready either to be read into the computer or transmitted to a card punch or line printer; all necessary code translations and word-length adjustments have already been made.

FORMAT BANDS

For each digit position on the information band, there is a corresponding digit position on each of the format bands. Specific format digits are written in these format-band positions; each of these digits is an instruction to the Cardatron telling it what to do with the corresponding digit of the information band. The digits of the format band instruct the Cardatron how to edit every column of the card. In addition, format digits may be included that insert either zeros or blank spaces for scaling or separating data. The format digits edit the information from punched cards or computer words, group the information into computer words for input, and spread it into card fields for output. There are special instructions in the Burroughs 220 vocabulary for loading format digits onto a specified format band.

Each format band on an input buffer drum contains the digits for editing one type of input card. Each format band on an output buffer drum contains the digits for

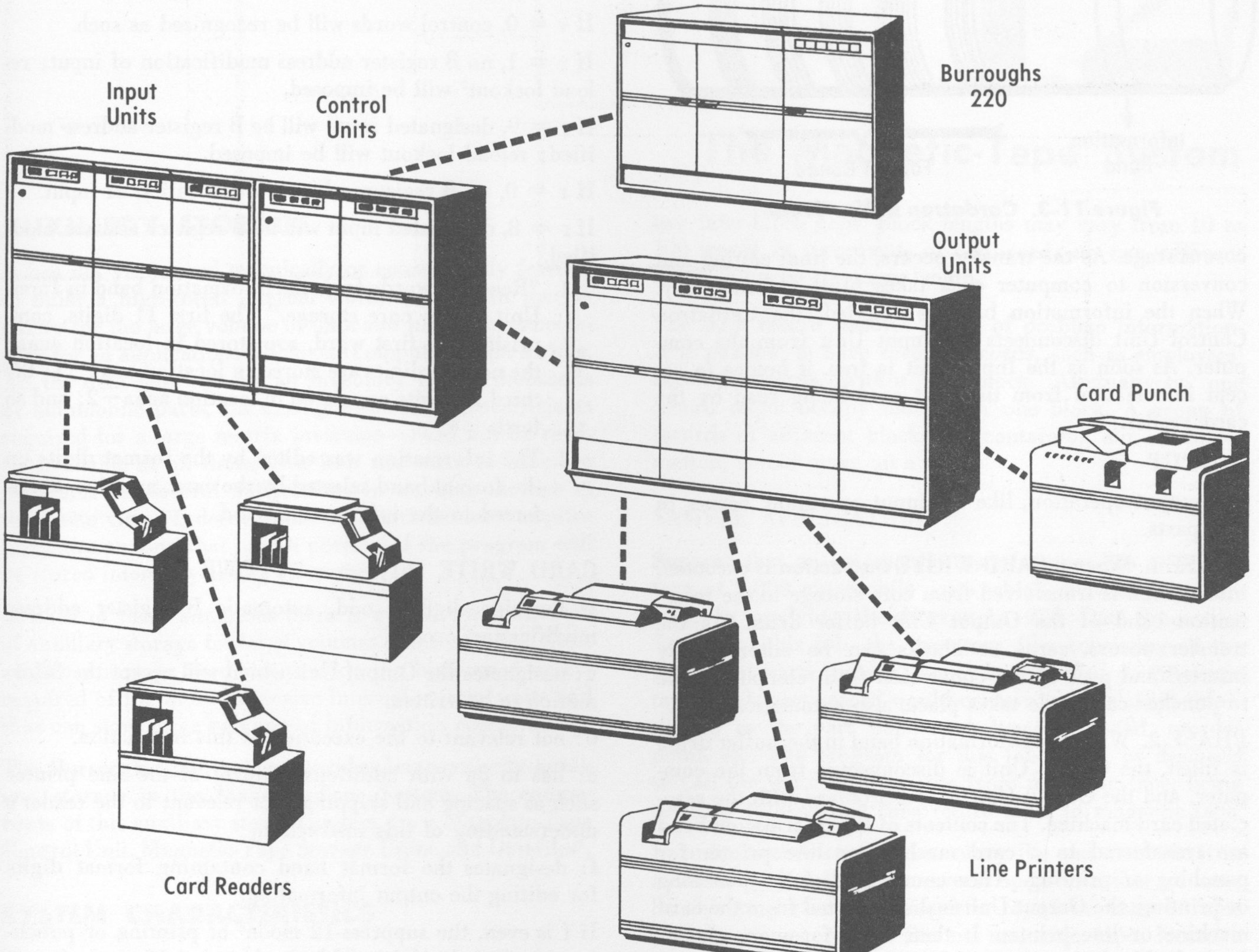


Figure 11-2. Cardatron System

producing one card or one printed line of information in a given format. Since each buffer drum has five format bands, any Input Unit can accept cards of up to six distinct formats (five plus the numeric) and any Output Unit can group information into any of five distinct formats.

On input, the format band that is to control a particular card is selected by a control punch in the card itself. On output, a format band is specified by a digit in the control portion of a CARD WRITE instruction. The digits 0, 1, 2, and 3 are used for format control.

INFORMATION FLOW

INPUT

The input operation of the Cardatron System—that is, the reading of information from punched cards into the computer—occurs in two steps: card to buffer drum, and drum to core storage.

PHASE 1. During the first phase, the Input Unit is synchronized with the card-reading device. At this time, one card is read by the card device, and the information contained on the card is recorded on the information band of the Input Unit buffer drum. The information stays on the buffer drum until the computer is ready to use it. As the information is transferred from the card reader to the buffer drum, either zeros or blanks can be automatically inserted and part of the conversion from punched-card code to computer code takes place, also automatically.

PHASE 2. When a CARD READ instruction is executed by the computer, the contents of the information band are transferred to a set of consecutively addressed locations in

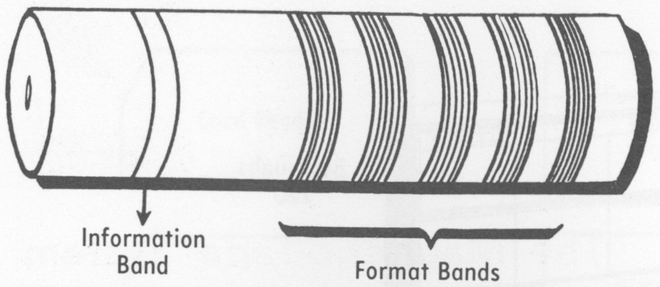


Figure 11-3. Cardatron Buffer Drum

core storage. As the transfer occurs, the final editing and conversion to computer code takes place automatically. When the information band is emptied, the Cardatron Control Unit disconnects the Input Unit from the computer. As soon as the Input Unit is free, it begins to accept information from the next card being read by the card reader.

OUTPUT

The output operation, like the input operation, occurs in two parts.

PHASE 1. When a CARD WRITE instruction is executed, information is transferred from core storage to the information band of the Output Unit buffer drum. As the transfer occurs, zeros or blanks can be automatically inserted and part of the conversion from computer code to punched-card code takes place, also automatically.

PHASE 2. When the information band in the buffer drum is filled, the Output Unit is disconnected from the computer, and the Output Unit is synchronized with the associated card machine. The contents of the information band are transferred to a card machine or line printer for punching or printing. After completion of the punching or printing, the Output Unit is disconnected from the card machine or line printer. It then waits for more signals from the Cardatron Control Unit.

SAMPLE INSTRUCTIONS

Following are the descriptions of the two Cardatron instructions selected as examples.

CARD READ (60) ± u 0 v r CRD aaaa

If the sign digit is odd, automatic B register address modification occurs.

u: designates the Cardatron Input Unit from which the information will be read.

0: not relevant to the execution of this instruction.

If v = 1, control words will not be recognized as such.

If v = 0, control words will be recognized as such.

If r = 1, no B register address modification of input; reload lockout[2] will be imposed.

If r = 9, designated input will be B register address modified; reload lockout will be imposed.

If r = 0, no B register address modification of input.

If r = 8, designated input will be B register address modified.

1. "Read the words from the information band in Input Unit u into core storage." The first 11 digits, comprising the first word, are stored in location aaaa; the next 11 digits are stored in location aaaa−1; the third 11 digits are stored in location aaaa−2; and so forth.
2. The information was edited by the format digits on the format band selected by the card, and then transferred to the information band.

CARD WRITE (61) ± u 0 c f CWR aaaa

If the sign digit is odd, automatic B register address modification occurs.

u: designates the Output Unit which will accept the information to be written.

0: not relevant to the execution of this instruction.

c: has to do with additional control of the line printer such as spacing and skipping; not relevant to the reader's understanding of this instruction.

f: designates the format band containing format digits for editing the output information.

If f is even, the suppress-12 mode[3] of printing or punching is selected; if f is odd, the suppress-12 mode is not selected.

1. "Write the contents of up to 29 core storage locations onto the information band in Output Unit u." The contents of location aaaa are the first to be written; the contents of location aaaa−1 are the second to be written; the contents of location aaaa−2 are written next; and so forth.
2. The total number of digits transferred from core storage to the information band depends on the configuration of the digits of format band f.

[2]If reload lockout is imposed, the contents of the next card will not be read onto the information band; that is, transfer of the information is inhibited.

[3]The suppress-12 mode has to do with overpunching the sign digits of numeric words. A detailed description of this operation is included in Operational Characteristics of the Burroughs 220, Bulletin 5020.

12 The Magnetic-Tape System

AUXILIARY STORAGE

It has not yet proved technically or economically feasible to build a high-speed internal storage unit with the capacity for the large volume of data and instructions necessary for all applications which one computer might handle. All the data required for an inventory file of thousands of automobile parts, for example—or all the coefficients required for a large matrix inversion—need not be readily available at all times. Nor is it necessary to have the programs for several applications in working storage at the same time. Indeed, some applications may require programs so long that only a portion of the program will be stored internally at any given time.

In each of these situations there is a need for some type of auxiliary storage for large volumes of information. Such auxiliary storage need not provide the high-speed access required of the more expensive internal storage units and thus can store large volumes of information economically.

The Burroughs 220 system provides large-capacity auxiliary storage in the Magnetic-Tape System. The components of this auxiliary storage system are a Magnetic-Tape Control Unit, Magnetic-Tape Storage Units, and Datafiles[1].

SYSTEM CHARACTERISTICS

The operation of the Magnetic-Tape System[2] is always initiated by the computer program. The Magnetic-Tape Control Unit houses the components for directing magnetic-tape operations once they have been initiated by instructions from the computer. Thus the computer communicates with the Magnetic-Tape Control Unit, which in turn relays the messages to the tape-handling units.

A Magnetic-Tape Storage Unit stores information on reels containing up to 3500 feet of tape with a capacity of up to approximately 1,400,000 11-digit words. The Datafile contains 50 lengths of magnetic tape, each 250 feet in length, and each hanging freely in its own bin. A single Datafile can store up to approximately 5,000,000 11-digit words. Each Burroughs 220 system can include up to ten Datafiles and Magnetic-Tape Storage Units in any combination.

Information is recorded on magnetic tape in separate units called blocks. A block is a group of words between two inter-block gaps. Block lengths may vary from 10 to 100 words in increments of one word; no two adjacent blocks need be the same length.

The term record denotes a unit of problem information. It is possible to have several records, such as employees' earning records, within one block. Alternatively, one record might occupy more than one block. A group of records in adjacent blocks, all containing similar information, would make up a file.

SAMPLE APPLICATION

Suppose that an automobile manufacturer has an inventory of 5,000 automobile parts which must be kept current. In applying the Burroughs 220 Data-Processing System to this job, the inventory file, consisting of the individual parts records, would be recorded on magnetic tape. Each part record would contain such relevant information as the part number, balance on hand, reorder point, etc. The program for this job would be designed to call for individual records from magnetic tape, bring them up to date, and then return them to magnetic tape.

In order to construct the inventory file and maintain it in a current status, there are several operations that we must be able to perform on magnetic tape: we must be able initially to record the file on magnetic tape, locate a particular record when it is needed, read the record from magnetic tape into core storage, and replace the updated record on magnetic tape. The Burroughs 220 system provides magnetic-tape instructions for each of these operations. The file would be written initially onto freshly edited tape; that is, tape that has been checked for flaws and on which all flaws have been indicated so that they will be skipped during writing.

A single instruction could write from one to ten blocks of the file. The first word of each block is regarded as its address. This address, like the address of a storage location, serves as a marker for locating the block. For ease of locating a block, the blocks written in any one lane must be recorded in order of increasing address, from beginning to end of the tape. (Refer to Appendix D for magnetic-tape format.)

Associated with each block written on the tape is a preface. The preface does not contain part of the information in

[1]Trademark of the Burroughs Corporation.

[2]For a detailed description of the Magnetic-Tape System and a complete list of the magnetic-tape instructions, refer to Operational Characteristics of the Burroughs 220, Bulletin 5020.

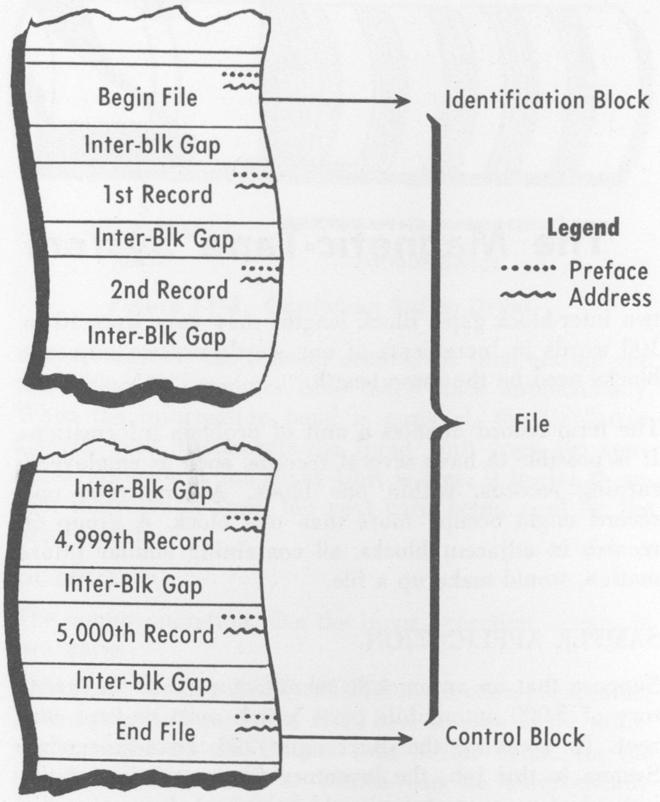


Figure 12-1. Inventory File of Automobile Parts

the block but is a word immediately preceding the block: it indicates the number of words in the block.

The inventory file would be bounded by unique blocks. The block at the beginning of the file is an identification block: it would tell the computer that this is the inventory file of 5,000 automobile parts. The block at the end of the file is called a control block: it would serve to notify the tape control unit if the end of the file is reached during an operation on the file. This block can contain information pertinent to the file for checking purposes: it might contain such information as control totals, the number of blocks in the file, etc. (Fig. 12-1 shows the inventory file, using one-block records.)

If the program to handle the daily inventory updating operation were already in core storage, this program would activate the tape control unit to locate the records—one at a time—for the parts used that day, read these records into core storage and update the balance on hand for each. If the balance on hand has fallen below the reorder point this information is printed out before the record is replaced on magnetic tape.

SEARCHING AND SCANNING. The question arises of how an individual parts record can be located in a file. There are two methods of locating a particular block on Burroughs 220 magnetic tape.

The first method is called searching. An instruction in the program can tell the tape control unit to search for the block with the address it specifies. The address specified by the instruction is called the search key. The searching operation consists of automatically comparing the search key with the first word of each block in a file until an equal comparison is obtained.

The second method of locating a particular block is called scanning. An instruction in the program can tell the tape control unit to scan all of the blocks in a file for those belonging to the category it specifies. The category specified by the instruction is called the scan key. Any one of the first ten words of a block may be used to designate a specific category; this word is called the category code. The scanning operation consists of automatically comparing the scan key with the word of each block which contains the category code until an equal comparison is obtained. For example, from the automobile parts inventory file we may wish to examine the records of all types of windshield wipers.

The search and scan operations can be partial-word operations: a partial-word search or scan key can be specified by the program. The tape control unit would then search or scan for the partial-word address or partial-word category code specified.

Once a search or scan operation has been initiated by the program in core storage, the tape control unit can carry on the operation independently of the computer.

SELECTIVE UPDATING. When a specified part record has been located in the file, it must be read into core storage before the updating process can begin. A single instruction in the program can read from one to ten blocks into core storage, starting with the location whose address is specified by the instruction.

The reading operation is one instance where the preface of a block is utilized. If the blocks to be read are all the same length, one type of reading operation takes place which does not necessarily include the preface: the first word read and sent to core storage is the address of the first block. (The preface could be read in, but there is no need for it.) The address is written in the location specified by the reading instruction—succeeding words of the first block and all following specified blocks are written into consecutively addressed locations following the location containing the address of the first block.

If the blocks to be read are of different lengths, another type of reading operation takes place: the first word read and sent to core storage is the preface of the first block. In this case, the preface of each block must be read into core storage immediately preceding the address of the block so that the proper number of storage locations for the words of the block may be allocated.

After the part record has been updated in core storage, the block or blocks containing the record must be recorded in the original position on the tape; that is, the updated record must be written over the old record. Before overwriting begins, the preface is checked. This is to insure against overwriting a block with a different number of words and is a check that the updated block is being written in the proper place.

Overwriting the updated record on magnetic tape completes the daily updating process. The method we have described is a selective updating process: individual records are selected, located, and updated.

TOTAL UPDATING. A total-update operation consists of reading each block of a file into core storage, updating it, then writing it onto a second tape. Thus after the operation there are two copies of the file: the original file and the new, updated one.

For example, in the automobile manufacturing industry, a new year brings a new model automobile containing many new parts. Also, a new year may bring the discontinuation of a number of parts that had previously been stocked. Therefore, the parts inventory must be brought up to date totally by inserting a record for each new part and deleting the records of all outdated parts.

Assume that the manufacturer has decided to have a large yearly inventory-file maintenance program run, during which the insertions and deletions are made to the file. The program to handle this yearly run would be designed to search for the first record of the file and read that record and each subsequent record of the file in turn into core storage. In core storage the program would check each record to see if it is to be deleted or if there is to be an insertion in the file between the previous record and the record being checked.

If the record is to be deleted, it simply is not written onto the new tape—the next record is read into core storage to be checked. If there is to be an insertion in the file, the new record to be inserted is written onto the new tape—the record in storage is then written onto the new tape immediately following the insertion.

MAGNETIC-TAPE OPERATION. Since blocks are recorded on magnetic tape in order of increasing address, many blocks may have to be passed on the tape before a particular record is located. This time-consuming operation has been alleviated by the use of two parallel lanes on Burroughs 220 tape and 100 parallel lanes in the Datafile. By means of a single instruction, the read-write head may be positioned in a particular lane, ready to perform operations on the blocks in that lane (Appendix D). This is done without moving the tape.

Other instructions provide means of positioning magnetic tape, interrogating tape-handling units to determine if they are ready for use, rewinding reels of magnetic tape, etc., under program control. These features of the Burroughs 220 Magnetic-Tape System augment the flexibility and efficiency of the system.

SAMPLE INSTRUCTIONS

Following are descriptions of two magnetic-tape instructions selected as examples.

MAGNETIC-TAPE READ (52)

$\pm$ u n 0 v MRD aaaa

If the sign digit is odd, automatic B register address modification occurs.

u designates the tape-handling unit from which the information will be read.

n specifies the number of blocks to be read: n = 1 means read one block, n = 0 means read the maximum of ten blocks.

0: not relevant to the execution of this instruction.

v = 8 or 9: designated input will be B register address modified.

1. "Read into core storage n blocks from magnetic-tape-handling-unit u." The first word of the first block read is stored in location aaaa. The remaining words of that block and the words of the following blocks are stored in consecutively addressed locations beginning with location aaaa+1.
2. The lane from which the blocks of information are read is the lane specified by the last magnetic-tape instruction referring to a specific lane.
3. A control block will be recognized as such if encountered during the execution of this instruction.[3]
4. An end-of-tape block will be recognized as such if encountered during the execution of this instruction.[3]

MAGNETIC-TAPE SCAN (51)

$\pm$ u h h k MTC aaaa

If the sign digit is odd, automatic B register address modification occurs.

If the sign digit is 4 or 5, another variation of the instruction is executed.

u designates the tape-handling unit.

hh: if u is a Tape Storage Unit, lane 0 is selected if hh is even and lane 1 is selected if hh is odd. If u is a Datafile, lane hh is selected.

k specifies the word of the block which contains the category code.

1. "Scan the lane specified by hh for a block whose category code in the kth word is equal to the scan key which is stored in location aaaa."
2. When a block whose category code is equal to the scan key is found, the scanning operation terminates with the read-write head positioned to read the sought-for block.
3. Once this instruction has been initiated by the computer, the scanning operation is carried out under control of the Magnetic-Tape Control Unit, independently of the computer.

[3]Ibid.

13 An Introduction to Automatic Coding

GENERAL

The preceding chapters have discussed the art of coding using the instructions of the Burroughs 220 vocabulary.

Such "machine-language" coding is the basic language for communicating with a computer system. However, there are now ways to simplify the job of coding—methods that make use of the computer itself to help in preparing programs. The several kinds of special programs that can be used to produce final programs have their own names and terminology; in general, they are all referred to as "automatic coding."

The classes of automatic coding schemes that will be mentioned in this chapter are assemblers, interpreters, generators, and compilers. Each class has a fairly well-defined purpose, but the programs within a class may differ widely in details. One compiler, for example, may be quite different from another, and compilers include both assembly routines and generators.

Before describing these four different kinds of programs, we should consider what purposes they are meant to serve and why and how they evolved.

When the management of an organization decides to acquire a computing system, they usually have one or both of two goals in mind: to do certain jobs faster and at less cost, or to take on jobs that otherwise could not be done at all.

The computer in a scientific installation, for example, would be used to reduce the time and the cost of complex computations. It would also make possible the solving of problems of such length and complexity that they could not even be undertaken without the computer.

A business application would be designed to speed up the handling of paper work, such as that involved in billing and inventory control. The management might also be able to take advantage of the speed of the computer to use the information assembled for the application to prepare special reports not otherwise feasible.

In either kind of application, the computing system must be supported by an immense amount of coding, especially during the early stages of installation. To reduce the time required for this coding, the user is likely to consider automatic coding. He can make use of existing routines from the computer manufacturer; he will also consider writing routines suited to his own special purposes. These "service routines" will require the investment of programming and coding time—an investment that will pay dividends later by saving both coding and computer time. The routines would include automonitors—programs that trace other programs and show, in printed form, a record of instructions executed and the results of their execution. Special routines that standardize the form and procedure of input and output operations, for example, can be developed and later become a part of more complex and comprehensive programs.

ASSEMBLERS

One of these major programs is the assembler. An assembly routine can eliminate many of the coder's difficulties. Consider, for example, the problem of inserting and deleting instructions in a section of a program that was thought to be completed.

The reader will recall that the instructions in the sample programs in this handbook are stored in consecutive locations in core storage. Yet the coder cannot know in advance—when he's working on a long program—exactly where every instruction and data word should be stored. He may, for example, realize while he's working on the fifth page of his coding sheets that he should have included another instruction on the first page. He may have forgotten it then, or he may not have realized until now that it would be required.

How is he to insert it at this stage? He started his program with location 0000 and he has used every location in sequence through 0100. To make an insertion in, say, location 0010 he would have to move every instruction from 0011 through 0100 to the location with the next higher address. But if he does this he has created new problems. Instructions regularly refer to other instructions and, since some of those referred to have been moved, the locations specified by the address portions of some instructions will also have to be changed. Therefore the assembler will include a means of avoiding these problems entirely; it will allow the use of symbolic addressing. Instead of coding the usual way, using actual (absolute) locations, the coder will be able to use symbolic addresses and the assembler will assign the absolute addresses.

How the symbolic addresses look will depend on the design of the assembler. They could be either alphabetic or numeric. One approach is to use numeric symbolic addresses in such a way that off-line card sorting can be used to reduce computer time in assigning absolute addresses. Another approach is to use extra digits when needed. Suppose, for example, that the coder has just finished writing an instruction for location 0100 when he decides he would like to insert an instruction in location 0010. All he need do is put the new instruction on the next line of his coding form and assign it location 0010.1; the assembly routine will insert it in the proper place—right after location 0010—and assign the absolute addresses, for the instructions that follow, in sequence. Even if the coder wanted to insert more than ten instructions between 0010 and 0011, he need only add another digit: 0010.11.

Symbolic addresses written this way would be called relative to zero. That is, the assembly routine would assign absolute addresses beginning with the first location in storage—0000. Symbolic addresses can also be made relative to other starting points; thus they can be grouped into regions. One region might be used for all constants, another for temporary storage, and so forth. Each region can be assigned a number and the coder will preface each entry in his program with its proper region. When the program is assembled, absolute addresses will be assigned beginning with the first location of each region.

Regions are especially useful when a long and complex program is to be written. The work can be divided into logical segments so that several coders can work at the same time, each on a different segment. The assembly routine will fit the sections together.

Another problem that the assembly routine can solve easily concerns the use of alphabetic operation codes. We have seen that it is simpler to use these mnemonic operation codes than the numeric codes they stand for. But these alphabetic characters must be translated to numbers—either by the coder or keypunch operator—before the computer understands them. By including in the assembler a table of corresponding alphabetic and numeric operation codes, the assembler itself can look them up and do the translating. Thus the coder is relieved of another chore.

When the assembly routine is completed and in use, it will produce a final (object) program that can be used at once or stored for later use. The object program can be recorded on cards, magnetic tape, or paper tape.

These are some of the features of a typical assembler. Many others could be added, such as checking facilities. An assembly routine might include, for example, means for checking all magnetic-tape instructions to see that the coder has included a digit to designate the tape unit to be used. This digit could be checked to see that it is not larger than the total number of tape units available to the system. These refinements, however—like the basic features of the assembler—are determined by the nature of the computer system and its application.

STAR 1

Star 1, an assembly routine used with the Burroughs 220, includes many of the characteristics described and other features.

It allows the coder to use either symbolic notation or machine language and will accept both paper-tape and punched-card input. Certain kinds of errors are recognized and an indication printed for the use of the operator. They include: input out of sequence, improper operation codes, storage overflow, improper field designation, etc.

Both printed and punched-card output are produced by the Star 1 routine, providing a complete record of the original symbolic input as well as the final assembled program.

INTERPRETERS

Interpretive routines are used to convert programs from one language to another and to execute the new program as it is produced. The language they recognize may be artificial—constructed for a special purpose—or it may be the machine language of another computer.

It is possible, for example, to make up a vocabulary of instructions for a hypothetical computer that would be very simple to program. Then an interpreter could be written to execute instructions written in this artificial language.

There is seldom a direct correspondence between an instruction to be interpreted and the machine language resulting from the interpretation: one original instruction may require several in the language of the computer being used. Unlike an assembler, an interpreter executes the final program as it is prepared; this object program is not recorded for future use. Therefore the complete interpretation must be done again each time the program is run.

THE BURROUGHS 205-220 SIMULATOR

An interpretive routine devised to accept the machine language of one computer and translate it to that of another is called a simulator.

One of the simulators available for the Burroughs 220 accepts the machine language of the Burroughs 205. A discussion of this simulator will illustrate the value of interpretive routines.

The 205-220 simulator was written to simplify the changeover of a computer installation from a Burroughs 205 to the larger Burroughs 220 system.

Such a simulator makes it possible to install the new computer without the necessity of first recoding all existing programs. With the simulator, programs written for the 205 can be run immediately on the 220. This procedure, of course, does not take full advantage of the much higher computation speed of the larger system. But it does allow the data-processing operation to continue with minimum interruption. Detailed coding can begin where the greatest advantage in speed can be realized, probably with subroutines. It is possible that those programs that are rarely used might never be recoded.

The simulator translates each Burroughs 205 instruction into one or more 220 instructions and executes the resulting instructions fast enough to maintain the regular operating speed of the smaller computer. However, the fact that in these circumstances the 220 operates no faster than the 205 illustrates the main weakness of interpretive routines, since the computation speed of the 220 is approximately ten times that of the 205.

While the simulator is operating a switch on the Console can be set to monitor the program. When the switch is set, a printer produces a printed record for each instruction simulated, showing its location, the instruction itself, and the contents of the simulated A, R, and B registers. The monitoring feature is used only occasionally, when the operator wants to check a portion of the program.

Interpretive routines are, in general, declining in popularity. They are still used regularly for simulation, but their other main purpose, providing a simpler language for the coder's use, is better served by compiling techniques. Compilers will be discussed later in this chapter.

GENERATORS

A generator is a program that produces a section of coding for a specific purpose. Generators may be included in assemblers and compilers, but they are also often written as separate programs.

The generating program is set up to allow the insertion of quantities to determine the details of the object program. Thus a generator written to produce search routines, for example, can generate an enormous variety of routines with details depending on the specifications—number of items, number of words per item, etc.

The routines produced by a generator may be used immediately or they may be stored as subroutines on magnetic tape or punched cards for future use.

A BINARY SEARCH GENERATOR

A generator has been written for the Burroughs 220 to produce binary search programs. The method of searching is called binary because each comparison of a reference with the key of a record in storage divides the number of keys still to be checked into two parts and eliminates one of the parts from further consideration.

The operator designates the parameters to be used by the generating routine; these values can be set up in the A and R registers from the Console control panel. For example, the four low-order digits of the A register represent the number of items in the table, digit positions 4, 5, and 6 represent the number of words in each item, and so forth. Other digit positions are used to designate such values as the length of record keys and the total words of storage available.

The routine checks for unreasonable values; if the number of items specified, for example, is 0 or 1, an error stop will occur.

Less than one second is required for the generator routine to produce a search program. The generated program may then be used immediately or punched into cards for later use.

COMPILERS

The compiler is the most comprehensive type of automatic coding system. It is designed to provide the person who originally poses the problem to be solved with a special language that is easy to learn, convenient to use, and acceptable to the computer.

In the brief history of computer installations, most of the coding has been done by specialists—not by the engineers, scientists or businessmen who originated the problems. Sometimes this situation has been a matter of policy. Even when it was not, those who originated the problems were not likely to have both the time and the detailed knowledge of computer techniques essential to efficient coding. Thus they had to explain to the coders exactly what they wanted to do. And, since their problems are specialized and arise in widely different areas, communication is difficult; the coder cannot be expected to be acquainted with the subject matter of every area of an organization.

Compilers have been developed to eliminate these difficulties. The problem can be stated in the symbolic notation—the "problem-oriented" language; the computer is used to produce the final machine-language program.

A compiler consists of two distinct parts: a system of symbolic notation and the machine-language routine that translates this notation and produces the final program.

We are most concerned here with the special notation and the over-all procedure from problem to object program. The compiler routine that does the translating will not be discussed in detail; it is a machine-language program and therefore similar to those suggested in other chapters—although extremely long, complex, and ingenious.

COMPILER NOTATION

Symbolic notation of various kinds has already been introduced in this handbook. The machine-language instructions themselves are a form of symbolic notation; the number 12, for example, can represent the phrase, "Add a number from storage to the contents of the A register." And the digit 1 in the sign position of a word can mean, "This is a negative quantity."

Compiler symbolic notation, however, is different in that the symbols are related directly to the problem and only indirectly to the computer. Standard English words are used, as well as combinations of numbers, letters, and special characters. Words such as "PRINT" or "HALT" might be used; arithmetic operations could be shown by the conventional $+$, $-$, $\times$, and $\div$; parentheses could be used, just as in algebra, to group expressions.

The compiler language allows these symbols to be grouped into "statements." There are two general classifications of compiler statements: arithmetic and logical.

Arithmetic statements describe the basic arithmetic operations to be done and define the data to be used in doing them.

Logical statements describe the sequence of operations. As in regular coding, this sequence may be complicated. A series of operations may be repeated a number of times, depending on the result of tests; a choice of one of several branches to further series of operations is made, depending on the outcome of comparisons. Therefore, compilers supply several kinds of logical statements for the user to choose from according to his needs; he may use each kind during the course of preparing his program.

To see how these logical statements might be used, consider the problem of transferring control, as discussed in the sections on looping and branching. In Burroughs 220 language, the coder chooses an instruction that calls for transferring control to a given location by specifying the operation code, control digits, and the address of the location to be next in sequence. His choice depends on knowledge of what the results will be after a previous instruction is executed—such as overflow, comparison high, etc.

In compiler notation, he would have written a series of statements, each with an identification number or letter. To transfer control, he might only have to write "GO TO 25." The compiler routine would assign the instructions with the proper addresses, operation codes, and control digits to effect the transfer.

USING A COMPILER

Whether a compiler is to be used or not, the first few steps of preparing a problem for solution by the computer are the same.

The problem must first be stated mathematically or logically. Then the general method of solution must be chosen and the problem defined in terms of the method.

At this point, the person who poses the problem might want to consult the staff of the computing center and decide, with their help, whether a compiler should be used. If, for example, the problem is one that will recur frequently with only minor changes, machine-language coding might be practical. Usually a program coded directly is more efficient; it saves computer time at the expense of coding time. (This difference in efficiency, however, may soon disappear as compiling techniques are further refined.) The decision would probably be made at this point, before flow-charting, because the nature of the compiler may influence the preparation of the flow chart.

The next step will be the preparation of the statement program in the symbolic notation of the compiler being used. When this is completed, the originator of the problem has, in effect, completed the programming.

Keypunch operators prepare the cards (or paper tape) from the statement program forms used by the originator. The keypunching, of course, must be done with extreme care; every period, comma, or other symbol will have a special meaning to the compiling routine.

The cards will then be given to the computer operator for compilation. The compiling routine will first check for invalid input—such as unacceptable combinations of symbols. If some are found, the portion of the statement program in error will be re-examined for keypunching and logical errors and new cards punched in the correct form.

Compiling will then begin and the object program will be produced.

As a final check, before entering the data and running the object program, a test problem that uses all possible branches may be run and the results compared to the known answer of the test problem.

The object program is then ready for use.

OTHER PROGRAMMING AIDS

Automatic coding methods are constantly being developed by both manufacturers and users of the equipment.

Routines available for the Burroughs 220 include, for example, tracing or monitoring programs used for checking out programs. Standard routines are provided for loading punched cards, subroutines for mathematical functions; such subroutines can be referred to by a program and included as parts of assemblers and compilers.

Present developments in automatic coding are toward the writing of compilers with a symbolic notation approaching more and more closely the language used to state problems—mathematics for scientific applications, English for the use of business.

Appendices

A. HOW INFORMATION IS REPRESENTED IN THE BURROUGHS 220

The Burroughs 220, like most digital computers, employs the binary number system for representation of information within the computer. This number system is built around only two digits, zero and one.

The reason for employing the binary number system is that a binary digit is so simple to represent physically, using a two-state device.

Groups of the two-state devices—each device representing one binary digit—are used to represent one decimal digit. Each two-state device within the group is assigned a specific value. The device is equal to its assigned value when it is in an on state. In its other state—off—it has a value of zero.

The Burroughs 220 has four devices per group: they are assigned the values 8, 4, 2, 1, respectively. Since the decimal digit represented by a group is the sum of the assigned values of the devices in an on state, this configuration could represent any number from 0 through 15. However, the Burroughs 220 uses the configuration to represent only the numbers 0 through 9 so that each group will represent just one decimal digit. The group of two-state devices representing a single decimal digit is called a decade.

This system of representing information is called the 8-4-2-1 binary-coded decimal system. (See Fig. A-1 for the various combinations of binary digits which represent the numbers 0 through 9.)

8	4	2	1	Digit Represented
Off	Off	Off	Off	0
Off	Off	Off	On	1
Off	Off	On	Off	2
Off	Off	On	On	3
Off	On	Off	Off	4
Off	On	Off	On	5
Off	On	On	Off	6
Off	On	On	On	7
On	Off	Off	Off	8
On	Off	Off	On	9

Figure A-1. Decimal Digit Representation

To represent a Burroughs 220 word—that is, 11 decimal digits—11 decades are required. Let us consider the data word + 4532 98 7604 (using 0 to represent the off state, 1 to represent on):

VALUE	+	4	5	3	2	9	8	7	6	0	4
8	0	0	0	0	0	1	1	0	0	0	0
4	0	1	1	0	0	0	0	1	1	0	1
2	0	0	0	1	1	0	0	1	1	0	0
1	0	0	1	1	0	1	0	1	0	0	0

B. HOW INFORMATION IS STORED IN THE BURROUGHS 220

The internal storage unit of the Burroughs 220 consists of a network of magnetic cores. A magnetic core is a small ring of ferro-magnetic material used as a two-state storage device. It can be magnetized in either one direction or the other: when magnetized in one direction, it is, logically speaking, "on"; magnetized in the opposite direction, it is "off" and stores a zero. In order to be magnetized in a specific direction, a core must be pulsed by a current from each of two wires, as shown in Fig. B-1. The representation is logical rather than electronic.

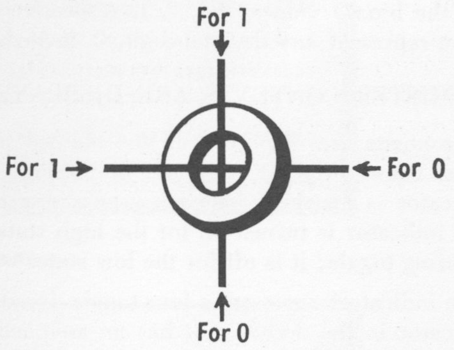


Figure B-1. Magnetic Core and Associated Wires

Thus we see that magnetic cores store information as it is represented—according to the rules of the binary number system. Each core stores one binary digit (1 or 0) of a binary-coded decimal digit. Four cores form a decade or one decimal digit (Fig. B-2).

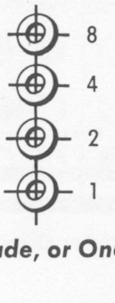


Figure B-2. Decade, or One Decimal Digit

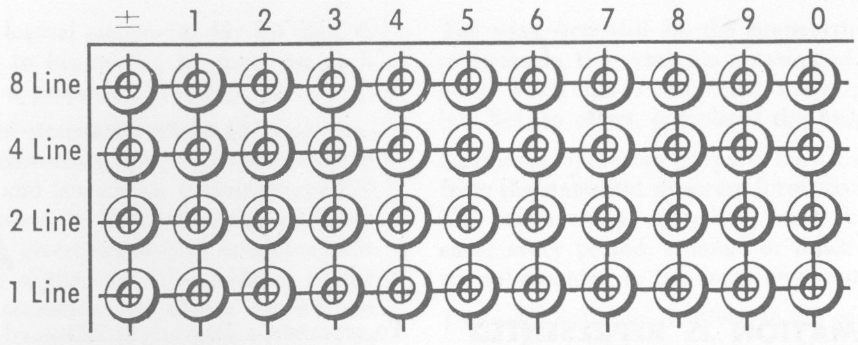


Figure B-3. Eleven Decades or One Word

A core is equal to its assigned value only when it is magnetized in the positive direction; otherwise it is equal to zero.

Forty-four cores store 44 binary digits and form 11 decades or one Burroughs 220 word (Fig. B-3). The internal storage unit of the Burroughs 220 is capable of storing up to 10,000 words.

C. BURROUGHS 220 REGISTERS

HOW INFORMATION IS STORED

Registers are made up of electronic circuits called toggles or "flip-flops." These are two-state devices that are made up of two vacuum tubes. Depending upon which of the tubes current flows through at a given time, the toggle is either in a "high" or a "low" state. If a toggle is in the high state, it is on or stores its assigned value; in the low state it stores a zero.

These toggles are grouped into four-toggle decades; they represent the binary values 8, 4, 2, 1, respectively. Each decade can represent any decimal digit, 0 through 9.

HOW REGISTER CONTENTS ARE DISPLAYED

Register contents are displayed on the control panel of the control console during computer operation. A small neon indicator is provided for each toggle in a register. The neon indicator is turned on for the high state of the corresponding toggle; it is off for the low state.

Four neon indicators represent a four-toggle decade. Each neon indicator in the decade row has an assigned value: from top to bottom their values are 8, 4, 2, and 1, respectively.

Groups of neon indicator decades make up the various register displays in the Burroughs 220 system. Each group

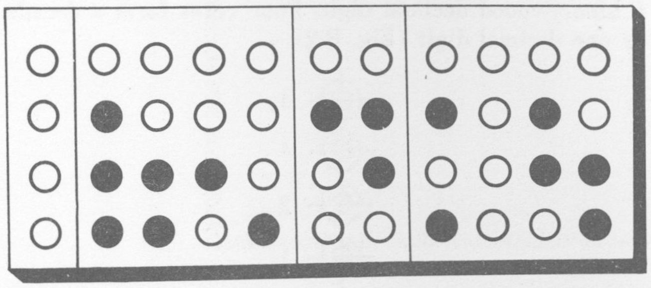


Figure C-1. The A Register Display of + 7321 46 5063

represents the contents of a specific register or group of toggles. The number + 7321 46 5063 in the A register would appear on the control panel of the Control Console as shown in Fig. C-1.

D. INPUT-OUTPUT MEDIA

PUNCHED PAPER TAPE

Punched paper tape is a specially treated strip of paper 7/8 inch wide in which a pattern of holes is punched. Numbers and letters are represented by a combination of the holes and blank spaces at each position along the length of the tape. The pattern of these holes and spaces in a particular row signifies a particular decimal digit or character.

The holes are punched in seven parallel channels along the length of the tape (Fig. D-1). These channels are functionally divided into three sections:

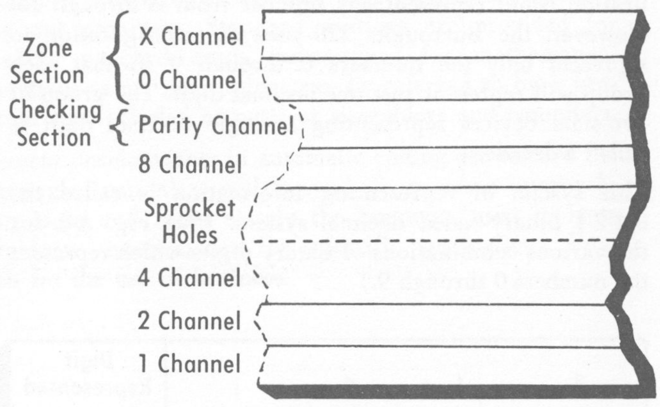


Figure D-1. Paper-Tape Structure

1. The zone section, consisting of two channels. These are designated as the X and 0 channels.
2. The checking section, consisting of one channel. This channel, called the parity-check channel, is used for checking purposes only.
3. The numeric section, consisting of four channels with assigned binary values of 8, 4, 2, and 1, respectively (thereby making a binary decade).

The paper tape also contains a continuous line of smaller holes running down the center. These are sprocket holes, used for control or timing purposes.

Digits are represented by one or more punches in the numeric section; letters of the alphabet and special characters by a combination of zone and numeric punches.

The mechanism used to enter into the computer the information punched into paper tape is a photoelectric reader. After a reel of punched paper tape has been mounted onto this photoreader, and the photoreader is activated, a servomechanism will drive the tape reels—moving the tape past a group of photo cells. Light is projected through the holes in the paper tape to form images of the holes on the photo cells. Each of these cells is connected to an amplifier which gives an input signal when a hole is read and no signal when there is a space.

The information read from paper tape is transmitted to core storage through a translator where the paper-tape code is translated into computer representation.

Information from core storage can be punched into paper tape. Each digit of information is sent through a translator where it is translated from computer representation to punched-paper-tape code. Then each digit is transmitted to a paper-tape punching device.

PUNCHED CARDS

The standard punched card contains 80 columns and 12 rows. The columns are numbered 1 through 80 (Fig. D-2); the rows are numbered 0 through 9 (Fig. D-3). Two additional rows appear as blank areas on the card (Fig. D-4).

The row just above the 0 row has several names. A punch in this row is called an 11 punch, an X punch, a zone punch, an overpunch, a control punch, or a minus-sign punch. The 11 punch or X punch is the most widely used term.

The top row also has several names: 12, R, high zone, plus-sign, etc. The 12 or R row, however, is the term most commonly used.

The 80 columns can be grouped into fields. A field is defined as one or more columns on the card containing a unit of information, for example, name field, amount field, identification field, description field, etc. (Fig. D-5).

When used with the Burroughs 220, a card will often be marked off in 10- or 11-column fields for computer words. If a 10-column field is used, an 11 or X punch can be punched over a numeric punch to denote a minus sign;

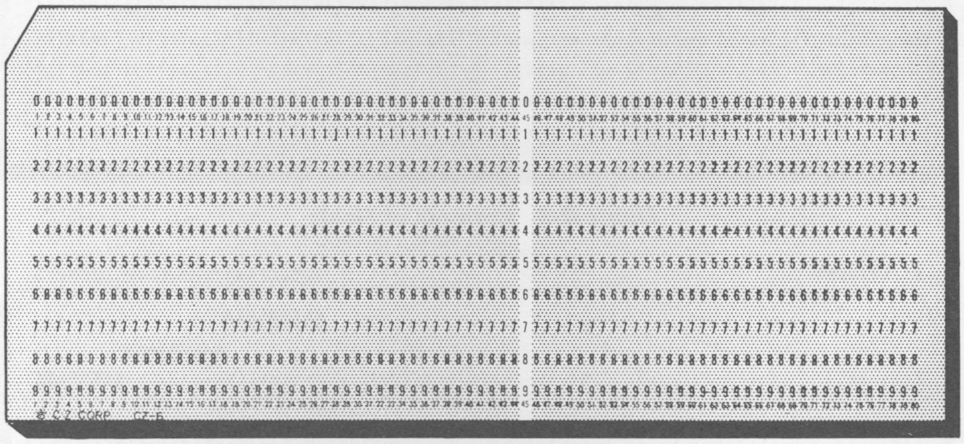


Figure D-2. Card Columns

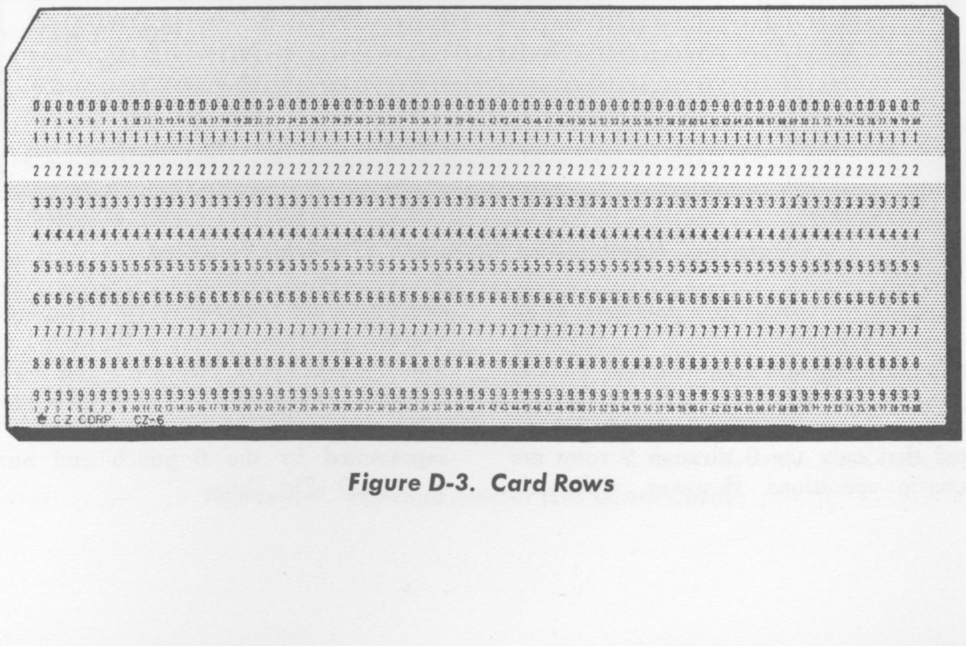


Figure D-3. Card Rows

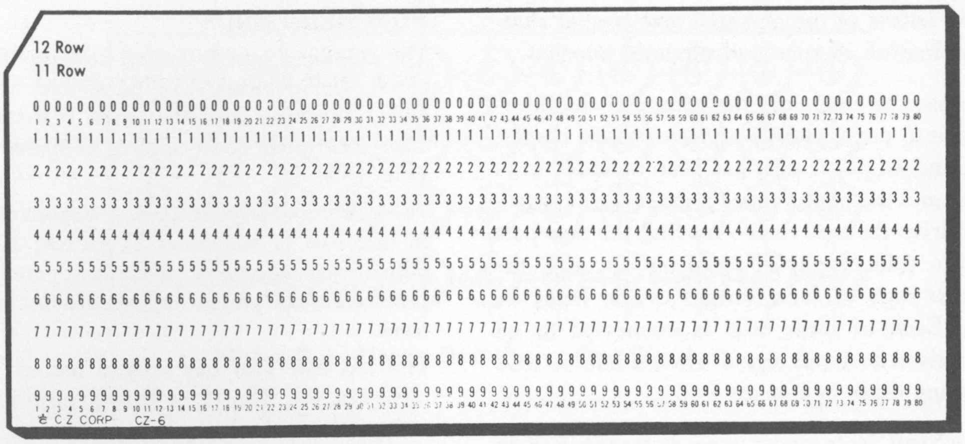


Figure D-4. Overpunch Rows

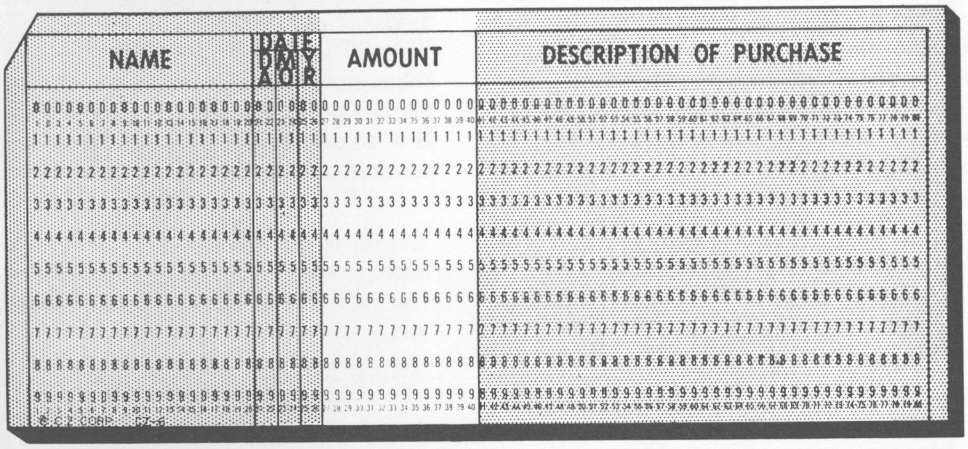


Figure D-5. Example of Fields

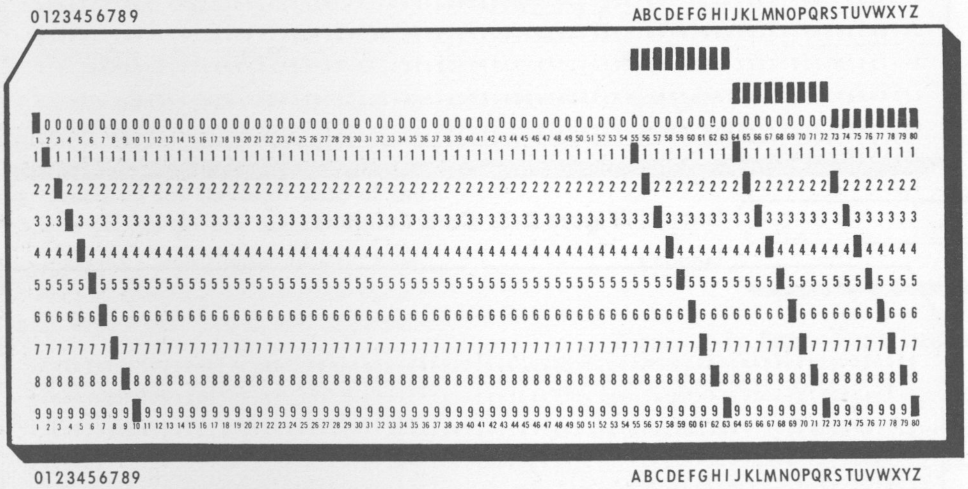


Figure D-6. Numeric and Alphabetic Character Punching

the lack of such a punch is considered a plus sign. If an 11-column field is used, a 1 punch in the first column of a field can denote a minus sign; a zero punch or a blank can denote a plus sign.

It should be noted that only the 0 through 9 rows are required for numeric operations. However, alphabetic characters are represented by a combination of zone and numeric punches. Letters A through I are represented by numeric punches 1 through 9 and a zone punch of 12; letters J through R are represented by an 11 punch and numeric punches 1 through 9; letters S through Z are represented by the 0 punch and numeric punches 2 through 9 (Fig. D-6).

Still other combinations of zone and numeric punches represent special characters and symbols, such as #, %, *, etc.

Note that the upper right-hand corner of the card shown in Fig. D-5 has been cut off. This is for identification purposes. As an example, a master card may have a left-hand corner cut and the detail cards a right-hand corner cut.

A master card usually contains lead information for the group of detail cards that follow it (Fig. D-7). For example, a master card might contain a man's name, clock number, rate of pay, social security number, etc. The detail cards would contain the numbers of the jobs on which he worked and the hours he worked on each job, etc.

After information has been punched into cards, that information must be entered into the computer from a card reader via the Cardatron System.

First, the cards are loaded into a hopper or loading receptacle on the card reader. With some card readers the cards are loaded so that the 9 row of the card deck is read first; with others, the 12 row is read first.

Once the cards have been loaded and fed into the read stations, the card reader is activated by the computer and the cards fed from the hopper. Each card passes over a metal drum or contact roll and under a row of metal read brushes. There is one brush for each of the 80 columns in a card. As each row of the card passes under the row of metal brushes, wherever a hole has been punched a sensing brush comes in contact with the metal drum or roller. Contact with the drum causes an electrical impulse. These impulses are recognized by the card reader as the values represented in the card and are transmitted to the Cardatron Input Unit. Here they are translated into information acceptable to the computer. From the Input Unit, the information is sent to the computer by way of the Control Unit.

When information stored within the computer is to be punched out on cards, it is transferred to the Cardatron Control Unit and then to the Output Unit. Here the information is translated into punched-card code. Then, as each blank card moves through the punching device, the information pertaining to a specific row in the card is transmitted to the card punch. The punch magnets within the device are activated, and all positions that are to be punched in a given row are punched simultaneously.

MAGNETIC TAPE

The magnetic tape used with the Burroughs 220 Magnetic Tape System is similar in operation and in form to the magnetic tape used in home recorders. The tape is a plastic strip, one side of which is coated with a magnetic oxide.

The tape can be considered as having provision for recording two lanes of information, parallel to one another, along the length of each tape. Each lane is divided into

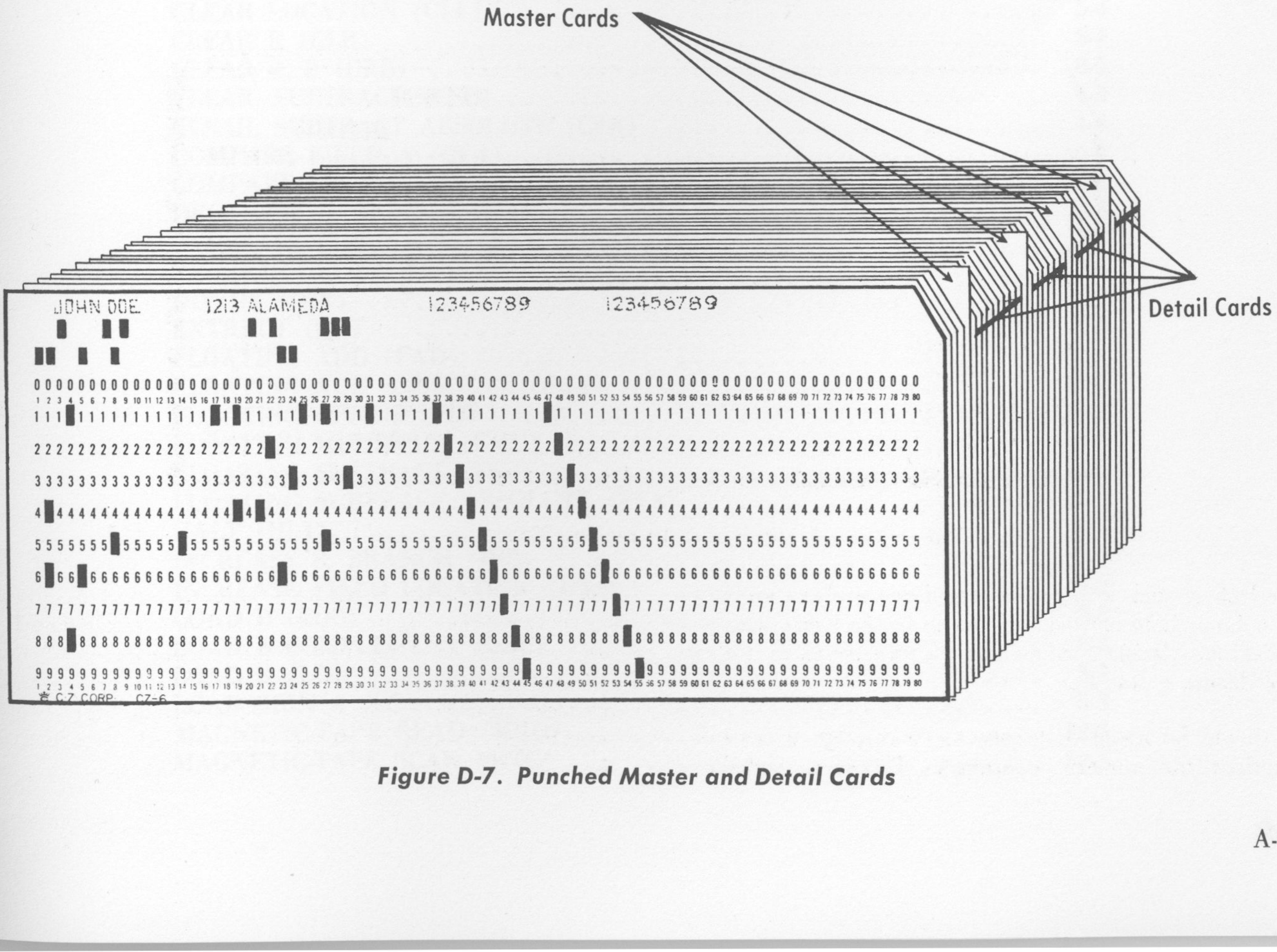


Figure D-7. Punched Master and Detail Cards channels: channels 1 through 4 are used to record the binary digits 8, 4, 2, 1; channel 5 is used as a checking channel; and channel 6 is used for control purposes (Fig. D-8).

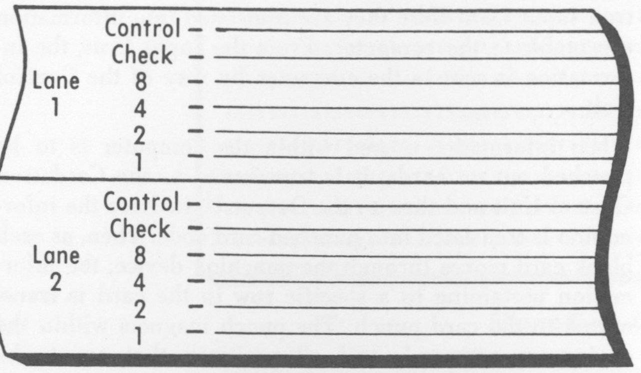


Figure D-8. Magnetic-Tape Structure

The binary digits in each channel are represented by magnetized spots placed or "written" on the tape surface. Utilizing two-state logic, a magnetized spot on the tape surface represents an on state, a non-magnetized spot represents an off state.

Words of information are written onto magnetic tape by special electromagnets called read-write heads. To write information onto the tape, the write head magnetizes spots on the tape as it passes by at high speed. To read information from magnetic tape, the read head senses the magnetized areas on the tape.

Words of information on magnetic tape are laid out in blocks. A block is defined as a group of words recorded serially without intervening blank spaces; a block may also be described as the information recorded between these blank spaces or inter-block gaps (Fig. D-9).

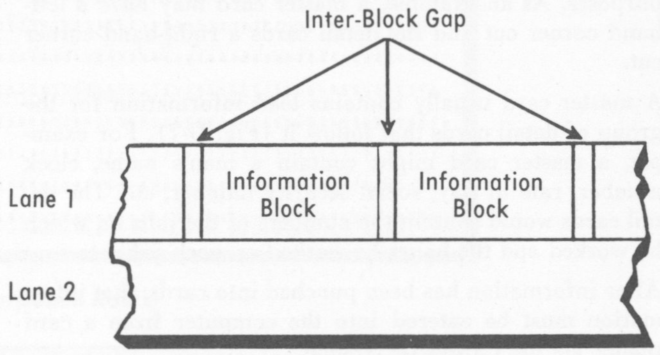


Figure D-9. Magnetic-Tape Blocks

PRINTED OUTPUT

Fanfold paper forms are used as output media by the character-at-a-time printer of the paper tape input-output system and by the line-at-a-time printer of the Cardatron input-output system.

For the character-at-a-time printer, information is taken from core storage, translated and printed out on the paper inserted in the printer. For the line printer, information from core storage is transmitted through the Cardatron for translation, to the line printer where it is printed out on the paper inserted in that printer.

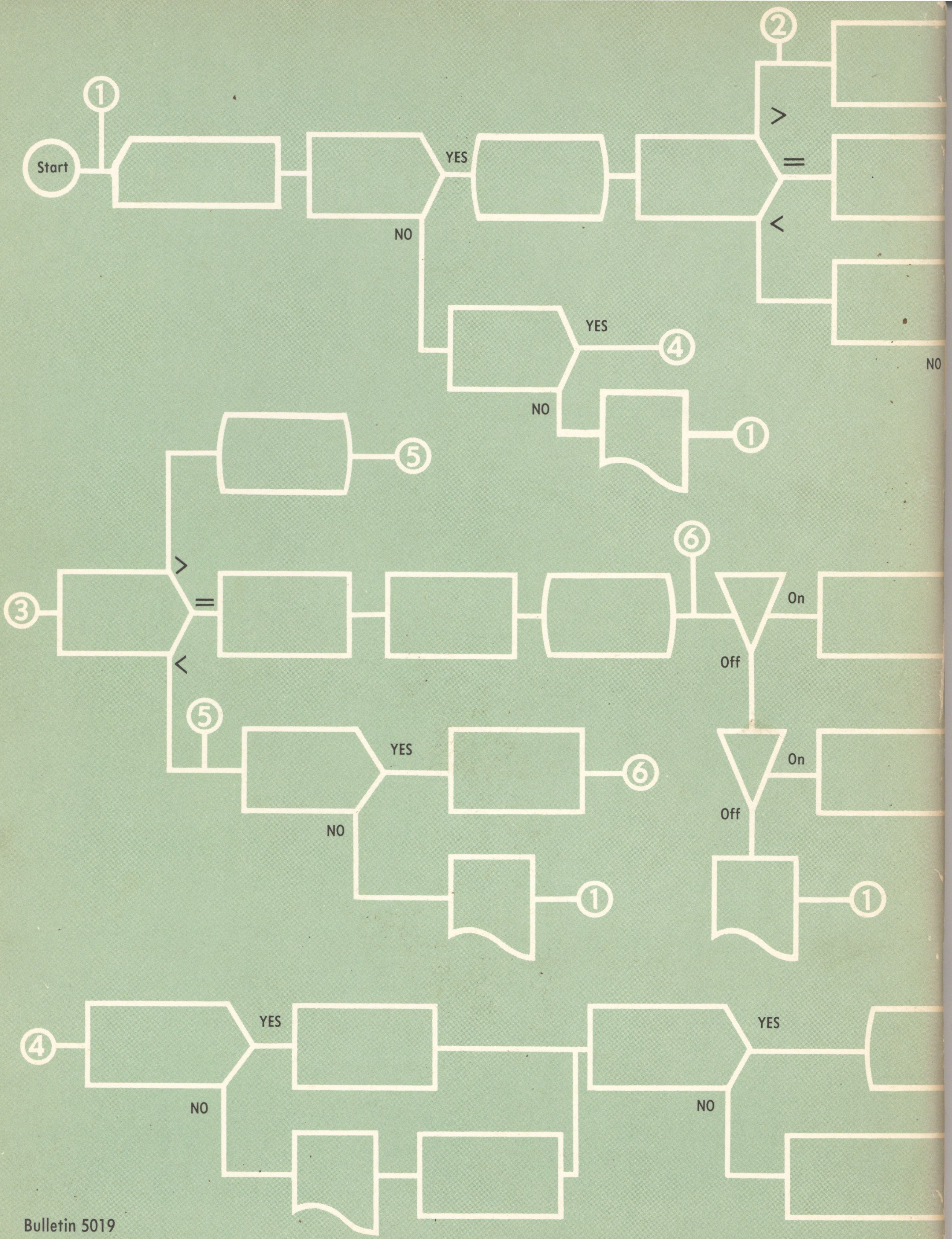
E. INDEX TO INSTRUCTIONS DESCRIBED IN THIS HANDBOOK

Note

For an explanation of instructions not included in this handbook, see Operational Characteristics of the Burroughs 220, Bulletin 5020.

Instruction and Mnemonic Operation Code

Page



Bulletin 5019