A Primer of Programming for Digital Computers

The IBM Type 650 Electronic Data Processing Machine—a typical stored program electronic computer. (*Reproduced by permission of International Business Machines Corporation.*)

A Primer of Programming for Digital Computers

MARSHAL H. WRUBEL III
Associate Professor of Astronomy
Indiana University

McGRAW-HILL BOOK COMPANY, INC.
New York Toronto London
1959

A PRIMER OF PROGRAMMING FOR DIGITAL COMPUTERS

 Library of Congress Catalog Card Number 58-13895

Preface

This book has been written for scientists, engineers, and their students who are planning to use electronic computers as tools of research. Its purpose is to answer the questions: what sort of problems can electronic computers solve and how does one go about using them? Throughout the book we shall regard the electronic calculator as a means to an end, rather than as an end in itself.

Even if the scientist does not expect to set the problem up for the machine without assistance, it is important for him to have a reasonable idea of what machines are like in order to explain his problem to a programmer and interpret it in terms suitable for machine computation.

It is not the purpose of this book to develop expert programmers. This is a book for beginners, and some of the subtleties of the art will be sacrificed for simplicity and clarity. Many simple examples are employed, chosen from problems encountered at the Research Computing Center of Indiana University. The reader is advised, however, to try to formulate problems which originate in his own field to illustrate the various points mentioned. In fact, it is best to approach the machine from the beginning with a problem in mind and to read this book in an attempt to solve it. But it should be emphasized that the experience gained by running even a single, simple problem of one's own is worth much more than reading any number of books on the subject.

To the beginner, there is a bewildering variety not only of machines but of different methods of programming a single machine. Calculators are, unfortunately, not yet as standardized as automobiles, but they each have certain recognizable parts and operations in common, and the general techniques described in this book are applicable to virtually any stored-program computer, regardless of manufacturer.

To illustrate most of the points discussed, we have chosen a single machine: the IBM Type 650. The reason for this choice was simply that machines of this type are widely distributed, and it is likely that anyone who wants to attempt to solve a problem in terms of the 650 can find a machine nearby on which the problem can possibly be tested and run. Special features of other machines will be mentioned when pertinent.

The book is divided into two sections: Elementary Programming and Advanced Programming. Many problems may be solved in their entirety by using the methods described in the first section. If possible, the reader should try to run a simple problem before continuing to the second section.

The text includes a Glossary of Terms so that the beginner, who may become confused by computer jargon, can refresh his memory as to the meaning of the most frequently used expressions.

I am indebted to the International Business Machines Corporation for permission to refer to punched card techniques and various methods of programming the IBM Type 650 throughout the text; and to Dr. V. M. Wolontis and the Bell Telephone Laboratories for permission to use their interpretive code.

I am also grateful to Mr. Dale J. Hall, Mr. Edward C. Olson, Dr. Richard L. Sears, and others connected with the Research Computing Center of Indiana University for detecting errors in the text.

Finally, it is a pleasure to record my gratitude to Miss Verona Sue Hughes for the care with which she prepared the manuscript.

Marshal H. Wrubel

Contents

Preface ix

PART ONE. ELEMENTARY PROGRAMMING

Chapter 1. General Introduction 3

1-1. What is an electronic digital computer? 3
1-2. The concept of a program 4
1-3. The program library 5
1-4. Numerical analysis and programming 6
1-5. The components of a computer 8
1-6. Some typical computer problems 9
1-7. When is a problem too large or too small for an electronic computer? 11

Chapter 2. The Approach to Simple Problems 13

2-1. Number systems 13
2-2. Word size 17
2-3. Data words: fixed- and floating-point representations 18
2-4. Addressable memory 21
2-5. Instruction words 22
2-6. Machine language 22
2-7. Interpretive programs 23
2-8. The Bell Telephone Laboratories code 24
2-9. Uses of storage 25
2-10. Arithmetic operations 26
2-11. The special role of address 000 29

2-12. Simple arithmetic problems 29
2-13. The punched card 33
2-14. Preparing punched cards 36
2-15. Loading 39
2-16. The READ instruction 42
2-17. The PUNCH instruction 43
2-18. Unconditional TRANSFER 43
2-19. A complete example 44
2-20. An additional example 46
2-21. Codes for common functions 46
Problems 49

Chapter 3. Introduction to Loops and Branches 51

3-1. Some general remarks 51
3-2. A simple counting loop 52
3-3. A loop with address modification 52
3-4. Loops without counting 54
3-5. A simple loop box 55
3-6. Programming the simple counting loop 59
3-7. A summation loop 60
3-8. The MOVE instruction 62
3-9. Terminating loops without counting 63
3-10. Branching without looping 66
3-11. Address modification loops without loop boxes 68
3-12. Symbolic representation of some of the preceding instructions . . 73
3-13. Summing an infinite series 73
3-14. Loops within loops; constructing a table with three parameters . 76
Problems 79

Chapter 4. Flow Diagrams, Subroutines, and the Program Library . . . 81

4-1. More about flow diagrams 81
4-2. Programming in boxes 82
4-3. Recurrent boxes; elementary subroutines 84
4-4. Subroutine entry and exit 86
4-5. A typical subroutine 90
4-6. Generalizing a subroutine 91
4-7. A complete example 93
4-8. Code words in general subroutines 93
4-9. Program write-ups 98
4-10. Maintaining a library 99
Problems 99

Chapter 5. Testing Programs 101

5-1. General comments 101
5-2. Checking in blocks; sample calculations 102
5-3. Memory dumps 103
5-4. Snapshots 104
5-5. Inserting instructions 106
5-6. Tracing 107
5-7. Stop codes 110
5-8. Comparison methods 112
5-9. Address search 113
5-10. Testing programs at the console 114
5-11. Tracing loops 115
5-12. Testing programs containing subroutines 116
5-13. Testing a subroutine 117
Problems 118

Chapter 6. Automatic Programming 119

6-1. General concepts 119
6-2. IT, FORTRAN, and FORTRANSIT 122
6-3. Symbolic variables 123
6-4. Constants 124
6-5. The concept of a "statement" 125
6-6. Source program and object program 128
6-7. Arithmetic symbols 129
6-8. Arithmetic statements 131
6-9. Special functions 134
6-10. Input and output statements 136
6-11. Unconditional transfer 138
6-12. PAUSE and STOP 139
6-13. Examples of simple programs 139
6-14. The loop statement 140
6-15. Arrays 144
6-16. DIMENSION statements 146
6-17. Loops involving arrays 147
6-18. Branching and terminating loops 149
6-19. Branching out of a subroutine 151
6-20. Debugging 152
Problems 153

PART TWO. ADVANCED PROGRAMMING

Chapter 7. Arithmetic Instructions in Machine Language and SOAP . . 157
7-1. Introduction 157
7-2. Word size, storage capacity, and the basic registers 158
7-3. Functions of the accumulator during arithmetic operations . . . 159
7-4. The form of a 650 instruction word 161
7-5. Optimization 162
7-6. An introduction to SOAP 163
7-7. The location of the decimal point 164
7-8. The addition pattern 165
7-9. Addition, subtraction, and storage codes 167
7-10. Format of a SOAP card; absolute addresses 168
7-11. SOAP addresses 170
7-12. Blank addresses 172
7-13. 8000 addresses in addition and subtraction 173
7-14. The coupled accumulator in addition and subtraction 175
7-15. Overflow during addition and subtraction 176
7-16. Multiplication 177
7-17. Division 179
7-18. Multiplication and division by 10: shifting 180
7-19. Scaling variables and constants 183
7-20. Scaling equations 184

Chapter 8. A Continuation of Machine-language and SOAP Instructions . 188
8-1. The format of data cards 188
8-2. The read band 189
8-3. The read instruction: load cards 190
8-4. Redistributing information on the drum 191
8-5. Initializing card reading from the console 192
8-6. The single-word loader 193
8-7. Other loading routines 195
8-8. Punch bands and the punch instruction 195
8-9. Branching codes 196
8-10. Constants in SOAP programs 200
8-11. Loops in SOAP 200
8-12. More about pseudo-instructions in SOAP 203
8-13. The pseudo-instructions BLR and BLA 204
8-14. The availability table 205
8-15. One symbolic address with two meanings: the pseudo-operation HED 206

8-16. Two symbols with the same absolute address: the pseudo-operation EQU 207
8-17. A symbol with a pre-assigned absolute memory address: the pseudo-operation SYN 208
8-18. The two forms of subroutines for use with SOAP 209
8-19. Subroutine entry and exit 212
8-20. No operation and halt 213
8-21. Some general procedures for SOAP assemblies 214
8-22. Debugging 215
8-23. Omissions 216
Problems 216

Glossary of terms 219

Index 225

PART ONE

Elementary Programming

1

General Introduction

1-1. What Is an Electronic Digital Computer?

An electronic digital computer is a device for manipulating numbers very rapidly. A high-speed machine can add two numbers in a few millionths of a second; even a relatively modest electronic calculator can perform a thousand additions in one second. In a matter of hours these machines can complete computations which would have taken years by hand. Since the solution of many problems in science and technology depends upon numerical calculations, the electronic computer has become an important research tool.

The term "digital" refers to the fact that the operations are carried out by using discrete numbers. An *analog* computer, on the other hand, deals with continuous quantities such as graphs, lengths, voltages, etc. When we use the term "electronic computer" in this book, we will mean a digital machine.

In order to use these devices properly, it is necessary to learn some new techniques. After all some instruction is needed even to learn the use of a slide rule or desk calculator. Actually with only a few hours of instruction and some laboratory experience, most students become sufficiently competent to program original and useful problems; and as one becomes more and more proficient the initial investment of time in learning computer techniques becomes insignificant compared with the time saved in calculation.

Of course, it is necessary to be selective in choosing what one learns. This field is advancing rapidly; new techniques are constantly being

devised and old methods are becoming obsolete. Clearly one should not learn obsolete techniques; on the other hand, it would be easy to spend all of one's time learning the newest methods and, consequently, never actually solve any problems. Since it is assumed that the reader is actively interested in solving numerical problems, the emphasis in this book has been given to techniques that can be put to immediate practical use.

1-2. The Concept of a Program

Let us consider how one goes about using a desk calculator. The operator enters the numbers on the keyboard and presses a button marked $+$, $-$, $\times$, or $\div$, depending upon the operation desired. The machine then mechanically computes the result. A great deal of time is spent in punching keys and writing down results. The most experienced desk machine operators plan their calculations so that they are tabulating the preceding result while the machine is calculating the next. This is an efficient procedure because both man and machine are kept working almost constantly.

A process of this kind could not be made appreciably faster by speeding up the machine alone, because so much time is spent punching the keyboard and writing answers. If the electronic calculator were operated by pushing buttons for each operation, it would be standing idle almost all the time, waiting, while the human operator was writing down the results. In the time necessary to press the button to do one addition, the computer can actually do several thousand additions. These machines are expensive to run and this would be a very uneconomical technique.

A completely different method has had to be devised. Ideally the man should perform his part of the work before the problem reaches the machine. Once the machine begins its operation there should be a minimum of human intervention. To accomplish this the problem must be planned in advance, in detail, and very carefully.

The high-speed computer can perform a limited number of basic operations—some as simple as "add" or "multiply", and others much more sophisticated. The solution to any particular problem must be expressed as a sequence of these basic operations. This sequence is called the *program*, and the process of preparing it is called *program-*

ming. Before the machine can respond to the program it must be expressed in terms of a *code* that the calculator has been constructed to obey.

The course of a calculation is very much like a railroad for which the track (the program) is laid out in advance. The computer begins at the beginning and follows the path laid out by the programmer. Just as in the case of a railroad, from time to time the computer pauses to discharge or receive information. A very important feature of railroads is the ability to choose between tracks depending on which way certain switches are set. In an entirely similar way the computer will perform one or another set of operations depending on certain switches. The computer itself can set the switches that lie ahead or behind it on the track, or the programmer can set the switches in advance. The track can loop back on itself so that the computer runs over the same part of the program many times and leaves this particular group of instructions only when a control switch is set in the opposite direction.

The problem of programming is the problem of setting up the track the computer is to follow in order to solve certain types of problems.

Programming is full of pitfalls, because it is very much like communicating with an accurate robot who has a very small vocabulary and who takes everything you say literally. He cannot exercise any originality so you must foresee all possibilities. If you say "sit", he will sit whether there is a chair there or not. If you want him to sit in a chair, you may have to say "go to chair", "bring chair here", "sit in chair". If you say "sit in chair" and there is no chair, he may blow a fuse.

Although poor programming is not likely to blow a fuse on a computer, it can lead to completely incomprehensible results. The machine inexorably exposes the programmer's weaknesses in logic. It does only what it is told to do—no more and no less. If it does not do what the programmer intended it to do, the programmer has no one to blame but himself.

1-3. The Program Library

Once a program is written it may be used as often as desired with different data. For example, a program written to find the inverse of

a matrix can be used to invert a wide variety of matrices. The intellectual effort necessary to write the program does not have to be duplicated provided the program is preserved.

Some types of calculations occur again and again in many applications. It does not matter that a programmer had agricultural data in mind when he wrote a program to find correlation coefficients. The same program may be used on psychological data, engineering data, or business data. The important thing is that a program has been written to find correlation coefficients. If the program is preserved, together with instructions for its use, the psychologist, engineer, or business analyst can use it without additional programming.

To make programs of such wide applicability easily available, every computing laboratory maintains a program library. A large library of programs represents the fruit of hundreds of hours of programming effort, and it becomes the heritage of every new programmer. Different installations having essentially the same equipment can expand their libraries by exchanging programs; each laboratory then has at its disposal the results of programming done all over the country. Occasionally it is possible to solve large problems by hooking together smaller programs from the library in proper sequence, with very little additional programming.

Without a library, each programmer would have to begin from scratch; with it, he need only program the parts of his problem which are unique.

1-4. Numerical Analysis and Programming

The branch of mathematics known as numerical analysis is devoted to solving numerical problems. This subject is older than electronic calculators by several centuries, and important contributions were made by a great many mathematicians. One function of the numerical analyst is to devise procedures, or recipes, or algorithms (as they are correctly called) for finding solutions to particular problems.

For example, consider methods of calculating the square root of a number. There are several ways of doing it. One that we are taught as children is illustrated by this computation of the square root of 10:

$$
\begin{array}{r}
3.\ 1\ 6 \\
\sqrt{10.0000} \\
9 \\
\hline
61 \mid\ 1\ 00 \\
61 \\
\hline
626 \mid\ 3900 \\
3756 \\
\hline
\end{array}
$$

etc.

With a desk calculator another method is frequently used which involves subtracting successive odd integers. Yet a third method, often used on electronic computers, is called Newton's method and is based on a repetitive or iterative procedure for improving an approximate value systematically. One can also find the square root using logarithms; so you see one has a considerable choice of methods even in such a simple problem.

In a similar way there are many methods for finding the solution of a system of linear algebraic equations such as

$$
\begin{aligned}
8x - 5y + 2z &= 31 \\
x + y + 9z &= 23 \\
2x - y - 8z &= -9
\end{aligned}
$$

To name but a few, we have Cramer's rule, Crout reduction, Gaussian elimination, and relaxation. The numerical analyst studies these methods with particular emphasis on such important questions as the estimation of errors and the rate of convergence.

Programming is the link between the algorithm and the machine. The programmer must specify the explicit sequence of operations to be performed by the machine in executing the algorithm.

Much of the numerical analysis must precede the programming; but it often occurs that problems of analysis arise during the programming or after test trials have been run on the machine. Numerical analysis and programming usually supplement each other until the problem is solved.

Many algorithms were established before the application of modern computers and are not adaptable to automation. More suitable algorithms are continually being devised and their properties are being studied by means of computers.

This book is not intended as a text in numerical analysis. No attempt will be made to derive any of the standard algorithms. Several complete and modern books on the subject are available and are an important part of a programmer's library. Our principal concern will be programming these procedures for practical application.

1-5. The Components of a Computer

Since this is not a book on building computers, we will not be concerned with their internal electronic design. To us, the computer will be a black box that obeys a limited set of commands. It is sufficient to recognize five basic parts of the computer: the arithmetic unit, the memory, the control, the input, and the output.

The calculation is actually carried out in the arithmetic unit. This is the part of the machine in which the numbers are manipulated and transformed into sums, products, quotients, etc.

Since the arithmetic unit deals with only a few numbers at a time, the information needed for the rest of the calculation has to be kept somewhere else; but it must be rapidly accessible when needed. The memory unit serves this purpose: in the course of a computation, information is brought from the memory to other units of the machine and from other units to the memory, as determined by the program. Indeed the program itself is stored in the memory; that is why machines of the type we will discuss are called "stored-program" machines. One reason for doing this is that it is much easier to locate an instruction electronically in the memory than to have to shuffle through a deck of punched cards. Many types of memory are used, including magnetic drums, magnetic cores, magnetic tapes, and electrostatic tubes. The term "storage" is used interchangeably with "memory".

The operations performed by the machine are regulated by the control unit, which sees to it that the instructions of the program are carried out correctly and in proper sequence.

Finally, in order to get information into and out of the machine, we use the "input" and "output" units. The input unit accepts information in the form of punched cards, punched tape, or magnetic tape. (Usually the cards or tapes themselves are called the input.) Results are produced by the output unit, again in the form of cards or tape.

In some cases the output unit can produce a printed sheet of answers directly; in other cases, however, the cards or tapes produced by the computer must be fed into an auxiliary machine in order to obtain a printed record.

The actual course of a computation may go something like this: the program and the data are prepared in advance on cards, tapes, or any other appropriate form of input. These are fed into the input unit; the information on them is transformed to electrical impulses and carried to the memory, where the information is stored. (Usually before any calculating is done the instructions of the program, and perhaps even all the data, are stored in the memory.) Then the calculation begins. The first instruction is taken from the memory to the control unit, which supervises its execution. When the first instruction is completed, the second instruction is executed; and so on. During the calculation numbers come from the memory to the arithmetic unit and intermediate results go back to the memory. From time to time it may be necessary to call on additional data or program from the input units. Eventually, when final results are computed, the output unit produces the answers on cards, tapes, or a printed sheet.

1-6. Some Typical Computer Problems

Such a variety of problems are suitable for high-speed computation that we can sample only a few to illustrate the versatility of this equipment.

Consider first the evaluation of a simple formula such as

$$z = \log (3x + \sqrt{\cos 2y})$$

for many pairs of x and y. For example, x and y might be experimental measurements for which we wish to find the corresponding values of z. This equation can be programmed easily, and once the program is written it may be used to compute z for as many pairs of x and y as desired.

This problem is not a difficult one for a desk machine either; but suppose there are a thousand pairs of x and y for which we want to compute z's. For a human operator at a desk machine this becomes a long, tedious computation. With an electronic computer, on the other hand, it becomes rather trivial. It can be accomplished with speed, accuracy, and a minimum of drudgery.

In some statistical operations the calculations themselves may be simple—multiplying, summing, and taking square roots. Nevertheless, to carry out these operations on thousands of numbers without error, in a reasonable time, taxes the ability of the most skilled desk calculator operator. On an electronic machine large statistical computations can be carried out as a routine matter using standard programs. The number and size of the data in a particular computation are usually limited only by the storage capacity of the machine.

There are applications in which only a few hundred combinations of data may be expected. In that case it might be advisable to construct a table listing the results of the calculation for all the likely combinations. For example, in the previous equation we may expect x to vary from 0.001 to 0.100 and y from 10° to 20°. It may therefore be adequate to construct a double-entry table, with x running from 0.001 to 0.100 in intervals of 0.001, and y from 10° to 20° in intervals of 1°. A further advantage of tables is that once they are calculated they can be used in the laboratory or in field work without going back to the computer again and again.

Sometimes information needed in the calculation is known only in tabular form. The machine can be used to find appropriate interpolation formulas. If the tables represent experimental data and are subject to errors of observation, interpolation curves can be found by using criteria of best fit.

Series expansions are sometimes available to represent special functions. In that case the machine can be instructed to terminate the series after a fixed number of terms, or when additional terms can be dropped without appreciable error.

As the number of sets of data to be treated becomes larger or as the formula becomes more and more complicated, the electronic machine gains increasing advantage over the human operator of a desk calculator. Electronic machines make it possible to solve large linear systems or simultaneous differential equations that have no known analytic solutions.

By removing drudgery from computation the electronic calculator permits the scientist or engineer to concentrate on the interesting aspects of the problem. Although a beginner usually starts by programming enlarged versions of desk machine calculations, he soon realizes that computers open completely new possibilities.

Consider, for example, a design problem which depends upon certain trial parameters or trial functions. Two computations with different trial values can be compared to determine which parameters yield a more effective solution. The machine can go further: it might be possible to program it to select the next trial function. The machine would then automatically perform the computation again, evaluate the results, and improve the function still further. This process can continue until some criterion of acceptability is satisfied.

This ability of computers to attack problems of a completely new dimension is certain to have a continuing influence on the growth of science and engineering.

1-7. When Is a Problem Too Large or Too Small for an Electronic Computer?

Speed and memory capacity usually limit the magnitude of problems that can be solved on a particular machine. Of course, a skilled programmer can push a calculator to the extremes of its capability, but there always remain some problems that are simply not practical on a particular piece of equipment. The alternative is to get a bigger machine and, if one does not exist, to build one. Constant pressure for larger and faster machines is being exerted by industrial and government laboratories to meet the demands of aircraft and rocket research. Fortunately, when these machines are produced, all fields of research profit from their use.

At the other extreme, no two people would agree on defining a problem too small for an electronic machine. This becomes a serious matter when schedules are tight and priorities must be assigned. An expert programmer with access to a machine can use it as easily as a slide rule, and he would rather write a program than evaluate some $f(x)$ by hand for more than three values of x. But is he using a sledge hammer to kill a fly?

There are at least three criteria that can be used to judge the case: time, expense, and the assurance of accuracy. Naturally, we always want accurate answers, but the relative importance of time and expense will vary with the circumstances. Let us briefly consider these criteria in turn.

Suppose time is our principal concern. In estimating time we must

consider all steps from the statement of the problem to the final result. When a problem is run on an electronic machine each of the following steps consumes time: programming, preparing input cards or tapes, waiting for an assignment of machine time, testing, perhaps waiting again for machine time, computing, and, finally, tabulating results. In a small problem, computing time may represent only a small fraction of the total time. If the programmer is inexperienced or if the electronic machine is not quickly accessible, the desk calculator may actually be more efficient.

The calculation of expense is, of course, interwoven with estimates of time. The electronic machine actually gives you your money's worth with respect to the number of computations per dollar. However, the programmer's time is not inexpensive. Furthermore, in a problem involving a few operations on a lot of data, the preparation of the input cards or tape may account for a considerable fraction of the expense.

Finally, a word about accuracy. The accuracy of a desk calculator operator is very variable. It may depend upon what he had for lunch or the status of a love affair. The more intricate the calculation, the more likely errors are to occur. Tables can be misread, numbers transposed, decimals miscounted, etc.

Most electronic machines are very reliable. They perform millions of operations without error and they perform difficult computations as accurately as simple ones. Most machines are tested daily as a matter of course. Some contain checking circuits and stop or repeat an operation if an error is detected; but this does not mean that machines are infallible. Estimates of reliability may be made on the basis of past experience with a particular machine; keep in mind, however, that only *detected* errors are recorded.

As you see, the problem of judging when a calculation is "too small" for an electronic machine depends on many factors. There is even an aspect we have thus far neglected to mention: the vanity of the programmer who is insulted if his problem is not worthy of the world's largest machine!

One solution that is becoming popular at those installations that can afford it is to provide a hierarchy of machines and to schedule a particular problem on the smallest machine that can accommodate it. The efficiency of this arrangement would be greatly improved if every machine in the hierarchy responded to the same language.

2

The Approach to Simple Problems

2-1. Number Systems[1]

Some computing machines carry out their operations in terms of the binary, octal, or sexadecimal number systems rather than the familiar decimal system. This is done because of some practical engineering considerations and not, as it may seem, to make programming an even more mysterious art. Actually, other number systems are not really difficult to understand once we know what the decimal system is all about.

First of all, we must realize that we are discussing the representation of a number, rather than the concept of a number. For example, the concept of four has many familiar representations:

4, four, IV

We choose different representations to suit different uses, but the concept of "four" is unaffected by its representation.

One would scarcely consider doing algebra with roman numerals. In a similar way, if other representations are more suitable for electronic computers, there is no reason why we should not use them.

[1] This section may be omitted if the programmer is concerned solely with a decimal machine.

As the name "decimal" implies, our common representation of numbers is based on ten, and requires ten symbols:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

In counting, we use these ten symbols in sequence. We count up to nine; then we "carry one" to the left, and write the next number as a composite symbol:

10

For the next nine numbers, only the right digit changes:

11, 12, 13, etc.

The 1 at the left is an abbreviated representation, meaning 1×10. That is, 12 means $(1 \times 10) + 2$; 18 means $(1 \times 10) + 8$, etc. Carrying this further, 45 means $(4 \times 10) + 5$ and 93 means $(9 \times 10) + 3$.

In the same spirit

$$6724 \text{ means } (6 \times 1000) + (7 \times 100) + (2 \times 10) + 4$$
$$\text{or } (6 \times 10^3) + (7 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

The last representation is the most revealing, because it shows us the fundamental idea behind our representation of numbers. The ten symbols 0, 1, . . . , 9 can be used to represent more than ten numbers by writing them down in a particular order. The representation 6724 when written out in detail is actually a sum of products. Each product consists of two factors: a digit (6, 7, 2, or 4) and a power of ten, determined by the position of the digit. The digit farthest to the right is multiplied by 10^0 or 1; the next digit to the left is multiplied by 10^1, etc. Ten is called the "base" of this system.

So far we have considered integers, but we can remove this restriction by introducing the decimal point. Digits to the right of the decimal point are to be multiplied by *negative* powers of ten. Thus

$$36.512 \text{ means } (3 \times 10^1) + (6 \times 10^0)$$
$$+ (5 \times 10^{-1}) + (1 \times 10^{-2}) + (2 \times 10^{-3})$$

Summarizing, we see that the decimal system is based on powers of ten and requires ten symbols. The location of these symbols relative to the decimal point indicates the appropriate power of ten to be used; positions to the left involve positive powers (beginning with zero) and those to the right, negative.

There is nothing to prevent us now from setting up a number system

using a different base. The binary system uses the base 2; the octal system, the base 8; etc.

The decimal system used ten symbols, but the binary system uses only two: 0 and 1. (A *b*inary dig*it* is frequently called a "bit".) The decimal system is based on powers of 10; the binary system uses powers of 2. The "binary point", just like the decimal point, separates the positive from the negative powers. Thus the binary number

$$101.11 \text{ means } (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (1 \times 2^{-2})$$

The equivalent decimal representation is 5.75.

The first eight integers in both binary and decimal representations are

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

As you can easily verify, the rules for counting (or, equivalently, addition) are very simple:

$$1 + 0 = 1$$

$$1 + 1 = 0, \text{ carry } 1$$

Because binary arithmetic requires only two symbols and has such simple rules, it can be easily mechanized. For example, the symbols can be treated as switches which are either open (0) or closed (1); and they can be made visible by means of lights that are either off (0) or on (1). There are disadvantages, however. In adding two binary numbers there are frequent "carries". Furthermore, it takes almost four times as many binary digits as decimal to represent a number.

If we wish, we can devise number systems to bases greater than ten. Indeed, we use several every day without being disturbed. We calculate in terms of degrees, inches, pounds, seconds, etc., none of which are based on the decimal system. We used two symbols (0 and 1) in

the binary system and ten symbols in the decimal system. If we were to stick to the requirement that a system with base sixty needs sixty symbols we would have to invent fifty new symbols; and this would be awkward, to say the least. Instead, we still use the ten decimal symbols, but we make our meaning clear by separating them with new signs:

$4^\circ\ 52'\ 8''$ means $(4 \times 60^2) + (52 \times 60^1) + (8 \times 60^0)$ seconds of arc

On the other hand, to use a base of sixteen (the sexadecimal or hexadecimal system), we need only six new symbols, and this is not difficult to manage. We could use letters (as they do for the Illiac at the University of Illinois):

Decimal	Sexadecimal
10	K
11	S
12	N
13	J
14	F
15	L

Then the sexadecimal number

2K3 would mean $(2 \times 16^2) + (10 \times 16^1) + (3 \times 16^0)$

or 675 in decimal notation.

Although a binary computer can be programmed to print or punch the final results in decimal form, the situation frequently arises in checking a program when one needs to be able to read the binary representation indicated by the console lights. Instead of reading the number digit by digit as, for example,

110011100111

it is more convenient to group the digits in sets of three. Each group can be read as a number between zero (000) and seven (111). That is

	110	011	100	111
becomes	6	3	4	7

The number 6347 is then the octal equivalent of the binary number.

To see that this transformation is legitimate consider the binary number 110 001

which means [illegible]

$$[illegible]$$

This is equivalent to

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

$$[illegible]$$

Each term in square brackets can therefore be regarded as an octal digit in binary form.

This mixture of number systems is called binary-octal. If the binary digits are divided in groups of four instead of three, we can form a binary-sexadecimal system. Octal and sexadecimal representations are more concise than binary, and with a little practice one can transcribe from binary to octal or octal to binary very quickly.

2-2. Word Size

Information is stored in the memory in the form of individual digits, but it would be very impractical if the programmer had to deal with each digit separately. The memory is therefore subdivided into groups of digits called words, which are usually treated as units. The number of digits in a word, called the word size, is one important characteristic of a machine. When we speak of a calculator having a word size of 10 decimal digits plus sign, we mean that information is stored in the memory in 10-digit groups (plus sign) and that most of the operations of the machine involve 10-digit numbers.

It is possible, though much more difficult, to deal with 20-digit numbers on a 10-digit machine. This technique, called "double precision", will be dealt with later on. (See Prob. 6, Chap. 8.) It is also possible to "pack" two 5-digit numbers into a 10-digit word. Here again the programming is more difficult, but we will return to it in Probs. 1 and 2, Chap. 8.

A word size of 40 binary digits is roughly equivalent to 12 decimal digits.

2-3. Data Words: Fixed- and Floating-point Representations

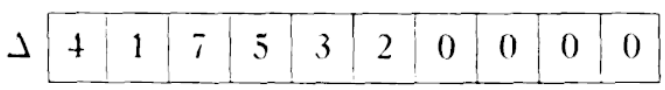
In arithmetic operations, the location of the decimal point is of great importance. To add two numbers, their decimal points must first be aligned; and to locate the decimal point in a product, the number of decimals in the multiplier and multiplicand must be known. (The same remarks hold with respect to the binary point in binary arithmetic.) In the previous section, however, we have not mentioned any provision for storing the decimal point. The fact of the matter is that there *is* no provision for storing the decimal point as a separate piece of information. There are, however, two methods of dealing with this problem.

In one procedure, called *fixed point*, the programmer has the responsibility of keeping track of the decimal point at every stage of the computation. In the other, called *floating point*, the machine itself is used to determine the decimal point. We will examine these alternatives in terms of a machine with a word size of 10 decimal digits.

In fixed-point calculations, the programmer is free to assign the location of the decimal point within each word. It is up to him to align numbers before adding, and he must count the decimals carefully in products and quotients. In all but the simplest problems, fixed-point calculations require skillful programming. (See Chap. 7.)

To illustrate only one of the limitations of fixed-point calculations, suppose variable x is always less than 1, and we wish to record its value to six significant digits. We can assume the decimal point to lie immediately to the left of the first digit of the data word.

The datum $x = 0.417532$ offers no difficulty. We merely add zeros to bring it up to word size:



On the other hand, the datum $x = 0.0000845713$ can be recorded only to five significant digits because the decimal point is immovable and we are therefore restricted to ten decimals:

(3)

As a consequence, the acceptable range of any variable is limited. In this case we can preserve six significant digits only if variable x lies in the range from

Δ	9	9	9	9	9	9	0	0	0	0

to

Δ	0	0	0	0	1	0	0	0	0	0

In many scientific problems, the range of some variables may be very great. In that case, fixed-point programming becomes complicated. We will return to these difficulties when we discuss "scaling" in Secs. 7-19 and 7-20. Meanwhile, we shall only use fixed-point arithmetic in simple applications.

Floating-point notation comes from the equivalence

$$.007435 = 7.435 \times 10^{-3}$$

The second representation employs a mantissa, containing the significant digits, multiplied by 10 raised to a power. Let us consider how the information contained in this notation can be confined to the limits of a 10-digit word with one sign.

To begin, we assign the first eight digits of the word, counting from the left, to the mantissa, and the two digits on the right to the exponent:

Mantissa

Exponent

Then we require that the first digit of the mantissa must be a significant digit and not zero. Further, we will select the power of 10 so that the decimal point in the mantissa lies to the right of the first significant digit. That is, we will accept 7.435×10^{-3}, but not the mathematically equivalent notation 74.35×10^{-4}. If there are any fewer than eight significant digits in the mantissa, zeros are added:

7	4	3	5	0	0	0	0

In the standard mathematical notation, both mantissa and exponent may be negative·

$$-.000431 = -4.31 \times 10^{-4}$$

Since there is provision for only one sign in the word, we will let it be the sign of the mantissa. Negative exponents are eliminated in a simple way. We merely take the power of 10 in the usual representation and add 50 to it. We can therefore represent the true powers from -49 to $+49$ by using positive numbers from 1 to 99. A few typical transformations are given below:

$575.82 = +$	5	7	5	8	2	0	0	0	5	2
$-.0037145 = -$	3	7	1	4	5	0	0	0	4	7
$-14116.3 = -$	1	4	1	1	6	3	0	0	5	4
$1.0 = +$	1	0	0	0	0	0	0	0	5	0
$.00078 = +$	7	8	0	0	0	0	0	0	4	6

Zero is represented with zero exponent as follows:

0	0	0	0	0	0	0	0	0	0

When performing the operations of floating-point arithmetic, the machine examines the exponents to align numbers before adding and to determine the exponents of products and quotients.

It should be noted that there is no uniformity in the use of floating-point notation. In some techniques the exponent is placed at the left instead of the right; in some the decimal point is assumed to *precede* the first significant digit rather than follow it. We shall try to make our notation clear in every case.

One clear advantage of floating-point operations over fixed point is the range of variable permitted—from 10^{-49} to 10^{+49} in the case we have just discussed.

But there are disadvantages too. Not all machines have instructions which perform floating-point operations at electronic speeds. Programs can be written to carry out floating-point arithmetic on fixed-point machines but these programs are invariably slow. They also occupy space in the memory. Therefore, when speed and capacity are

critical, floating-point operations on fixed-point machines are not recommended.

There is also a loss in the number of significant digits that can be stored in a single word. In a fixed-point word, all ten locations can carry significant digits, but in a floating-point word two are used to locate the decimal. In some problems eight significant digits are insufficient and therefore a single floating-point word cannot be used.

A further serious drawback is the difficulty of error analysis in floating-point calculations. This is an aspect of numerical analysis where much still remains to be done and further discussion is beyond the scope of this book.

We shall take advantage of the simplicity offered by floating-point operations and use them freely in learning the rudiments of programming. Nevertheless, we must keep the disadvantages in mind and return to them when we have mastered the fundamentals.

2-4. Addressable Memory

When performing a calculation it is necessary to refer to the precise memory location of a particular word. Therefore, an important part of every machine is an addressable memory in which the location of each word is assigned an address. For example, the Illiac has an electrostatic memory with capacity for one thousand twenty-four 40-binary-digit words, each having a decimal address between 0000 and 1023. The IBM Type 650 has a magnetic drum that can store two thousand 10-decimal-digit words, each having an address between 0000 and 1999.

When information is "sent to" a particular storage location in the memory the previous word is automatically erased before the new word is recorded. It is important to remember that the new information completely replaces whatever was there before; the previous information disappears without a trace.

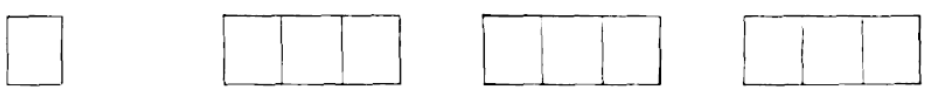
Conversely, the only time information is erased in the memory is when new information replaces it. This means that, even though we speak of a word being "brought from memory location 1324 to the arithmetic unit", we actually mean the number in location 1324 is copied in the arithmetic unit, but it remains in location 1324 of the

memory as well. This permits us to retain the same information as long as we desire without modifying it.

2-5. Instruction Words

In a stored-program machine instructions of the program are stored in the memory along with the data. In some machines each instruction is one word long; in others two or more instructions can be put in a single word.

Every digit of an instruction has its particular function. Some of the digits indicate the operation to be performed; others indicate the memory address from which numbers are to be taken or to which numbers are to be sent. For example, a 10-digit instruction might be divided as follows:



Operation Code	Address A	Address B	Address C

Suppose the operation code "3" means "take the number in memory location A; multiply it by the number in location B and store the product in location C". When executed by the machine the instruction

3 724 516 053

would multiply the number in memory location 724 by the number in location 516 and store the product in location 053.

An instruction of this type is called a "three-address instruction" for obvious reasons. Some instruction words have only a single address; others contain even more than three.

It ought to be pointed out that it is impossible to distinguish the instruction word 3 724 516 053 from the data word 3724516053. Both contain ten digits and when they are stored in the memory they are identical. The only difference lies in the way they are used in a particular problem.

2-6. Machine Language

You will recall that the control unit is responsible for executing the instructions of a program. An instruction word is brought from the

memory to the control unit. There the digits representing the operation code are examined and the appropriate instruction is executed.

The codes that are recognized by the electronic control unit form the "machine language". The number of codes built into a machine is limited, usually by economic considerations. Every new code requires additional electronic circuitry to carry out the operation associated with it and this is expensive. However, enough basic operations are provided in the machine language to be able to achieve very elaborate operations by programming. For example, there may not be a *single* machine code to find the square root of a number, but a square root procedure can be programmed in perhaps a dozen machine-code steps.

Although ultimately every computation is carried out in machine language, there are other languages that are easier to use in programming. These codes are not normally recognized by the control unit but the machine can be used to translate from the language of the program to the language of the machine.

It is true that an expert programmer can use the machine language to write very efficient programs, but alternative languages offer many advantages. One of the most important is that the likelihood of errors in programming is very greatly reduced and consequently programming time is decreased.

2-7. Interpretive Programs

One of the ways to simplify programming is to use an interpretive system. In this case a pseudo-code is created which has outwardly little to do with the machine language. Each instruction of the pseudo-code represents a set of machine-language instructions so that one can perform rather complicated operations with a single pseudo-code instruction. By using an interpretive program you effectively substitute the language of a machine which does not exist for the language of the machine you are actually using.

When using an interpretive system it is necessary to store an "interpreter" on the drum, in addition to the pseudo-code program and the data. The interpreter is a program in machine language which analyzes each pseudo-code instruction in turn and executes the appropriate machine-language steps.

Interpretive systems have several advantages. Pseudo-codes are often easier to learn than the machine language; only a single pseudo-code instruction is needed to generate frequently used functions such as square root, sin x, exp x, and log x. We have already mentioned that interpretive systems make programming easier and cut down on mistakes.

Interpretive systems, however, do have some disadvantages. They are usually slow to run because each instruction must be analyzed before being executed. Furthermore, the interpreter itself occupies considerable storage, restricting the storage available for the pseudo-code program and the data. Nevertheless, interpretive systems are very useful, particularly in small problems in which time and storage are not critical.

2-8. The Bell Telephone Laboratories Code

For illustrative purposes in the first part of this book, we will use a code developed at Bell Telephone Laboratories by V. M. Wolontis and associates. It is a simple, yet versatile, language with which we can learn some basic programming principles. This is not the machine language of any computer; it was actually developed as an interpretive system for the IBM Type 650, and if a 650 is accessible, examples coded in the Bell Laboratories language can be tested and run on it. Detailed instructions for using the Bell Laboratories language on the 650 are given in "IBM Technical Newsletter No. 11".

The word size of this code is 10 decimal digits. Numerical applications are carried out in floating-decimal arithmetic with an eight-digit mantissa, and a two-digit exponent on the right. Exponents are formed by adding 50 to the power of ten (Sec. 2-3). The decimal point follows the first digit of the mantissa. The integer 1 is therefore represented as 1000 0000 50. The number 0 is represented by ten zeros. The next smallest number is 1×10^{-49}, represented by 1000 0000 01. The largest is $9.999\ 9999 \times 10^{49}$, represented by 9999 9999 99.

An instruction word consists of 10 decimal digits, subdivided as follows:

x xxx xxx xxx

There are two types of instructions: in one the first digit is zero; in the other it is not. When the first digit is not zero, the subdivisions are called

x	xxx	xxx	xxx
Op 1	A	B	C

Otherwise, they are labeled

0	xxx	xxx	xxx
	Op 2	B	C

One thousand memory locations, with addresses 000 to 999, are available to the programmer. Location 000 has some special functions and its use is restricted; otherwise, any location may be used to store data or instructions.

The location of the first word of a program can be chosen arbitrarily by the programmer. After it is executed, the machine automatically goes on to execute the instruction in the next memory location, and so on. That is, if the program begins in location 445, the second instruction would normally be in 446, the third in 447, etc. This is called *sequential* programming. It is frequently necessary, however, to interrupt the sequence, and there are instructions available to do that.

2-9. Uses of Storage

Memory locations may be used to store any kind of information. Usually we use two broad classifications: program and data. There are, however, several kinds of data that we can recognize. For example, we must store constants that do not vary from one computation to another and variables that take on different numerical values for each calculation. In addition, we must provide space in the memory to store intermediate results and final answers.

Consider the calculation

$$z = \log (\sqrt{3 + y} + \cos x)$$

In this case, 3 is a constant, unchanged throughout the series of computations. Each calculation will be carried out for a different pair of variables, x and y. During the computation, as we shall see, we will have to store the intermediate result $\sqrt{3 + y}$ during the calculation of $\cos x$.

Constants are usually fed into the machine at the very beginning of a series of calculations. In the preceding example, the number 3 might be stored in memory location 601. We must be careful not to

use address 601 for anything else, because if we accidentally store something else in 601, the constant will be erased in the process.

Suppose we decide to store the first pair of variables x and y in 602 and 603. After the computation is completed for the first pair, we do not need this pair any more. We could, therefore, read the second pair into the machine after the first computation is finished, and store them in the same locations, 602 and 603.

Some of the intermediate results may be needed for only part of the computation. When we are through with one, we can store another in the same location. This is particularly advisable if the problem is a large one and the memory will be crowded.

We may summarize by noting that some storage locations will contain the same numbers throughout a series of computations. Others will always be assigned to the same variable or the same function, although the numerical values may change. Still others will serve as temporary storage for several different functions.

It may be obvious but it is worth repeating: be sure not to destroy a result before you are through with it!

2-10. Arithmetic Operations

The arithmetic codes of the Bell interpretive system are easy to remember. The first digit of an arithmetic instruction describes the operation:

First digit	Operation
1	addition
2	subtraction
3	multiplication
4	division

Each of these operations involves the combination of two numbers to produce a third. For example, in multiplication, the multiplier and multiplicand produce the product. Therefore we must specify, in the instruction word, the memory address of the multiplier and multiplicand as well as the address to which the product will be sent.

For example, suppose that we begin with the number 4 in 301, 8 in 302, and 7 in 303. (We are not concerned, for the moment, with how these numbers got there.) Recalling the floating-point notation of

this system (Sec. 2-8), we note that the contents of locations 301, 302, and 303 will actually be

301: +4000 0000 50
302: +8000 0000 50
303: +7000 0000 50

If we were to execute the instruction

2 303 301 624

the number in 301 would be subtracted from the number in 303 and the difference would be stored in 624. That is, after the order is executed the contents of the memory would be

301: +4000 0000 50
302: +8000 0000 50
303: +7000 0000 50
624: +3000 0000 50

Whatever had been in 624 has been replaced by the new result.

Following the execution of the instruction

1 302 303 418

the contents of 301, 302, 303, and 624 are unchanged but memory location 418 now contains

418: +1500 0000 51

After the following instructions are executed in sequence:

2 302 301 401
1 301 303 402
4 302 301 403
3 303 302 404

locations 401 through 404 will contain

401: +4000 0000 50
402: +1100 0000 51
403: +2000 0000 50
404: +5600 0000 51

An instruction of the form

1 302 302 405

in which one address is repeated, is entirely legitimate. As a result of this operation twice the number in 302 will be stored in 405.

The instruction

2 301 301 407

is a simple way to generate a zero, since it subtracts the number in 301 from itself.

What will happen when the instruction

3 301 303 301

is executed? First the product of the numbers in 301 and 303 will be formed; then this product will replace the number in 301. At the end of this operation, storage locations 301 to 303 will contain

301: +2800 0000 51
302: +8000 0000 50
303: +7000 0000 50

After the instruction

4 302 302 302

is executed, location 302 will contain

302: +1000 0000 50

If the instructions

4 301 302 406
2 303 301 406

are performed, one right after the other, the result of the first operation is lost, because the second result is stored in the same location, 406.

Actually these arithmetic instructions describe operations to be performed on the *contents* of the specified storage locations. For example, the instruction

1 A B C

means: "Replace the contents of storage location C by the sum of the contents of storage locations A and B". Denoting by $\overline{A}$ the *contents* of memory location A, we can describe code 1 by the symbolic relation

$$\overline{C} = \overline{A} + \overline{B}$$

This is not an equality in the ordinary sense. It describes a dynamic operation in which the contents of C are replaced by the sum.

We can describe all the arithmetic operations in this way. If a general operation is in the form

Op 1 A B C

the result of the operation will be

Op 1	Result
1	$\overline{C} = \overline{A} + \overline{B}$
2	$\overline{C} = \overline{A} - \overline{B}$
3	$\overline{C} = \overline{A} \times \overline{B}$
4	$\overline{C} = \overline{A} \div \overline{B}$

2-11. The Special Role of Address 000

At the end of an arithmetic operation, the result is not only sent to address C but also to address 000. Therefore, if we want to use the result of the previous operation as either the A or B factor in a new instruction, we can do so by using address 000 for either A or B.

Suppose

x is in 425
y is in 426
z is in 427

The result of the successive operations

1 425 426 500
1 000 427 501

is that

500 contains $x + y$
501 contains $x + y + z$
000 contains $x + y + z$

If we are interested in preserving only $x + y + z$ and not $x + y$ itself, we could have given the instructions

1 425 426 000
1 000 427 501

2-12. Simple Arithmetic Problems

Let us try our hand at programming some simple problems using the arithmetic codes. We will arbitrarily begin our programs in location 101; store variables in the 200's; constants in the 300's; use the 400's for intermediate results; and put final results in the 500's.

Example 1: Compute

$$f = \frac{x + 8y}{y - 2z}$$

First we plan our storage. We will store

x in 201
y in 202
z in 203
8 in 301
2 in 302

and at the end of the computation we will store

f in 501

We will still neglect, for the time being, the method for storing the variables and constants originally.

We can do only one arithmetic operation at a time. Therefore we must describe the calculation of f in individual arithmetic operations:

$$\begin{aligned} p &= 8 \cdot y \\ s &= x + p \\ t &= 2 \cdot z \\ u &= y - t \\ f &= s \div u \end{aligned}$$

This is our *program*, but we must put it into the *code* before it can be understood by the machine. Each line of the program becomes an instruction. Beginning in location 101, we will store the instructions as follows:

101:	3	301	202	000	$p = 8 \cdot y$
102:	1	201	000	400	$s = x + p$
103:	3	302	203	000	$t = 2 \cdot z$
104:	2	202	000	000	$u = y - t$
105:	4	400	000	501	$f = s \div u$

Since p, the result of the first calculation, is immediately used in the second, we do not bother to assign it a storage location of its own. Only s has to be stored for future use. Similarly, t is used immediately, and so is u. The final result f is stored in 501.

Example 2: Find f, given

$$\frac{1}{f} = \frac{1}{x} + \frac{1}{y} + \frac{1}{z}$$

Again we begin by planning the storage. Put

x in 201
y in 202
z in 203
1 in 301

and at the end of the computation, store

f in 501

The program is as follows:

$$\begin{aligned} p &= 1/x \\ q &= 1/y \\ s &= 1/z \\ t &= s + q \\ u &= t + p \\ f &= 1/u \end{aligned}$$

In code this becomes

101:	4	301	201	401	$p = 1/x$
102:	4	301	202	402	$q = 1/y$
103:	4	301	203	000	$s = 1/z$
104:	1	000	402	000	$t = s + q$
105:	1	000	401	000	$u = t + p$
106:	4	301	000	501	$f = 1/u$

Example 3: Evaluate the polynomial

$$f = x^4 + 7x^3 + 3x^2 + 5x + 12$$

We assign storage locations as follows:

x in 201
7 in 301
3 in 302
5 in 303
12 in 304

The final result will be

f in 501

We could, of course, write a program as follows:

$$\begin{aligned} p &= 5 \cdot x \\ q &= x \cdot x \end{aligned}$$

$$
\begin{aligned}
s &= 3 \cdot q \\
t &= x \cdot q \\
u &= 7 \cdot t \\
v &= x \cdot t \\
w &= v + u \\
y &= w + s \\
z &= p + y \\
f &= z + 12
\end{aligned}
$$

This program would take 10 steps and would require at least five storage locations for the intermediate results p, q, s, t, and u.

A more efficient way to evaluate a polynomial can be found by noting that we can also write it as

$$f = (((x + 7)x + 3)x + 5)x + 12$$

Then the program can be written as

$$
\begin{aligned}
p &= x + 7 \\
q &= p \cdot x \\
s &= q + 3 \\
t &= s \cdot x \\
u &= t + 5 \\
v &= u \cdot x \\
f &= v + 12
\end{aligned}
$$

This takes only seven steps and requires no temporary storages. Coded in the Bell language it becomes

```
101:  1 201 301 000
102:  3 000 201 000
103:  1 000 302 000
104:  3 000 201 000
105:  1 000 303 000
106:  3 000 201 000
107:  1 000 304 501
```

This method of rewriting a polynomial for programming is worth remembering. It also illustrates the point that the most obvious program is not always the most efficient. With a little thought it is often possible to save storage locations or instructions and these are always important considerations.

It is very important to note that the essential step is the one from

the equation to the program: $p = x + 7$, $q = p \cdot x$, etc., rather than from the program to the specific Bell code. The program contains the details of the solution to the problem; the code is merely the adaptation of the program to a particular machine.

Although the machine at hand does influence some of the programming details, programming techniques are the same for all stored-program machines up to a point. Therefore, although we are using the Bell code to make our examples specific, the methods we are developing are quite general.

2-13. The Punched Card

The programs we have written thus far are incomplete because we have made no provision for getting information into and out of the machine. Before we can discuss instructions of this kind, let us consider one frequently used device for recording computer information—the IBM punched card.

All cards used by IBM equipment are identical in size and thickness. Information is recorded in the form of rectangular holes punched on the card (Fig. 1). Printing on the card and colored stripes mean nothing to the machines; all they can recognize is holes. Colors and printing, however, are useful to the human operator as a quick means of identifying groups of cards.

Every card contains 80 columns and it takes one column to represent one decimal digit. Therefore, 80 decimal digits can be recorded on one card. It takes only one hole in a column to represent a decimal digit; for example, the lowest position in any column represents the number 9; the next lowest, the number 8; and so forth. The bottom of the card is therefore called the "9 edge".

Working our way up the card, we pass the 9 row, the 8 row, the 7 row, etc., until we come to the 0 row. However, we have not yet come to the top of the card. Two more rows remain, called the X, 11, or minus row and the Y, 12, or plus row. For obvious reasons, the top of the card is called the "12 edge". The punched card representation of the decimal digits and the + and − signs is indicated in Fig. 2.

Alphabetic information is recorded by using two punches in the same column: the upper punch (sometimes called the "zone") is always

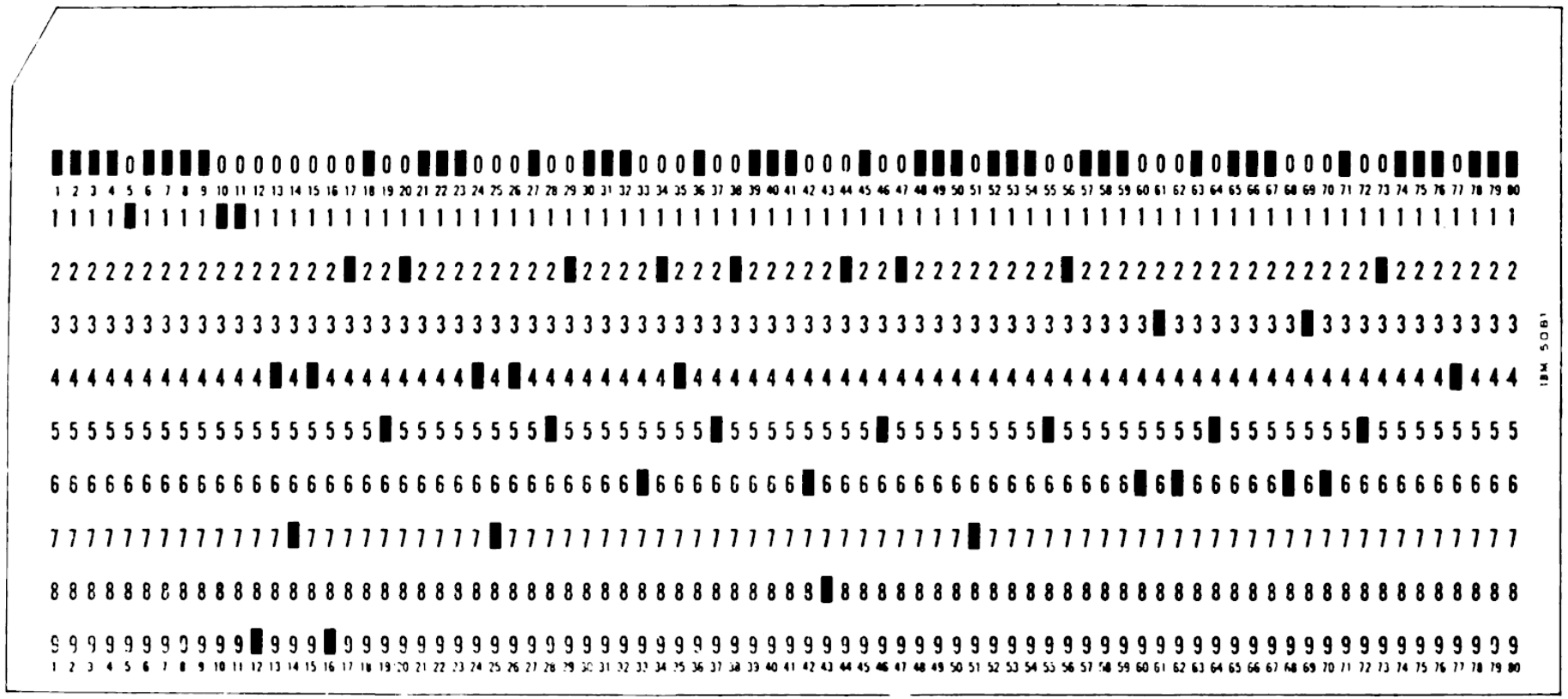


FIG. 1. (*Reproduced by permission of International Business Machines Corporation.*)

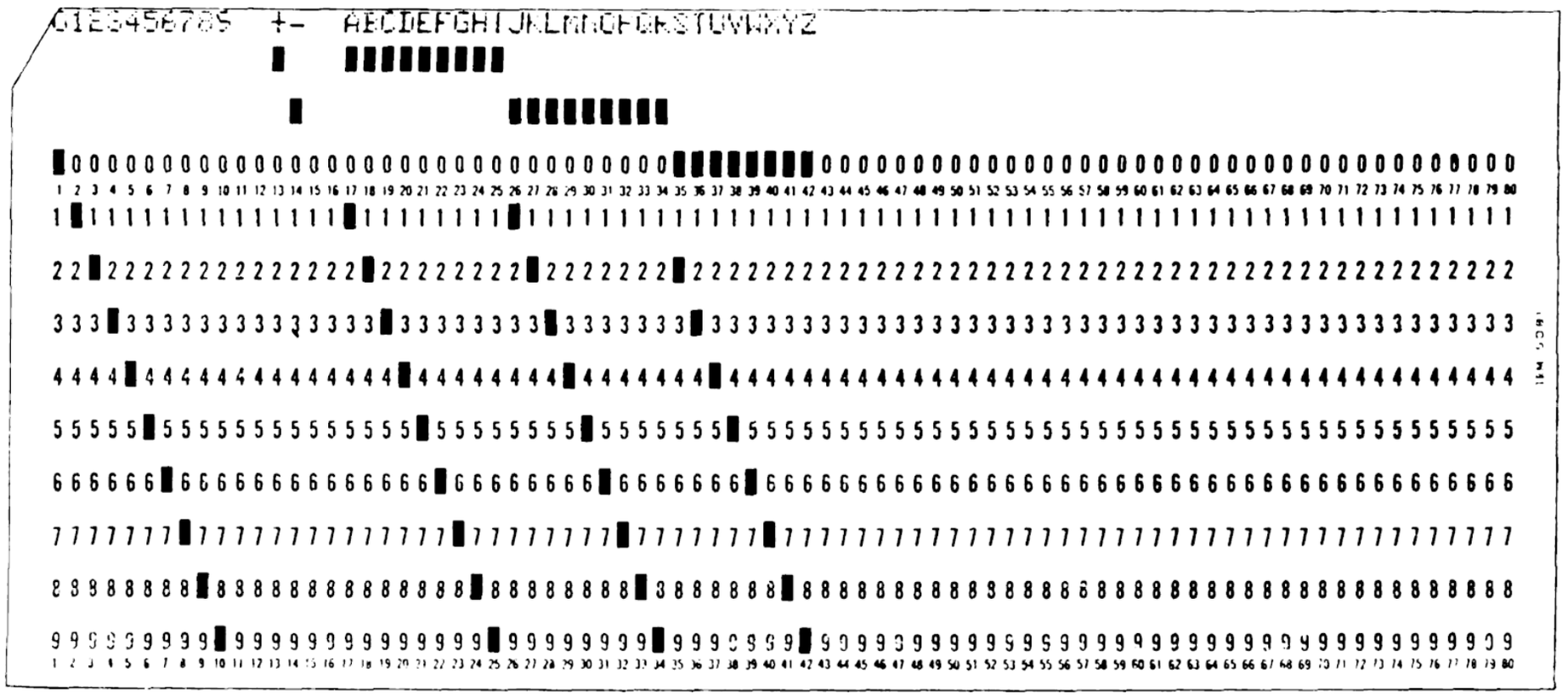


FIG. 2. (*Reproduced by permission of International Business Machines Corporation.*)

a 12, 11, or 0 and the lower varies from 1 to 9. Table 1 gives the combinations used and they are illustrated in Fig. 2.

Table 1

	1	2	3	4	5	6	7	8	9
12	A	B	C	D	E	F	G	H	I
11	J	K	L	M	N	O	P	Q	R
0		S	T	U	V	W	X	Y	Z

For example, the letter M is represented by an 11 and a 4 punch in the same column. The combination 0 and 1 is not used.

The format of a card in the Bell scheme is illustrated in Fig. 3. Eleven columns are used for each word: one for the sign and ten for the digits. From one to six words can be punched on one card; the first word is punched in columns 11 to 21, the second in columns 22 to 32, etc. Both instructions and data are punched on identical cards.

The number of words on a card is indicated by the digit punched in column 10. Columns 7 to 9 indicate the memory location in which the first word is to be stored. When this card is read by the machine the first word is copied in this memory location and words two to six will automatically be stored in successive memory locations. That is, if columns 7 to 9 read 315, the word punched in columns 11 to 21 will be stored in 315, the word punched in columns 22 to 32 will be stored in 316, etc.

The card in Fig. 3 contains four words to be stored in 401, 402, 403, 404, respectively.

2-14. Preparing Punched Cards

Cards can be punched on a machine called the Key Punch or Card Punch (Fig. 4). This has a keyboard somewhat like a typewriter, but instead of printing characters on a sheet of paper, it punches a card. The machine is loaded with unpunched cards and, when the proper buttons are pressed, the first column of the first card is placed in position to be punched. Cards are punched from left to right, one column at a time, with either numeric or alphabetic symbols. When the key corresponding to an alphabetic character is depressed, the corresponding pair of holes is punched; for example, 0 and 3 for the letter T.

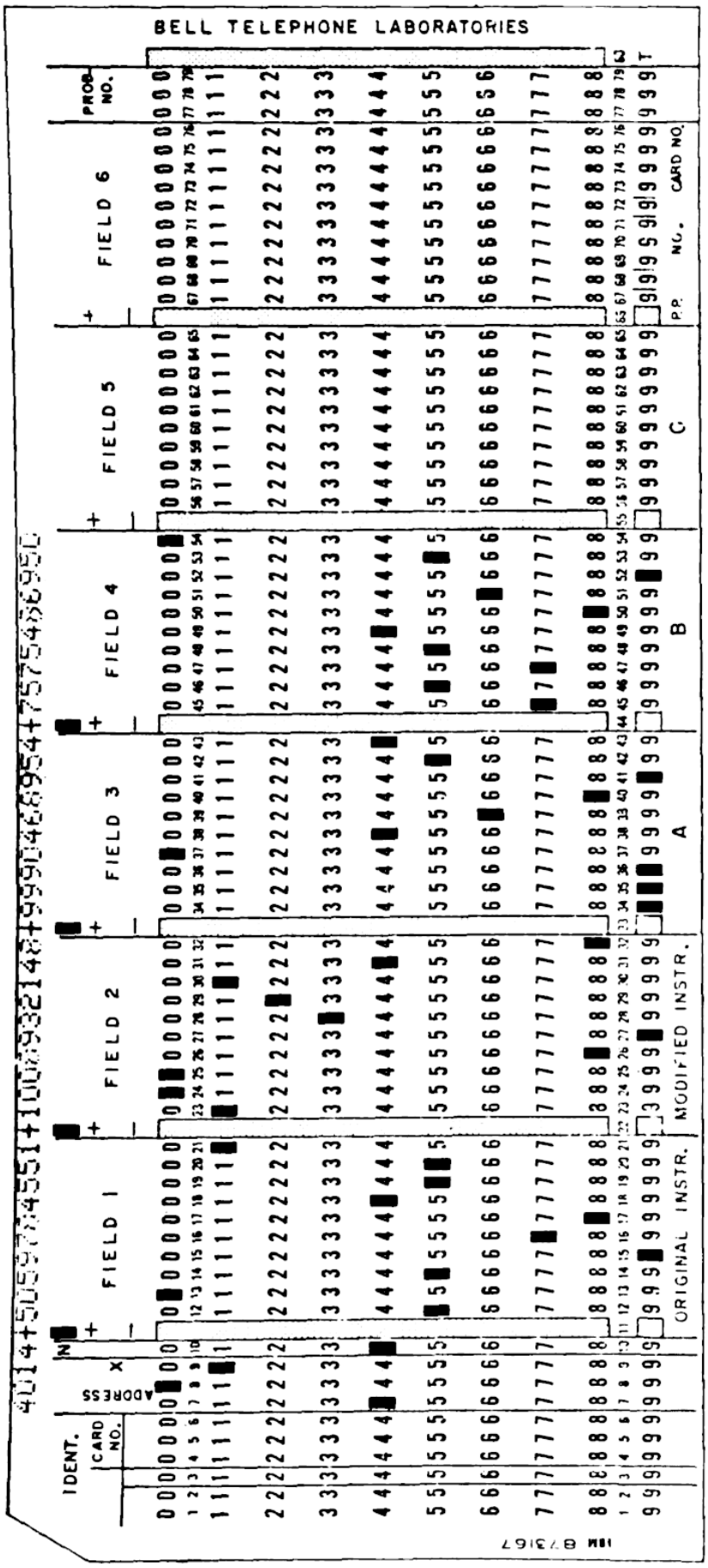


FIG. 3. (*Reproduced by permission of Bell Telephone Laboratories.*)

Some punch models can also print the letter T in the same column, just above the 12 row.

Cards should always be checked carefully to be sure the proper information has been punched. Preferably, checking should not be

FIG. 4. The IBM Type 026 Card Punch. (*Reproduced by permission of International Business Machines Corporation.*)

done by the same person who did the punching; the same mistake might be made twice.

Human errors can be all too easily introduced in transcribing from a handwritten sheet to punched cards. In particular, the distinction between the number 0 and the letter O, the number 1 and the letter I,

must be absolutely clear. It is wise to adopt some convention for writing these symbols, such as Ø for the letter O.

There are several techniques for checking cards. The brute-force method is to read the holes, column by column; but this is so tedious it is only done in an emergency.

If the cards were prepared on a printing key punch, the symbols can be read from the top of the card. If the cards were not printed when punched, a machine called an Interpreter can be used to print the information across the top of the card. Reading the printed symbols is far easier than reading the holes, but it is still relatively inefficient.

Yet another method involves the Tabulator. This is a device which produces a printed sheet on which each line represents the information on a single card. The printed sheet can be produced at a rapid rate and then proofread to check the cards.

The most automatic method is to use a Verifier. This device has a keyboard resembling the one on the Key Punch, but it cannot punch cards. The Verifier is loaded with cards that have already been punched. Then the operator proceeds to use the keyboard just as though the cards were going to be punched. In this case, however, the machine compares the key depressed with the holes already on the card. If the two agree, the card is advanced to the next column; if they do not, an error light flashes on. If, after three attempts, the key and the card do not agree, the card is notched in the incorrect column. A new card with the correct data must eventually be punched (on the Card Punch) to replace the erroneous card.

Whichever method is adopted, accuracy must be emphasized. After all, the machine cannot produce anything but incorrect results with incorrect data.

2-15. Loading

Every calculation begins by loading the program. By the term *loading* we mean that the information on the cards or tape is "read" by the input unit and stored in the proper memory locations.

This process is controlled by a loading program, which, for the time being, we will assume to be built into the circuitry of the machine.

We need not concern ourselves for the moment with the detailed

operation of the loading program; the only thing we need to know is that it automatically reads the cards and stores the information (see Sec. 2-13). It is important to realize, however, that during loading the instructions on the cards are not executed, but only copied in the memory. The actual computation does not begin until the loading operation is completed.

Constants are usually loaded together with the program. They will remain unchanged throughout the computation so they can be stored at the very beginning, once and for all.

On the other hand, as described in Sec. 2-9, it is often advisable to store the data for one computation at a time, completing that calculation before new data are read in. In that way we can use the same storage locations for the same *variables* each time; consequently the same program will compute the same *function* of these variables each time. Consider Example 1, Sec. 2-12. If we always store x, y, and z in locations 201, 202, and 203, then the program we wrote will compute the corresponding value of f and store it in 501. Indeed, this is the feature that makes it possible to use programs again and again with different data.

Clearly, we must have a method for interrupting the loading routine to begin computation, and also a method for loading the data one group at a time.

The initial loading can be interrupted by punching a special symbol on a card. When this card is read the input unit recognizes that loading is ended and control of the machine operation is to be *transferred* from the loading routine to the calculation program.

In the Bell scheme this is accomplished by a card with a zero in column 10. You will recall that column 10 is normally used to specify the number of words on this card; however, a card with zero words stops the loading. A card of this type is called a *transfer card.*

Since the location of the first instruction of the program is arbitrary, the machine must be told where to begin. This is accomplished by punching the location of the first program step in columns 7, 8, and 9 of the transfer card. Finally, for identification purposes a problem number must be punched in columns 77, 78, 79 of the transfer card.

A typical transfer card for problem number 032 for which the program begins in location 098 is shown in Fig. 5.

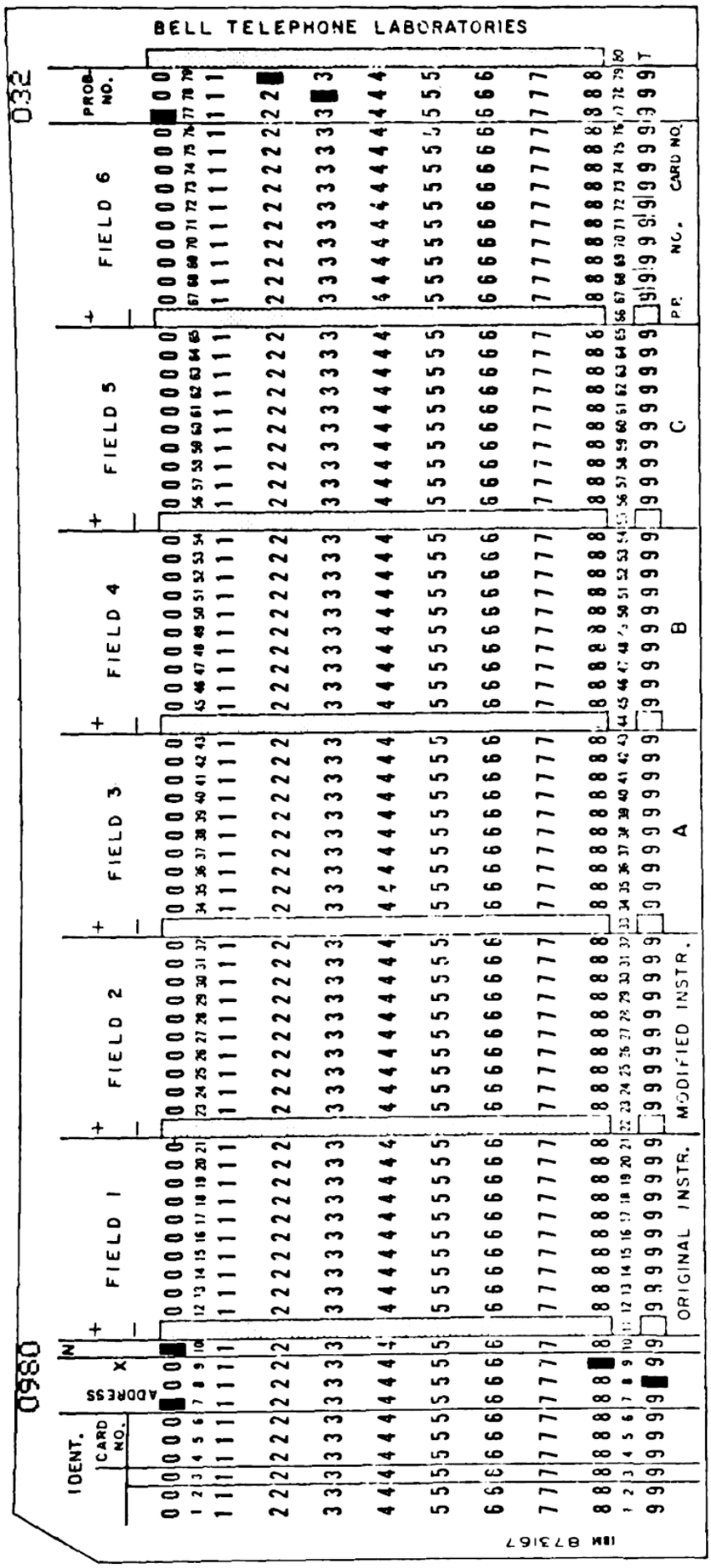


FIG. 5. (*Reproduced by permission of Bell Telephone Laboratories.*)

2-16. The READ Instruction

The reading of new data is handled by a READ instruction. Instructions of this type become part of the program in the same way as the arithmetic instructions.

A READ order is an instruction of the second type—one in which the first digit is zero and the first group of three digits represents an instruction code rather than the address of a number. In its general form, a READ instruction is

0 400 B C

400 is the code for READ. This instruction means "read cards and store the information in successive memory locations from B to C".

Thus the instruction

0 400 201 248

would read data from cards into locations 201, 202, 203, etc., up to and including 248. The data should be punched in standard form on as many cards as necessary. For example, they could be punched six words per card on eight cards, three words per card on sixteen cards, etc. For the protection of the programmer, the machine examines columns 7, 8, and 9 of each card to be sure the data will be stored as intended. Thus, if the machine is given the above instruction and the card it reads has 308 in columns 7, 8, and 9, it will stop and indicate an error.

If the 48 words are punched four per card, the first card read must have 201 in columns 7, 8, and 9; the second must have 205, the third 209, etc. Each card must also have a 4 in column 10.

There is no requirement that each card must have the same number of words. The following arrangement is entirely permissible:

Cols. 7–9	Col. 10
201	6
207	2
209	5
214	1
etc.	etc.

After the machine has stored a word in location C, it automatically stops reading cards and proceeds with the next instruction. There is no need for a transfer card in this case.

2-17. The PUNCH Instruction

An instruction similar to the READ instruction will activate the output unit to punch cards.

0 410 B C

instructs the machine to punch the contents of locations B through C.

These cards are identical in form to the cards used as input. The information is punched six words per card, except the last card. If the total number of words is not a multiple of six, this card will have fewer words. In any event, column 10 as usual indicates the number of words on that card and columns 7, 8, 9 will give the memory address from which the first word came.

Thus the instruction

0 410 504 508

will produce one card containing five words—the contents of storage locations 504 through 508. Similarly

0 410 684 703

will produce four cards labeled as follows:

Cols. 7–9	Col. 10
684	6
690	6
696	6
702	2

After the contents of location C have been punched, the machine proceeds to the next instruction.

2-18. Unconditional TRANSFER

With one additional type of instruction, we will be in a position to write programs that are complete. This is the command that interrupts the usual flow of the program. As you will recall, the instructions have been stored in the memory in sequence and normally the instruction in 185 will be executed immediately after the instruction in 184.

The unconditional TRANSFER instruction

0 203 B C

breaks this sequential operation. On encountering this order, the machine skips to C for its next instruction. (The B address is ignored.)

Thus if location 117 contains

117: 0 203 000 246

the next instruction executed after 117 will be the instruction in 246, instead of 118.

Our immediate use of this instruction will be to start the calculation over again. If the last instruction of the program is an unconditional TRANSFER back to the first instruction, the program will be repeated. In fact, it will continue to be repeated, ad infinitum, unless something stops it. In practice, if the program involves a READ instruction, and there are no more cards in the read hopper, the machine will stop in trying to execute the READ order.

It is important to note the distinction between a transfer *card* and a transfer *instruction*. They both accomplish a transfer of control, but they are used under different circumstances. The transfer card is never recorded in the memory. As soon as it is read by the input unit, control is transferred to the memory location specified on that card.

On the other hand, a transfer instruction is stored in the memory just like an arithmetic instruction. It is only when this instruction is executed in the course of a calculation that it breaks the regular sequence of program steps.

2-19. A Complete Example

We are now equipped to finish the programs we began in Sec. 2-12.

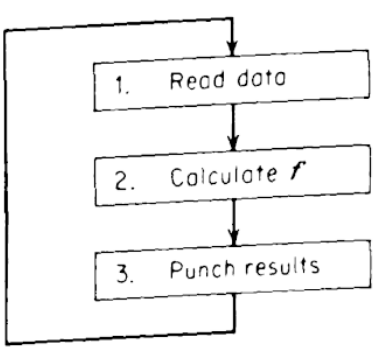


Fig. 6

By means of a schematic diagram, called a *flow chart*, we can indicate the sequence in which the major operations are to be performed (Fig. 6). All the examples in Sec. 2-12 can be described in the same simple way.

The program will begin with a READ instruction and end with a PUNCH instruction followed by an unconditional TRANSFER back to the beginning.

In Example 1, we planned the computation of

$$f = \frac{x + 8y}{y - 2z}$$

and our data were stored in 201, 202, and 203. The READ instruction that begins this program will therefore be

100: 0 400 201 203

We store it in location 100 to precede the computation that we programmed to begin in 101.

At the end of the calculation, the instruction in 105 stored f in 501. We could now give a PUNCH instruction

106: 0 410 501 501

This would punch a card containing a single word—the contents of 501. But this would not be wise; it would be much better to have some additional information on the card to identify it as the result of a particular computation. A simple way to accomplish that, in this case, would be to punch x, y, and z on the card along with f. This tells you immediately the data that were used, together with the result.

It would therefore be advisable to modify our original program so as to store f in 204, then give a PUNCH order to punch the contents of 201 through 204.

```
...   . ... ... ...
105:  4 400 000 204   f = s ÷ u
106:  0 410 201 204   PUNCH
```

Finally, we complete the program with a return to 100:

```
107:  0 203 000 100
```

The entire program is now as follows:

```
100:  0 400 201 203   READ x, y, z
101:  3 301 202 000   p = 8 · y
102:  1 201 000 400   s = x + p
103:  3 302 203 000   t = 2 · z
104:  2 202 000 000   u = y − t
105:  4 400 000 204   f = s ÷ u
106:  0 410 201 204   PUNCH
107:  0 203 000 100   TRANSFER to 100
301:  8 000 000 050   CONSTANT
302:  2 000 000 050   CONSTANT
```

These instructions are punched on cards and constitute the program deck.

Note that the constants 8 and 2 have been stored in floating point at the end of the program. These are to be loaded with the program deck for reasons previously explained.

When the problem is to be run, the program deck is followed by a transfer card:

Cols. 7–9	Col. 10	Cols. 77–79
100	0	Problem number

and this is followed by as many sets of x, y, z as desired.

The machine will load the program. When it comes to the transfer card it will stop loading and perform the instruction in location 100. This will call for the first data card. As soon as information has been read into 201, 202, and 203 the computation of f begins. The instructions are now performed in sequence. After f has been computed, the instruction in 106 will cause a card to be punched, and the order in 107 will send the machine back to 100. This will call for more data. The process continues until the machine runs out of data cards.

2-20. An Additional Example

In a similar way we can complete Example 2 of Sec. 2-12.

Example 2:

$$\frac{1}{f} = \frac{1}{x} + \frac{1}{y} + \frac{1}{z}$$

100:	0	400	201	203	READ x, y, z
101:	4	301	201	401	$p = 1/x$
102:	4	301	202	402	$q = 1/y$
103:	4	301	203	000	$s = 1/z$
104:	1	000	402	000	$t = s + q$
105:	1	000	401	000	$u = t + p$
106:	4	301	000	204	$f = 1/u$
107:	0	410	201	204	PUNCH
108:	0	203	000	100	TRANSFER
301:	1	000	000	050	CONSTANT

2-21. Codes for Common Functions

One of the advantages of an interpretive system is that some of the most commonly used functions can be calculated by giving a single

instruction. We can immediately increase our vocabulary to include these in the Bell code. All of them involve operating on one number to produce a result, for example, $\sqrt{x}$ or sin x. These instructions are all of the same form

0 Op 2 B C

In each case, the operation is performed on the number in storage location B and the result is stored in C.

The operations are listed in Table 2. You will note that square root, absolute value, trigonometric functions, logarithms, and exponentials can all be found using a simple instruction. There is also symmetry of a sort: 301 and 302 refer to the base e; 351 and 352 refer to the base 10. Also 303, 304, and 305 refer to angular measure in radians, while 353, 354, and 355 refer to degrees.

Table 2

Code (op 2)	Result
300	$\overline{C} = \sqrt{\overline{B}}$
301	$\overline{C} = e^{\overline{B}}$
302	$\overline{C} = \log_e \overline{B}$
303	$\overline{C} = \sin \overline{B}$ ($\overline{B}$ in radians)
304	$\overline{C} = \cos \overline{B}$ ($\overline{B}$ in radians)
305	$\overline{C} = \tan^{-1} \overline{B}$ ($\overline{C}$ in radians)
350	$\overline{C} = \lvert\overline{B}\rvert$
351	$\overline{C} = 10^{\overline{B}}$
352	$\overline{C} = \log_{10} \overline{B}$
353	$\overline{C} = \sin \overline{B}$ ($\overline{B}$ in degrees)
354	$\overline{C} = \cos \overline{B}$ ($\overline{B}$ in degrees)
355	$\overline{C} = \tan^{-1} \overline{B}$ ($\overline{C}$ in degrees)

Example 1: Given a, b, and θ (in degrees), find c, where

$$c^2 = a^2 + b^2 - 2ab \cos \theta$$

Store a in 201
b in 202
θ in 203
c in 204
-2 in 205

100:	0	400	201	203	READ a, b, θ
101:	3	201	201	206	$a^2 = a \cdot a$
102:	3	202	202	207	$b^2 = b \cdot b$
103:	0	354	203	000	$p = \cos \theta$
104:	3	000	201	000	$q = a \cdot p$
105:	3	000	202	000	$r = b \cdot q$
106:	3	000	205	000	$s = -2 \cdot r$
107:	1	000	206	000	$t = s + a^2$
108:	1	000	207	000	$u = t + b^2$
109:	0	300	000	204	$c = \sqrt{u}$
110:	0	410	201	204	PUNCH
111:	0	203	000	100	TRANSFER
205:	−2	000	000	050	CONSTANT

Example 2: Given x, find cosh x, where

$$\cosh x = \frac{e^x + e^{-x}}{2}$$

Store x in 201
cosh x in 202
2 in 203
1 in 204

100:	0	400	201	201	READ
101:	0	301	201	205	$p = e^x$
102:	4	204	000	000	$q = 1 / p \ (= e^{-x})$
103:	1	205	000	000	$r = p + q$
104:	4	000	203	202	$\cosh x = r / 2$
105:	0	410	201	202	PUNCH
106:	0	203	000	100	TRANSFER
203:	2	000	000	050	CONSTANT
204:	1	000	000	050	CONSTANT

Example 3: Given x and y $(y > 0)$, find

$$z = y^x$$

First we note that

$$\log z = x \log y$$

$$z = \text{antilog } (x \log y) = 10^{x \log_{10} y}$$

Store x in 201
y in 202
z in 203

```
100:  0 400 201 202  READ x, y
101:  0 352 202 000  p = log10 y
102:  3 000 201 000  q = x · p
103:  0 351 000 203  z = 10^q
104:  0 410 201 203  PUNCH
105:  0 203 000 100  TRANSFER
```

In all the examples given above, the program deck is followed by a transfer card and as many sets of data as desired.

Problems for Chapter 2

The operations necessary to solve these problems are described in the preceding chapters. Before actually testing a program on a machine, however, it would be wise to study some of the testing procedures described in Chap. 5.

1. Write all the prime numbers between 1 and 50, and their reciprocals
   (*a*) in binary representation
   (*b*) in floating-point decimal form.
2. Write a program to compute

$$y = \frac{2x^3 + 3x^2 - 5x + 2}{4x^2 - x - 2}$$

3. Complete the programming of Example 3, Sec. 2-12.
4. Revise the program for Example 1, Sec. 2-12, so that only the storage locations between 116 and 130 are used.
5. Calculate

$$f = \frac{\dfrac{1}{x}}{\dfrac{1}{1+y} + \dfrac{1}{1+\dfrac{1}{z}}}$$

6. Write a program to compute

$$f = \frac{3 \log x + e^{\cos y}}{\sqrt{2z}}$$

where y is given in radians.

7. A function is known to pass through the points (x_0,y_0), (x_1,y_1), (x_2,y_2), and (x_3,y_3). The x's are in increasing order. We want to find the value of y corresponding to x, where x lies between x_1 and x_2. Consult a book on numerical analysis and write a program to interpolate for y if

(*a*) x_0, x_1, x_2, and x_3 are equally spaced

(*b*) they are not equally spaced.

8. Consult a book on numerical analysis and write a program to integrate by Simpson's rule

$$\int_0^{\pi} \cos^2 x \, dx$$

3

Introduction to Loops and Branches

3-1. Some General Remarks

Having already encountered the arithmetic operations and codes for special functions, the reader may be surprised that so much of the book lies ahead. Actually, in many if not most programs, fewer than half of the instructions are purely arithmetic; the rest are concerned with other aspects of the problem. Many of these attempt to duplicate the sort of decisions the human operator of a desk calculator might be called upon to make and without commands of this type it would be impossible to make computation fully automatic. As we shall see, manipulations that could be described to a person in a few sentences may take many instructions to a machine; and it is here, rather than in the arithmetic operations, that most programming errors are made.

The more complicated the problem, the more the programmer depends on these logical operations; and the larger the vocabulary of instructions of this type, the more flexible and versatile the machine.

In the sections which follow, we will examine a few cases in which the need for this new kind of instruction is apparent.

3-2. A Simple Counting Loop

Let us consider once again the computation

$$c^2 = a^2 + b^2 - 2ab \cos \theta$$

When we programmed this in the preceding chapter, we assumed a, b, and θ varied in each calculation. Suppose, however, that a and b are constant for twenty values of θ at a time.

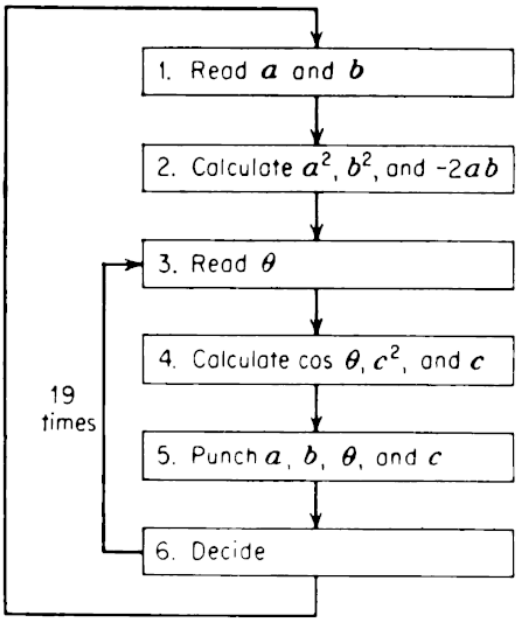


Fig. 7

Of course, nothing would stop us from proceeding just as before, but clearly computing time would be saved if a^2, b^2, and $(-2ab)$ were calculated only once for every twenty values of θ. We can accomplish this with the flow diagram of Fig. 7.

First a card is read containing a and b. This is followed by twenty cards with different values of θ. Then comes another card with a and b, followed by twenty θ cards; etc.

Note that there are two paths leading out of box 6: one back to box 3 and the other back to box 1. Boxes 3, 4, 5, and 6 constitute a "loop" which is executed a total of twenty times before the alternate path back to box 1 is taken. We must provide some means for choosing between branches at box 6 so that we go back to box 3 nineteen out of twenty times.

Before programming this in detail let us consider two other examples.

3-3. A Loop with Address Modification

In Sec. 2-12, we discussed a method for evaluating a polynomial of the form

$$y = x^4 + a_1x^3 + a_2x^2 + a_3x + a_4$$

If a_1, a_2, etc., are stored in the successive memory locations 301, 302, etc., and x is in 201, the program can be written as

100:	0	400	201	201	READ x
101:	4	201	201	000	$p = 1$
102:	3	000	201	000	$p = p \cdot x$
103:	1	000	301	202	$y = p + a_1$
104:	3	000	201	000	$p = p \cdot x$
105:	1	000	302	202	$y = p + a_2$
106:	3	000	201	000	$p = p \cdot x$
107:	1	000	303	202	$y = p + a_3$
108:	3	000	201	000	$p = p \cdot x$
109:	1	000	304	202	$y = p + a_4$
110:	0	410	201	202	PUNCH
111:	0	203	000	100	TRANSFER

This differs in some respects from our original program. Of course, we have included the READ, PUNCH, and TRANSFER instructions, but that is not all. For example, the instruction in 101, which serves to put a 1 in 000, does not seem to be necessary. Also, the instruction in 103 stores a result in location 202 for no apparent reason; this number is never called out of 202 and is, in fact, erased when the instruction in 105 sends a new result to 202. The result of the order in 107 destroys this and is itself erased, in turn, by the instruction in 109.

Before explaining why these queer instructions were written, let us look at the symbolic description of the program as given to the right of the numerical instructions. Here again we have departed from previous practice and instead of assigning a new symbol for each function, we have used the same symbol whenever the same memory location is used. Thus, every function stored in 000 is called p, and every function stored in 202 is called y. This leads to some strange-looking "equations", such as

$$p = p \cdot x$$

But this is not an algebraic equation. If it were, and x were arbitrary, it would imply that $p = 0$. Instead, this equation means, as we have pointed out before, that the number p is multiplied by x and becomes the new number called p.

Notice that the instructions now occur in a pattern repeated four times. The pair in 102 and 103 is very much like the pair in 104 and 105. In fact, 102 and 104 are identical; and 103 and 105 differ only by 1 in the B address. Similarly the instructions in 106 and 108 are

the same as 104; and 105, 107, and 109 differ by 1 in the B address.

Symbolically, the pattern is

$$p = p \cdot x$$

$$y = p + a_i$$

These instructions are repeated with $i = 1, 2, 3, 4$.

If it were possible to instruct the machine to execute this pair four times, modifying the instruction $y = p + a_i$ each time so that a_i is taken first from 301, then from 302, etc., we could program the polynomial computation as in Fig. 8. Originally, box 4 is executed with $i = 1$. Then the loop from box 4 back to box 3 is executed three times, each time with i increased by 1. As we shall see, a scheme like this will permit us to program the computation in seven instructions instead of twelve. A repetitive calculation of this type is called a loop with address modification.

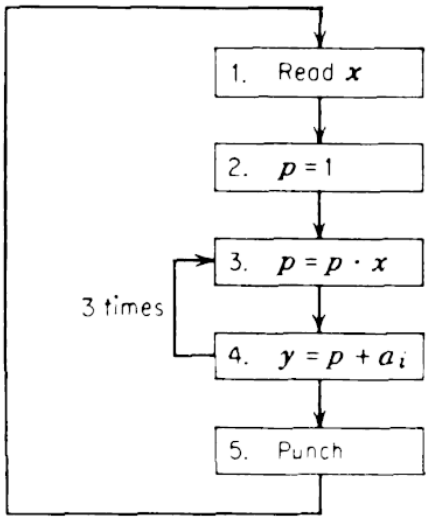


FIG. 8

Now we can understand why the instruction in 101 was added and why information was stored in 202 that was never used. It was done to point out the pattern. By setting $p = 1$ initially, we could write 102 as $p = p \cdot x$, identical to 104, 106, and 108. Also, since we wanted the final result, calculated in 109, to be stored in 202, we had to write

109: 1 000 304 202

There was no harm in using 202 beforehand and it had the advantage of making 103, 105, 107, and 109 similar, though not identical.

We will return to this problem after one more example.

3-4. Loops without Counting

The loops we have discussed so far terminate after a fixed number of repetitions; but not all loops are of this kind. A typical loop of another type is illustrated by the Newton-Raphson method for finding the cube root of x. Given an approximate value y_0, an improved value y_1 can be obtained from the relation

$$y_1 = \frac{1}{3}\left(\frac{x}{y_0^2} + 2y_0\right)$$

and succeeding approximations are found from

$$y_{i+1} = \frac{1}{3}\left(\frac{x}{y_i^2} + 2y_i\right)$$

The process is repeated until two successive iterates agree to any desired accuracy:

$$|y_{i+1} - y_i| < \epsilon$$

The flow diagram is illustrated in Fig. 9.

We cannot be sure ahead of time how many times this loop will have to be executed before the desired accuracy is achieved. The decision to leave the loop is based upon a calculation made at the end of box 3. If $|y_{i+1} - y_i| \geq \epsilon$ we follow the path back to box 2; otherwise we can go on.

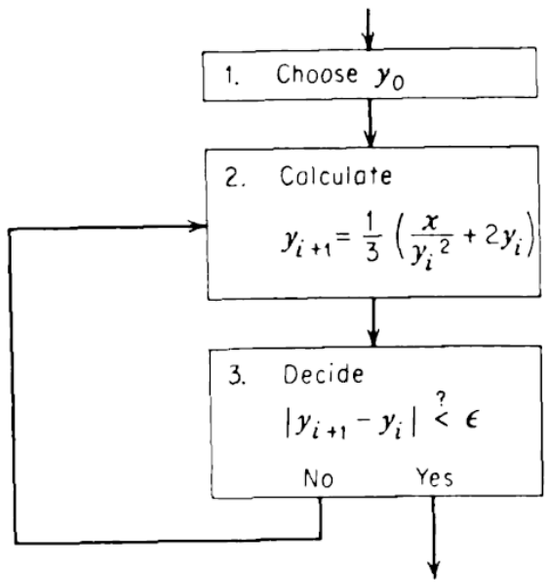


FIG. 9

3-5. A Simple Loop Box

Counting loops, with or without address modification, can be handled by devices variously called "loop boxes", "B registers", etc. To understand the operation of a loop box, let us go back to the example in Sec. 3-3.

You will recall that we want to execute the instructions

```
3 000 201 000
1 000 301 202
3 000 201 000
1 000 302 202
3 000 201 000
1 000 303 202
3 000 201 000
1 000 304 202
```

One way to do this is to assign each instruction to a separate storage

location; but, it would be more efficient to program this as a loop. We will store only the two instructions

3 000 201 000
1 000 301 202

Then, somehow, we want to execute these instructions three more times, varying the B address of the second instruction by 1 each time.

Suppose that somewhere in the machine the number

0 000 001 000

is stored, and just before the second instruction is executed, this number is added to it:

$$\begin{array}{rr} & 1\ 000\ 301\ 202 \\ + & 0\ 000\ 001\ 000 \\ \hline & 1\ 000\ 302\ 202 \end{array}$$

The instruction actually performed would be the sum

1 000 302 202

although the instruction stored in the memory is still

1 000 301 202

If, the next time through the loop, the number added were

0 000 002 000

the instruction performed would be

1 000 303 202

(Note that these additions are in fixed-point rather than floating-point arithmetic.) The instruction stored in the memory is still unchanged.

That is the idea of a loop box. Just before an instruction is executed, the content of the loop box is added to it and the instruction actually performed is the sum of the two.

Clearly we do not want this to happen to every instruction. In the particular case we are considering, the instruction

3 000 201 000

should not be modified. We must therefore *tag* the instructions to be modified by some method so that those instructions and no others will be modified by adding the contents of the loop box. The method

that has been adopted in the Bell scheme is to tag instructions to be modified by a minus sign. So far we have not needed to distinguish between positive and negative instructions, but now negative instructions will have a definite meaning. Negative instructions are tagged, which means that the contents of the loop box will be added to them just before they are executed. (When this addition is performed the instruction is given a positive sign; otherwise we have

$$\begin{array}{r} -1\ 000\ 301\ 202 \\ +0\ 000\ 001\ 000 \\ \hline -1\ 000\ 300\ 202 \end{array}$$

which would not be the desired result.)

The instruction controlling the loop immediately follows the last instruction of the group to be repeated. It contains the following information:

1. Which addresses are to be modified (A; B; C; A and B; A and C; B and C; or all three)
2. How many times the instructions are executed
3. The address of the first instruction of the loop

These three functions are assigned to the Op 2, B and C parts of a LOOP instruction.

Op 2	Result
100	Modify only A
010	Modify only B
001	Modify only C
110	Modify A and B
101	Modify A and C
011	Modify B and C
111	Modify A, B, and C

The B part of the LOOP instruction specifies the number of times the instructions are to be performed and the C address is the location of the first instruction of the loop.

For example, a LOOP instruction in location 119,

119: 0 101 015 105

means, "execute the instructions between 105 and 119 fifteen times, modifying the A and C addresses of all negative instructions just before they are executed".

As another case,

239: 0 011 024 210

means, "execute the instructions between 210 and 239 twenty-four times, modifying the B and C addresses of all negative instructions just before they are executed".

Returning to the polynomial problem, the program can now be written as

100:	0 400 201 201	READ x
101:	4 201 201 000	$p = 1$
102:	3 000 201 000	$p = p \cdot x$
103:	−1 000 301 202	$y = p + a_i$
104:	0 010 004 102	LOOP to 102
105:	0 410 201 202	PUNCH
106:	0 203 000 100	TRANSFER

This is what will happen when the program is executed. The instructions in 100, 101, and 102 will be performed unchanged. Then the loop box will be added to the instruction in 103 because that instruction is negative. Initially the content of the loop box is zero, so this instruction is executed without modification.

The instruction in 104 is the LOOP instruction. When it is executed the following sequence of operations takes place:

1. 010 indicates that the B address alone is to be modified. A 1 is therefore added to the B part of the loop box giving

L: 0 000 001 000

where we have used the symbol L for the loop box.

2. The number in the B part of the loop box (001) is compared with the number in the B part of the LOOP instruction which describes how many times the orders are executed (004). Since they do not agree, the next instruction executed is taken from 102 (the C address of the loop instruction) instead of 105 (the next address in sequence).

We now go back and repeat the instructions beginning in 102. The instruction in 102 is again executed unchanged. The negative instruction in 103 is once again added to the loop box, which now contains

L: 0 000 001 000

The instruction actually performed is therefore

1 000 302 202

When we again come to the instruction in 104, the B part of the loop box is increased by 1 to read 002. It is again compared with 004 and since the two still disagree, we go back to 102.

This time the order in 103 is executed as the sum

103:	1 000 301 202
L:	0 000 002 000
order executed:	1 000 303 202

Once again, when 104 is executed we increment the B part of the loop box and go back to 102.

This time through, 103 is executed as

1 000 304 202

However, when 104 is executed, the B part of the loop box is increased by 1 and it now reads 004. This is the same as the B part of the original LOOP instruction. Therefore we do not go back to 102 but go on to 105 instead. Before going on, the loop box is cleared to zero, so that the next calculation begins with a clean slate.

The PUNCH and TRANSFER instructions are executed in the usual way and we are ready for the new value of x.

Actually, it is a good deal more complicated to describe what goes on in detail than to write a LOOP instruction. You simply decide which addresses are to be modified, how many times these instructions are executed, and where the loop begins.

3-6. Programming the Simple Counting Loop

We can now treat the counting loop of Sec. 3-2 by means of loop boxes. In this case we use the loop box merely to keep count and, since none of the instructions are to be modified, none of them are tagged with a negative sign. We could use any one of the loop instructions, but we will use Op 2 = 100 in this example.

The flow diagram of Fig. 7 is easily translated to the following program:

100:	0 400 201 202	READ a and b
101:	3 201 201 301	$r = a^2$
102:	3 202 202 302	$s = b^2$
103:	3 205 201 000	$p = -2 \cdot a$

104:	3 000 202 303	$u = p \cdot b$
105:	0 400 203 203	READ θ
106:	0 354 203 000	$p = \cos \theta$
107:	3 000 303 000	$p = p \cdot u$
108:	1 000 301 000	$p = p + r$
109:	1 000 302 000	$p = p + s$
110:	0 300 000 204	$c = \sqrt{p}$
111:	0 410 201 204	PUNCH
112:	0 100 020 105	LOOP to 105
113:	0 203 000 100	TRANSFER
205:	−2 000 000 050	CONSTANT

According to this program, the instructions from 105 to 112 will be executed twenty times (the B part of the instruction in 112) before going on to 113. This instruction then sends us back to 100 to read new values of a and b.

The decision in box 6 of Fig. 7 is made by the LOOP instruction in 112. In spite of the fact that none of the instructions are modified, the loop box continues to count (in this case, by adding 1 to the A part of L). If the count is less than 20, we go back to 105; when it reaches 20, the loop box is cleared to zero and we go on to 113.

3-7. A Summation Loop

We can also use loop boxes to handle finite summations, as described by the following example.

We are given two groups of numbers:

A group:	a_1 in 301
	a_2 in 302
	a_3 in 303
	a_4 in 304
B group:	b_1 in 305
	b_2 in 306
	b_3 in 307
	b_4 in 308

and we want to find

$$c = \sum_{i=1}^{4} a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

(The reader may recognize this as the scalar product C of two vectors A and B.)

The "brute-force" program might be

```
3 301 305 309   c1 = a1 · b1
3 302 306 310   c2 = a2 · b2
3 303 307 311   c3 = a3 · b3
3 304 308 312   c4 = a4 · b4
1 309 310 000   p  = c1 + c2
1 000 311 000   p  = p + c3
1 000 312 313   c  = p + c4
```

It would be better to write it as

```
2 313 313 313   c  = 0
3 301 305 000   p  = a1 · b1
1 000 313 313   c  = p + c
3 302 306 000   p  = a2 · b2
1 000 313 313   c  = p + c
3 303 307 000   p  = a3 · b3
1 000 313 313   c  = p + c
3 304 308 000   p  = a4 · b4
1 000 313 313   c  = p + c
```

Here we have set aside 313 to accumulate the sum, and each term is added to it as it is calculated. Now it is simple to see that we have a loop (Fig. 10), and the program becomes

```
      2 313 313 313   c  = 0
α:   −3 301 305 000   p  = ai · bi
      1 000 313 313   c  = c + p
      0 110 004  α    LOOP
```

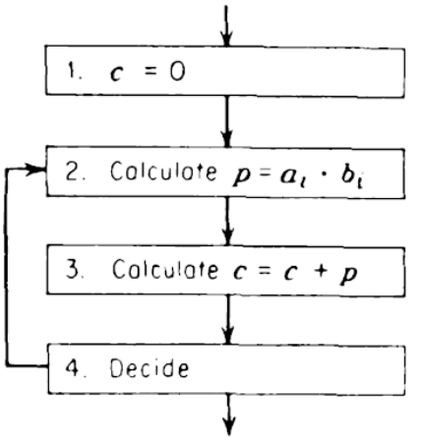


Fig. 10

The LOOP instruction Op 2 = 110 was used because the A and B addresses are both to be modified. In writing this program we have not bothered to assign memory locations to the program steps; these are arbitrary anyway. However, we have had to indicate the locations of the first instruction in the loop, and this we have done with a symbol α.

3-8. The MOVE Instruction

In some of the programs that follow it will be necessary to bring information from one part of the memory to another without altering it in any way. This can be accomplished by a MOVE instruction of the form

9 A B C

A tells us how many words are to be moved; B is the original location of the first word; and C is the location in which it is to be copied. For example,

9 003 801 923

will move 3 words from 801, 802, 803 to 923, 924, 925. Or again

9 007 116 164

will move 7 words from 116, 117, . . . , 122 to 164, 165, . . . , 170.

The information also remains where it was originally unless it was overwritten, as it would be in this case:

9 004 321 323

In the process of moving the information, the numbers originally in 323 and 324 are replaced, but they are correctly copied in 325 and 326. That is, the net effect is to move the original contents of

321 to 323
322 to 324
323 to 325
and 324 to 326

The information originally in 321 and 322 remains there as well.

Similarly

9 004 323 321

moves the contents of

323 to 321
324 to 322
325 to 323
and 326 to 324

and the information originally in 325 and 326 is still there.

There are two instructions that will move one word:

9 001 B C
and 9 000 B C

The difference is that in the former case the content of 000 is unchanged. In the latter case the number in B is put in C and 000 as well. This will be quite useful, as we will see.

3-9. Terminating Loops without Counting

The LOOP instructions we have discussed thus far are adequate to treat loops that terminate after an assigned number of repetitions. However, when a decision must be made on the basis of the magnitude or the sign of a result, new instructions must be added to our vocabulary.

Instructions of this kind permit us to choose between paths in the flow chart; they are therefore sometimes called "branching" operations. Since they also interrupt the usual sequence of instructions, they are a form of transfer order; however they are "conditional transfers" because they transfer control only when specific conditions are met.

Two useful instructions of this kind are

TRANSFER on sign: 0 201 B C

and TRANSFER on exponent: 0 202 B C

The "TRANSFER on sign" instruction examines the result of the most recent arithmetic operation. If it was positive, the machine executes the instruction in memory location B next; if it was negative, it executes the instruction in C next.

Thus, after the sequence

```
2 301 302 000
0 201 112 115
```

the machine would proceed to 115 if the number in 302 is greater than the number in 301; otherwise it would go to 112 for the next instruction.

The B part of the "TRANSFER on exponent" instruction serves a somewhat different purpose. It is used to specify a floating-point exponent. If the result of the last previous mathematical operation has an exponent greater than or equal to this number, the machine goes to C for its next instruction; otherwise it proceeds sequentially.

For example, after the sequence

```
115:  3 301 302 000
116:  0 202 048 105
```

the machine will go to 105 for the next instruction if the product of the numbers in 301 and 302 is greater than or equal to 10^{-2} (floating-point exponent 48); otherwise it will go on sequentially to 117.

Consider the example of Sec. 3-4. If it is adequate to find $\sqrt[3]{x}$ to three decimals, we can terminate the loop as soon as

$$|y_{i+1} - y_i| < 10^{-3}$$

Let us see the way a programmer tackles this problem. First it is necessary to assign storage locations for x and the trial values: y_i, the ith approximation, and y_{i+1}, the improved value. Let us say we will put

x in 300
y_i in 301
y_{i+1} in 302

We will begin by programming the computation for finding y_{i+1}, given y_i. To do this we will need the constant 3, which we will store in 303:

```
105:  3 301 301 000   p = y_i^2
106:  4 300 000 000   p = x/p
107:  1 000 301 000   p = p + y_i
108:  1 000 301 000   p = p + y_i
109:  4 000 303 302   y_{i+1} = p/3
```

This is completely straightforward. At the end of these instructions, we have computed

$$y_{i+1} = \frac{1}{3}\left(2y_i + \frac{x}{y_i^2}\right)$$

Now we must compare the new value y_{i+1} with the old y_i. We do this in two instructions: first we subtract the two values and then we do a "TRANSFER on exponent". For the time being we will leave the C address of the latter instruction unspecified:

```
110:  2 302 301 000   p = y_{i+1} - y_i
111:  0 202 047   C   TRANSFER on exponent
```

If the result of the subtraction had a floating-point exponent less than 47 (= 10^{-3}), the machine will go on to 112 for the next instruction. In that case, the latest approximation is the best. If desired we can punch the result y_{i+1} from 302.

```
112:  0 410 302 302
```

On the other hand, if we have not yet achieved the desired accuracy, we want to repeat the calculation, but now we must use the value just obtained, y_{i+1}, in the same role as we used y_i before. We can accomplish this by moving the number from 302 to 301, and begin the new loop with the instruction

104: 9 000 302 301

By using 000 as the A part, instead of 001, we can put this value in 000 also where it is ready for the next instruction (see Sec. 3-8).

If we assign this instruction to location 104, our program looks like this, so far:

```
104:  9 000 302 301
105:  3 000 000 000
106:  4 300 000 000
107:  1 000 301 000
108:  1 000 301 000
109:  4 000 303 302
110:  2 302 301 000
111:  0 202 047 104
112:  0 410 302 302
```

But there is one thing we have not yet considered. What about the very first trial? In order to cut down on the number of iterations, we want to start with as good an approximation as possible. In any particular problem we might have some idea of the expected range of the answer so our first guess might be a number near the middle of that range.

In any event we must precede the instruction in 104 by one which will put the first trial in location 302. Then the instruction in 104 will put this number in location 301 for the first calculation. If we put the first guess in 304, we can use the instruction

103: 9 001 304 302

You might wonder why we store this first trial value in 304 and bring it to 302 instead of storing it in 302 to begin with. The reason is simply that any number stored in 302 will be erased when other numbers are stored there during the calculation; but we will want to preserve our initial trial value for use in other calculations.

A complete program may read as follows:

```
102:  0 400 300 300   READ x
103:  9 001 304 302   MOVE 304 → 302
104:  9 000 302 301   MOVE 302 → 301
105:  3 000 000 000   p = y_i^2
106:  4 300 000 000   p = x/p
107:  1 000 301 000   p = p + y_i
108:  1 000 301 000   p = p + y_i
109:  4 000 303 302   y_{i+1} = p/3
110:  2 302 301 000   p = y_{i+1} - y_i
111:  0 202 047 104   TRANSFER on exponent
112:  0 410 302 302   PUNCH
113:  0 203 000 102   TRANSFER
303:  3 000 000 050   CONSTANT
304:  5 000 000 050   FIRST TRIAL
```

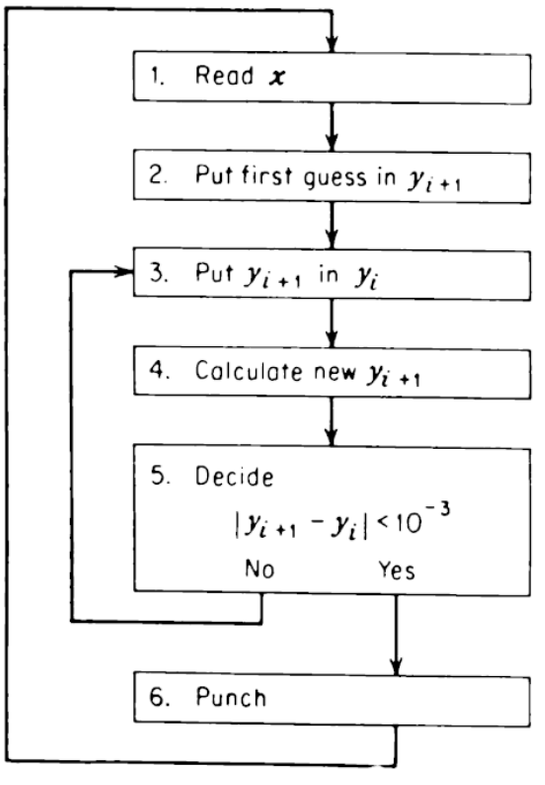


FIG. 11

The flow diagram is given in Fig. 11.

3-10. BRANCHING WITHOUT LOOPING

The use of branching instructions is not restricted to loops. There are many other occasions when they are useful.

Consider the quadratic equation

$$ax^2 + bx + c = 0$$

which has the roots

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In general, these roots may be complex, but if $b^2 - 4ac$ is positive, they are real.

To program this, we must set aside two storage locations for each root—one for the real and one for the imaginary part. When $\sqrt{b^2 - 4ac}$ is real, the imaginary part must be set equal to zero. Otherwise, the imaginary parts are $\pm\sqrt{|b^2 - 4ac|}/2a$. The flow diagram is given in Fig. 12.

If we store

a in 301	real part of first root	in 401
b in 302	imaginary part	in 402
c in 303	real part of second root	in 403
2 in 304	imaginary part	in 404
4 in 305		
0 in 306		

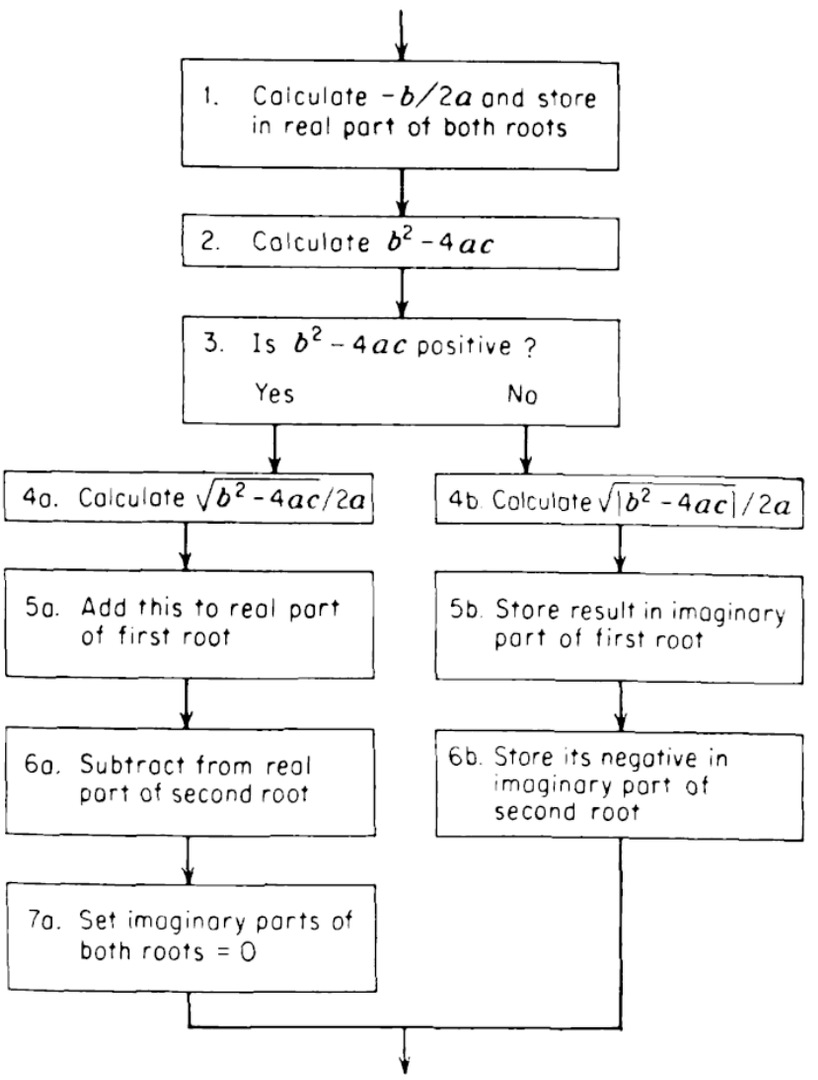


FIG. 12

the program reads as follows:

```
100:  3 301 304 405 }
101:  2 306 302 000 } box 1
102:  4 000 405 401 }
103:  9 001 401 403 }
```

```
104:  3 302 302 406 }
105:  3 305 301 000 }  box 2
106:  3 000 303 000 }
107:  2 406 000 407 }
108:  0 201 109 116   box 3
109:  0 300 407 000 }  box 4a
110:  4 000 405 408 }
111:  1 401 000 401   box 5a
112:  2 403 408 403   box 6a
113:  2 402 402 402 }  box 7a
114:  2 404 404 404 }
115:  0 203 000 121   TRANSFER
116:  0 350 407 000 }
117:  0 300 000 000 }  box 4b
118:  4 000 405 408 }
119:  9 001 408 402   box 5b
120:  2 306 408 404   box 6b
121:  NEXT INSTRUCTION
```

It is particularly important to understand why the transfer instruction in 115 is necessary. The instructions from 109 to 114 are executed only when $b^2 - 4ac$ is positive. In that case we want to skip the instructions from 116 to 120. This is accomplished by the TRANSFER in 115. If it were not there, the machine would go on to 116 every time.

At 121 the two branches of the program come together again, as shown in the flow diagram. When the roots are complex, 116 to 120 are executed and the machine goes on to 121 as usual. When the roots are real, the TRANSFER in 115 sends the machine to 121.

3-11. Address Modification Loops without Loop Boxes

The LOOP instruction we have discussed in earlier sections saves us a lot of trouble in programming, but situations will arise when we must use another method of programming loops. Although this alternative procedure is somewhat more complicated, it will indicate some of the general concepts of loop programs that we must understand completely.

There are four essential steps in writing a loop:

1. Initialization
2. Execution
3. Modification
4. Testing

The most obvious part, step 2, execution, refers to the instructions that are to be repeated. Step 3, modification, incrementing, or "updating" as it is often called, describes the process of changing the program and data for the next execution of the loop. Step 4, the test, determines whether the loop has been "satisfied", that is, if it has been executed the requisite number of times. If so, we can go on; but if not, we must go back to step 2.

This leaves only step 1 to be explained, but the need for it will be obvious as we proceed.

Let us look at the polynomial loop of Sec. 3-3 again and write the loop without using a loop box. We begin by writing down the "execution" part of the loop:

```
102:  3 000 201 000
103:  1 000 301 202
```

This is the way the instructions are to be executed the first time. The second time, however, the B address of the instruction in 103 is to be increased by 1. This is accomplished by a new instruction: "Add to B". In this case the instruction we want is

```
104:  0 060 103 001
```

The code Op 2 = 060 indicates that this is an "Add to B" instruction; 103 is the location of the instruction to be altered and 001 is the amount to be added to the B address. In general, the order

```
0 060  B   C
```

will add C (not the *contents* of C, but the number C itself) to the B address of the instruction in B. (This addition takes place in fixed arithmetic, rather than floating.)

There are parallel orders for adding to the A, B, and C addresses:

Op 2	Order
600	Add to A
060	Add to B
006	Add to C

For example, if the contents of 112 are

112: 3 515 317 482

after the instruction

0 600 112 002

the contents of 112 will be

112: 3 517 317 482

because 002 is added to the A address of the instruction in 112.

Note that these orders actually change the instructions stored in the memory. This is contrary to the operation of a loop box which leaves the instructions in the memory unchanged, but alters them just before execution.

We have used the "Add to B" instruction to modify, increment, or "update" the B address of the instruction 103 by 001. Now we must test to see if the loop is "satisfied". We must put an instruction in 105 that will either send us back to 102 or let us continue with 106.

How are we to tell if we have performed the loop the correct number of times? We can find out by examining the instruction in 103. This instruction changes every time 104 is executed.

Let us look at the contents of 103 while the loop is executed four times (Table 3).

Clearly we don't want to execute the loop again because we don't want to execute the instruction

103: 1 000 305 202

Therefore, when we test and find this instruction in location 103, we know the loop is satisfied and we go on to 106.

In this case, since it is the B address that changes, we use a "TRANSFER on B" instruction:

105: 7 103 305 102

This is an instruction of the first type again, that is, one in which the first digit is not zero. Op 1 = 7 means this is a "TRANSFER on B" instruction. The A address (103) gives the location of the instruction to be tested; the B address (305) gives the value when the loop is satisfied; and the C address (102) tells where to go if the loop is *not* satisfied.

Table 3

Location of instruction executed	Instruction executed	Contents of location 103
102	3 000 201 000	1 000 301 202
103	1 000 301 202	1 000 301 202
104	0 060 103 001	1 000 302 202
105	TEST	1 000 302 202
102	3 000 201 000	1 000 302 202
103	1 000 302 202	1 000 302 202
104	0 060 103 001	1 000 303 202
105	TEST	1 000 303 202
102	3 000 201 000	1 000 303 202
103	1 000 303 202	1 000 303 202
104	0 060 103 001	1 000 304 202
105	TEST	1 000 304 202
102	3 000 201 000	1 000 304 202
103	1 000 304 202	1 000 304 202
104	0 060 103 001	1 000 305 202
105	TEST	1 000 305 202

Here again there are parallel instructions:

Op 1	Order
6	TRANSFER on A
7	TRANSFER on B
8	TRANSFER on C

Consider the instruction

271: 6 255 394 240

This means, "If the A address of the instruction in 255 is not 394 go back to 240; if it equals 394 go on to 272".

Similarly,

345: 8 319 040 305

means, "If the C address of the instruction in 319 is not equal to 040 go to 305; otherwise go on to 346".

Now we are in a position to understand the problem of initialization. After the loop was satisfied in the polynomial we have been discussing, the instruction in 103 is still

103: 1 000 305 202

When we go back to begin the calculation for a new value of x we must somehow restore this instruction to its initial value:

103: 1 000 301 202

We can do this by using the "SET B" instruction

100: 0 050 103 301

Op 2 = 050 is the "SET B" code; 103 is the location of the instruction to be changed; and 301 is the number to replace whatever was in the B address.

Once again, we have the related instructions

Op 2	Order
500	SET A
050	SET B
005	SET C

The instruction

098: 0 500 641 119

means, "Replace the A address of the instruction in 641 by 119"; and

533: 0 005 416 777

means, "Replace the C address of the instruction in 416 by 777".

Returning to the polynomial program, in its completed form it becomes

99:	0	400	201	201	READ x
100:	0	050	103	301	INITIALIZE
101:	4	201	201	000	$p = 1$
102:	3	000	201	000	$p = p \cdot x$
103:	1	000	(301)	202	$y = p + a_i$
104:	0	060	103	001	INCREMENT
105:	7	103	305	102	TEST
106:	0	410	201	202	PUNCH
107:	0	203	000	099	TRANSFER

Actually, the instruction in 101 is part of the initialization operation, too, since it gets things ready for the first execution of the loop. We have put the B part of 103 in parentheses because it is modified by the program.

It would have been possible to test *before* incrementing. The orders

necessary for this can be found in our vocabulary and it would be instructive for the student to try it.

3-12. Symbolic Representation of Some of the Preceding Instructions

We can describe the instructions introduced in the preceding section by devising a new symbolism. We have already used a bar to denote "the contents of"; for example, $\overline{B}$ represents "the contents of address B"; or $\overline{415}$ means "the contents of location 415".

Suppose we look at the contents of 415:

415: 3 521 862 385

This has an A part (521), a B part (862), and a C part (385). We can therefore speak of "the C part of the contents of location A" and denote it by $\overline{A}_c$.

Consider the following situation:

301: 1 517 644 703
517: 2 304 303 307
644: 6 302 115 303
703: 4 309 302 801

Then

$$\overline{\overline{301}_b} = 644$$
$$\overline{\overline{517}_a} = 304$$
$$\overline{\overline{703}_a} = 309$$
$$\overline{\overline{703}_b} = 302$$
$$\overline{\overline{301}_a} = 517, \text{ etc.}$$

Using this terminology we can construct Table 4.

We have added to this table the set of instructions "SUBTRACT from A, B, or C". These are very similar to "ADD to A, B, or C".

3-13. Summing an Infinite Series

In scientific calculations it is frequently necessary to approximate the sum of an infinite series. For example (for $x^2 < 1$)

$$\sin^{-1} x = \frac{x}{1} + \frac{1}{2}\frac{x^3}{3} + \frac{1}{2}\frac{3}{4}\frac{x^5}{5} + \frac{1}{2}\frac{3}{4}\frac{5}{6}\frac{x^7}{7} + \cdots$$

Table 4

Op 1	Op 2	Function
6		If $\overline{A}_a \neq B$ execute $\overline{C}$ next
		If $\overline{A}_a = B$ proceed sequentially
7		If $\overline{A}_b \neq B$ execute $\overline{C}$ next
		If $\overline{A}_b = B$ proceed sequentially
8		If $\overline{A}_c \neq B$ execute $\overline{C}$ next
		If $\overline{A}_c = B$ proceed sequentially
	500	$\overline{B}_a = C$
	050	$\overline{B}_b = C$
	005	$\overline{B}_c = C$
	600	$\overline{B}_a = \overline{B}_a + C$
	060	$\overline{B}_b = \overline{B}_b + C$
	006	$\overline{B}_c = \overline{B}_c + C$
	700	$\overline{B}_a = \overline{B}_a - C$
	070	$\overline{B}_b = \overline{B}_b - C$
	007	$\overline{B}_c = \overline{B}_c - C$

Although there are infinitely many terms, we need only consider a finite number of them to get the result to an assigned accuracy. Suppose, for example, we decide to neglect all terms less than 10^{-5} in magnitude.

Sums of this kind can be treated as loops because every term can be found from the preceding term by successive applications of some simple rules. As in every loop, we must consider the four steps: initialization, execution, modification, and testing.

The initialization operation consists of two parts. We will set aside one storage location in which to accumulate the sum, and initially this must be cleared to zero. Then we must set up the first term. In the execution part, we merely add this term to the sum. Modification involves constructing the next term from the preceding one. Then we test its magnitude to see if we can terminate the loop.

Obviously, the most difficult part of this program will be the modification. Let us see how a programmer might analyze the situation.

Compare the fourth term T_4 with the third T_3. We note that

$$T_4 = 5T_3 \cdot \frac{5}{6} \cdot \frac{x^2}{7}$$

Similarly

$$T_5 = 7T_4 \cdot \frac{7}{8} \cdot \frac{x^2}{9}$$

In general

$$T_{j+1} = (2j - 1)\, T_j \frac{2j - 1}{2j} \frac{x^2}{2j + 1}$$

This is the rule for finding T_{j+1} from T_j.

We could keep count of j and compute $(2j - 1)$, etc., each time; but there is a simpler procedure. The third term is

$$\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{x^5}{5}$$

Suppose we store

$$V = \frac{1}{2} \cdot \frac{3}{4} \cdot x^5$$

and

$$W = 5$$

Then the third term is V/W. We can also use V and W to build up the fourth term:

$$T_4 = \frac{1}{2} \cdot \frac{3}{4} \cdot x^5 \cdot x^2 \cdot \frac{5}{6} \cdot \frac{1}{7}$$

$$= V \cdot x^2 \cdot \frac{W}{W + 1} \cdot \frac{1}{W + 2}$$

This suggests that we construct a new V and W from the old and use them to find T:

$$V_{j+1} = V_j \cdot x^2 \cdot \frac{W_j}{W_{j+1}}$$

$$W_{j+1} = W_j + 2$$

$$T_{j+1} = \frac{V_{j+1}}{W_{j+1}}$$

We can easily verify that the fifth term will be properly computed by this scheme.

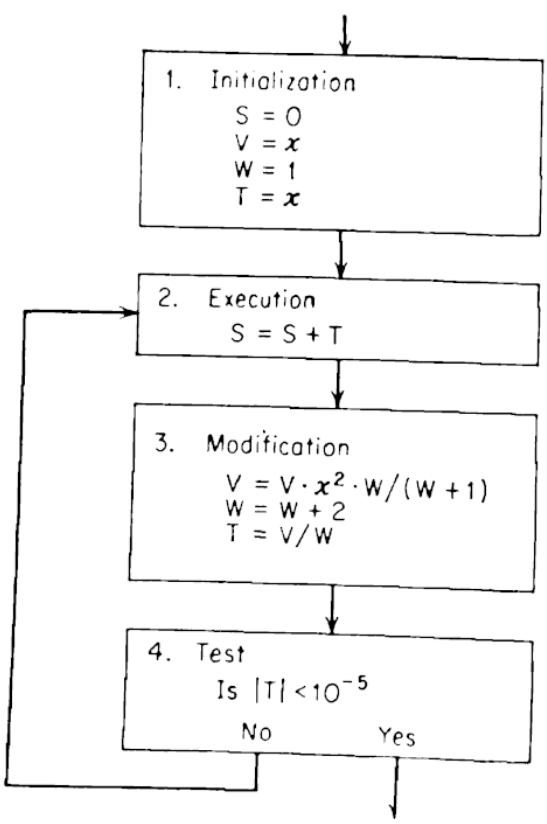


FIG. 13

Since there is no need to retain all the values of V, W, and T, we can assign one storage location to each as well as one to the sum. The flow diagram is given in Fig. 13.

Putting this in the Bell code, we assign

x to 300
S to 301
V to 302
W to 303
T to 304
1 to 305

Then the program becomes

```
100:  2 301 301 301   S  = 0
101:  9 001 300 302   V  = x
102:  9 001 305 303   W  = 1
103:  9 001 300 304   T  = x
104:  1 301 304 301   S  = S + T
105:  3 302 300 000   p  = V · x
106:  3 000 300 000   p  = p · x
107:  3 000 303 302   V  = p · W
108:  1 303 305 303   W  = W + 1
109:  4 302 303 302   V  = V/W
110:  1 303 305 303   W  = W + 1
111:  4 302 303 304   T  = V/W
112:  0 202 045 104   TEST on exponent
113:  NEXT INSTRUCTION
```

We have not put in any READ or PUNCH instructions.

3-14. Loops within Loops; Constructing a Table with Three Parameters

We frequently encounter flow diagrams that contain loops "nested" one inside the other. As a simple example, consider building up a table of

$$c = (a^2 + b^2 - 2ab \cos \theta)^{1/2}$$

Let a vary from 1 to 5 in intervals of 0.25, b from 5 to 10 in intervals of 0.5, and θ from 10° to 30° in intervals of 1°. Since there are 17 values of a, 11 values of b, and 21 values of θ, and since we want all combinations, there will be $17 \times 11 \times 21$ entries in our table.

We can calculate all values systematically by keeping a and b constant and equal to 1 and 5, respectively, and computing c for all values of θ. Then we can increase b to 5.5 and again go through all values

of θ. By increasing b again and again, each time using all values of θ, we finally have used all combinations of b and θ for the first value of a. The complete range of b and θ must then be used for the next value of a; and so on. We can therefore construct a flow diagram that consists of three nested loops: in the innermost loop θ increases systematically, in the next b, and in the outermost a (Fig. 14).

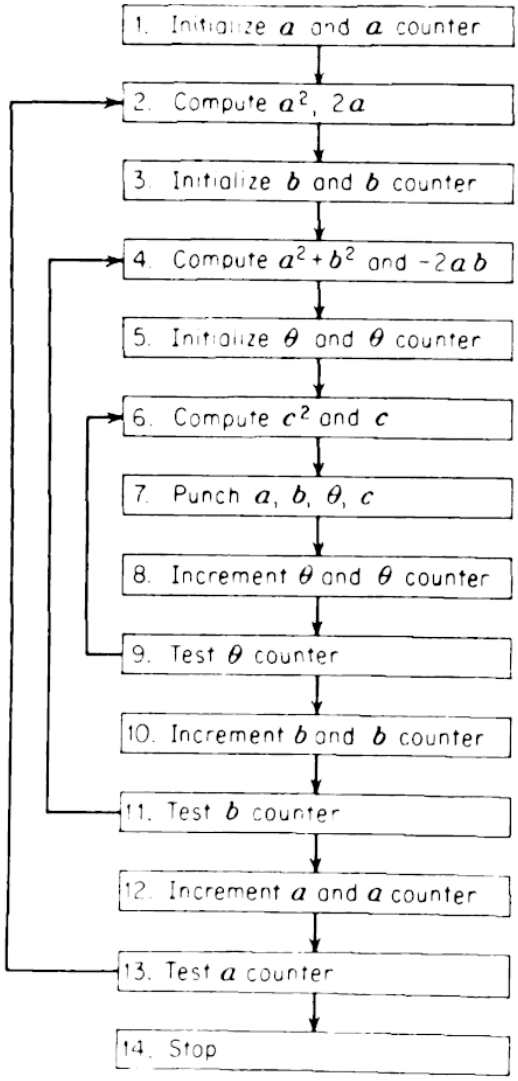


FIG. 14

In this scheme we will set aside three storage locations, one to count the number of times we have executed each loop: we will call these the a counter, b counter, and θ counter. Part of the initialization of each loop will be to clear the appropriate counter to zero. In addition we will set up the initial value of the appropriate parameter: a, b, and θ.

In the incrementing operation we will increase the counter by 1 and increase the value of the corresponding parameter by 0.25, 0.5, or 1 depending upon whether we are incrementing a, b, or θ.

In testing, we will test only the counter. If the loop is not satisfied, we will repeat it; if it is satisfied, we will test the next outer loop.

Suppose we assign storage locations as follows:

301:	a counter	321:	initial value of a
302:	b counter	322:	initial value of b
303:	θ counter	323:	initial value of θ
311:	increment in a	331:	current value of a
312:	increment in b	332:	current value of b
313:	increment in θ	333:	current value of θ
		334:	current value of c

We can then proceed to program as follows:

100:	2 301 301 301	Clear a counter
101:	9 001 321 331	Initialize a
102:	3 331 331 335	Compute a^2
103:	1 331 331 336	Compute $2a$
104:	2 302 302 302	Clear b counter
105:	9 001 322 332	Initialize b
106:	3 332 332 000	Compute b^2
107:	1 335 000 337	Compute $a^2 + b^2$
108:	3 336 332 338	Compute $2ab$
109:	2 303 303 303	Clear θ counter
110:	9 001 323 333	Initialize θ
111:	0 354 333 000	Compute $\cos\theta$
112:	3 000 338 000	Compute $2ab \cos\theta$
113:	2 337 000 000	Compute c^2
114:	0 300 000 334	Compute c
115:	0 410 331 334	PUNCH a, b, θ, c
116:	1 333 313 333	Increment θ
117:	0 006 303 001	Increment θ counter
118:	8 303 021 111	Test θ counter
119:	1 332 312 332	Increment b
120:	0 006 302 001	Increment b counter
121:	8 302 011 106	Test b counter
122:	1 331 311 331	Increment a
123:	0 006 301 001	Increment a counter
124:	8 301 017 102	Test a counter
125:	0 000 000 000	Stop

If the θ loop is not satisfied, we return to 111; if the b loop is not satisfied we return to 106; if the a loop is not satisfied we return to 102. Note that the initialization of both b and θ is part of the a loop and the initialization of θ is part of the b loop. That is, every time we change b we start with the first value of θ all over again; and every time we change a we start with the first value of both b and θ.

Note that this is a program with no READ instruction. All the necessary information, such as the increments and the number of times each loop must be executed, can be read into the machine along with the program deck.

Our preceding programs have usually included READ instructions and the machine has stopped when there were no more cards to be read. In this case, however, we have introduced a new instruction at the very end:

0 000 000 000

to signify the end of the program. That is, the machine automatically stops when both Op 1 and Op 2 are zero.

In this program we have kept count and tested the counter to determine when the loop was satisfied. We could eliminate the counter completely, if we wished, and test the value of a, b, or θ directly. In that case, we would store the maximum values of each of the parameters and subtract them from the current values. If the result is negative, it would indicate that the current value is smaller and the loop should be repeated.

We can illustrate this using the θ loop. Let us store $\theta_{\max} = 30°$ in 343. The testing operation would be carried out with two instructions *before* incrementing θ:

2 333 343 000
0 201 B C

The machine would go back to C if the loop is not satisfied or on to B if it is.

Problems for Chapter 3

The operations necessary to solve these problems are described in the preceding chapters. Before actually testing a program on a machine, however, it would be wise to study some of the testing procedures described in Chap. 5.

1. You are given 80 cards, each containing one number, either positive or negative. Write a program that will do all of the following:
   (*a*) count the number of negative values
   (*b*) count the number of positive values
   (*c*) calculate the sum of all the negative values
   (*d*) calculate the sum of all the positive values
   (*e*) calculate the sum of all the values.
2. Write a program for evaluating

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 2 \cdot 1$$

(of course, the accuracy of this program will be limited by the word size of the computer).

3. Rewrite the loop of Sec. 3-14 using a loop box for the θ loop, a counter for the b loop, and the value of a itself to test the a loop.

4. Build a compound interest table for

$$f = (1 + i)^n$$

where n varies one year at a time from 1 to 50 and i varies from 3 per cent to 5 per cent in intervals of 0.25 per cent. In standard table notation, this can be written $n = 1(1)50$, $i = .03(.0025).05$.

5. Evaluate

$$\tanh^{-1} x = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \cdots$$

assuming $x^2 < 1$ and neglecting terms less than 10^{-5}.

6. Write a program to compute

$$E_1(x) = \frac{e^{-x}}{x}\left[1 - \frac{1}{x} + \frac{2!}{x^2} - \frac{3!}{x^3} + \frac{4!}{x^4} - \cdots\right]$$

for large x. Terminate the loop either when the next term in the brackets is $< 10^{-5}$ or when the terms begin to increase in magnitude, whichever occurs sooner.

7. Given a set of numbers in locations 301 to 400. Write a program that will find the largest and smallest of these in absolute magnitude.

4

Flow Diagrams, Subroutines, and the Program Library

4-1. More about Flow Diagrams

The flow diagrams we have constructed so far have been very simple and, although they did help us see the broad outline of the problem, we could probably have done our programming without them. As problems become more and more complicated, flow diagrams become more useful and, indeed, essential.

Although the amount of detail included in a flow diagram is a matter of personal preference, and some people claim to program without any flow diagram at all, it is a good practice to approach all programming by way of a flow diagram. They are useful at every stage of the game. An accurate flow diagram is the best preparation for a good program. In a complicated situation with many alternative paths the programmer who lays out his plan in a flow diagram and then follows it through is much less likely to make an error in logic. At the testing stage, as we will see in detail in the next chapter, a flow diagram helps in locating errors. Even when the program is working correctly the flow diagram retains its importance. Months or years later, it may become necessary to refer to this problem again, and the flow diagram is much more easily understood after a long interval than the detailed instructions of a program. It is also much easier to use the flow diagram

when explaining the problem to someone else than to struggle through the program itself.

Finally, the flow diagram is more easily adapted to another computer with a different language than the detailed instructions of a program or code. In fact, the flow diagram is so useful after the program has been checked out that even many programmers who do not use flow diagrams in the early stages construct them at this point. But if one is going to draw a flow diagram at any stage, why not begin with one?

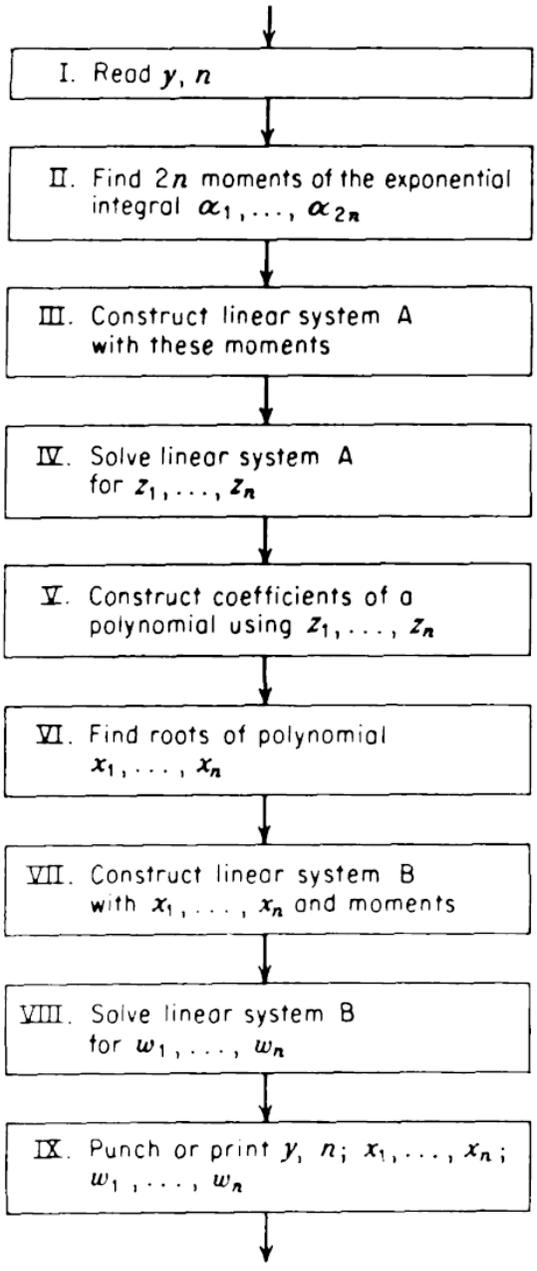


FIG. 15

4-2. Programming in Boxes

Every problem usually falls into some natural subdivisions. Unlike the simple programs we have discussed so far, most problems in practice involve the computation of several different quantities which are then combined to form other functions or subjected to a variety of operations such as interpolation, integration, etc. Each major subdivision becomes a *box* in the flow diagram. In a very complicated problem it is wise to construct several flow diagrams, progressively more detailed. In the first, only the broadest outline of the problem is given; then each box of this flow diagram is further subdivided into boxes; and, perhaps for a few of these boxes, additional details are desirable. It is somewhat like writing an outline for a book. The first step is to decide what each chapter should contain; then each section; and perhaps in a few cases, each paragraph.

Figure 15 illustrates this point. It gives the major steps of a particular problem. The nature of the computation is not of importance, but, for the curious reader, it describes the procedure for finding the points and weights needed to perform a Gaussian type of quadrature, in which the weight function is the exponential integral function.[1]

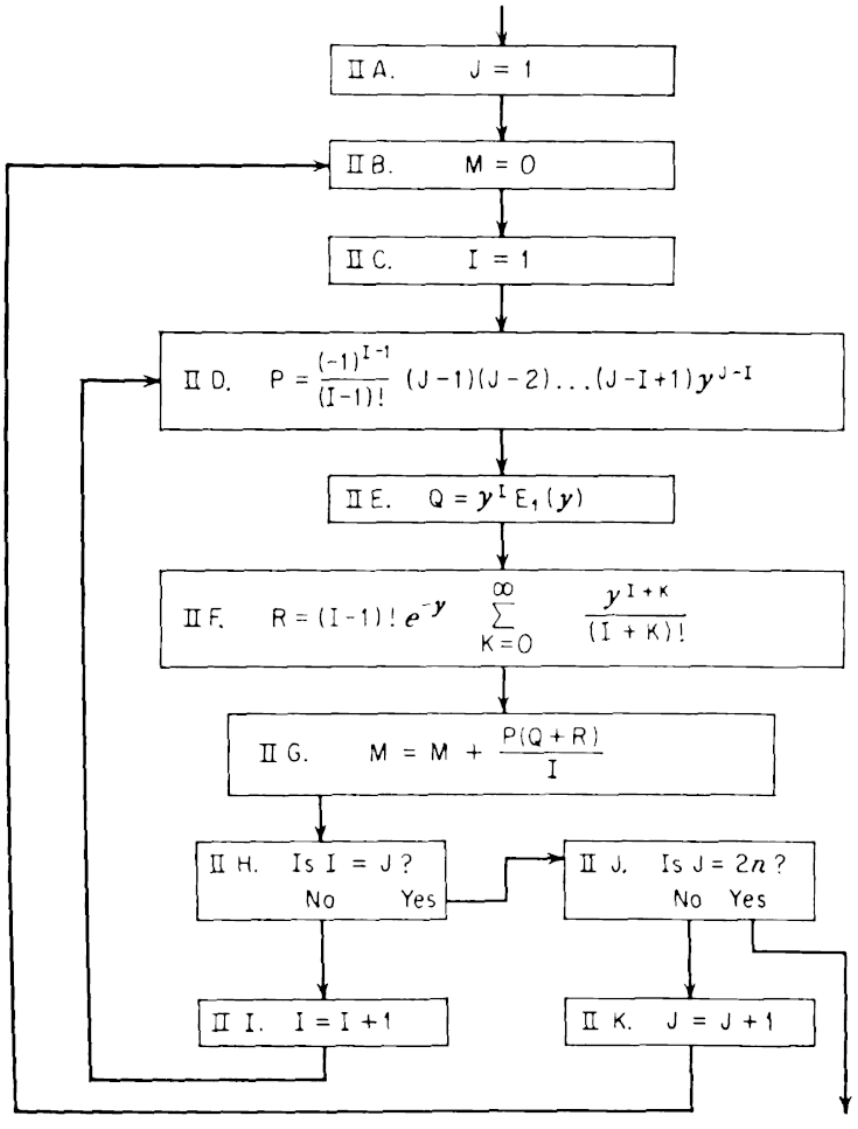


FIG. 16

In Fig. 16, box II is further subdivided. We could subdivide box II F still further to indicate how we would evaluate each term of the infinite series and when we would terminate. Calculations of this kind were discussed in the previous chapter. Of course, the ultimate

[1] See S. Chandrasekhar, "Radiative Transfer", Oxford, 1950, chap. II.

subdivision is in terms of operations the machine can perform, but the flow diagram need not be as detailed as that.

There are several advantages in regarding a problem in terms of boxes. For one thing, as was pointed out earlier, the logical sequence of steps is clearly described and logical errors are considerably reduced. Furthermore, the detailed programming and coding can be done one box at a time. This permits the programmer to proceed systematically through the problem. Each major box can be assigned a region or *block* of locations in the memory. If the problem is very long and complicated each block can be tested individually (see Chap. 5), making it much easier to localize errors and correct them.

One important feature is the ease with which the boxes of a flow diagram can be adapted to somewhat different situations. A box can be modified or replaced without much difficulty, leaving the rest of the program unchanged. In Fig. 15, for example, only box II refers to the specific weight function, in this case the exponential integral. The same program can be used to find points and weights for any other function if the corresponding block in the memory is replaced by a program for finding the $2n$ moments of the corresponding function. Once the moments are numerically known, the rest of the procedure is the same.

It is frequently of interest in research to try the effect of slightly different assumptions. This might require changing only part of a very long computation. To create a program for the modified calculation, it is only necessary to reprogram parts of the problem and a well-constructed flow diagram will indicate which parts.

4-3. Recurrent Boxes; Elementary Subroutines

Suppose our flow diagram looks like Fig. 17. The contents of the blank boxes are not important. The point worth noting is that we must calculate the hyperbolic cosine four times during one run. The program for cosh x offers no difficulty. In fact, it was one of the first examples we tried (Sec. 2-21). We could, of course, copy the complete sequence of four instructions:

$$\begin{aligned} p &= e^x \\ q &= 1 / p \\ r &= p + q \\ \cosh x &= r / 2 \end{aligned}$$

wherever they are needed in the program. This is somewhat inefficient because we are duplicating virtually identical instructions four times. The situation becomes much more serious when the recurrent box contains fifty or one hundred instructions. It would be ridiculous to store these instructions in several different regions of the memory. Yet our previous technique of writing loops to avoid repeating instructions will not work in any simple way here, because the cosh boxes are not located in any simple symmetrical pattern. We must develop a new technique to use in cases like this.

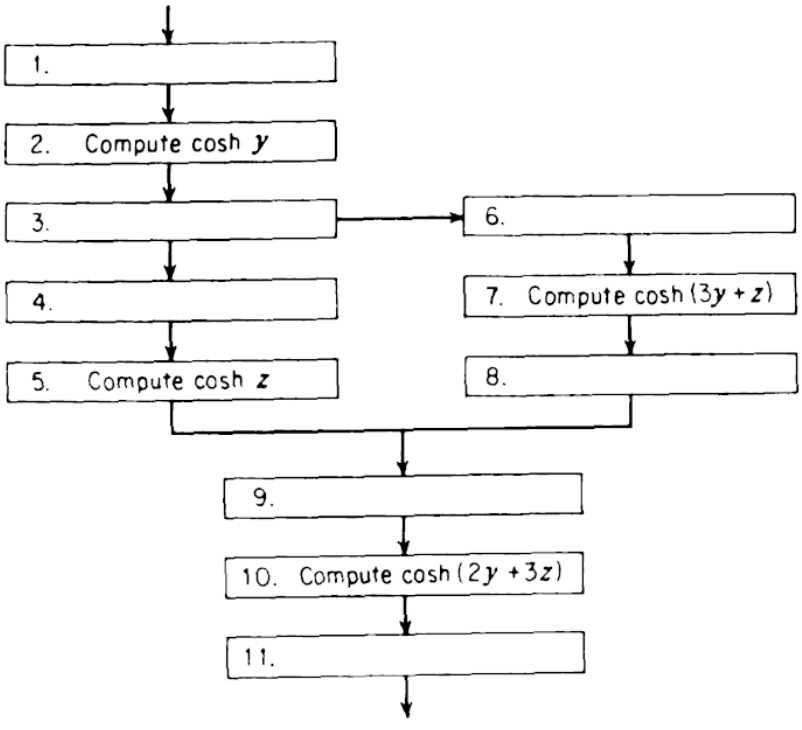


FIG. 17

We will store the orders for finding cosh x only once, but execute them four times. This group of instructions constitutes a *subroutine*. Each time we will "enter the subroutine" with the argument x in 000 and, after the orders are executed, cosh x will be in 000. Of course, the argument will vary: it will be the numerical value of y, z, $(3y + z)$, and $(2y + 3z)$, depending upon where we are in the flow diagram. Nevertheless, if we want the hyperbolic cosine of any number, all we have to do is put that number in 000 and enter the subroutine. When we leave the subroutine, we will have computed the hyperbolic cosine

and stored it in 000. From that point on, it can be used in any way desired.

Note in Fig. 17 that the first time we enter the subroutine it is from box 1; and we go to box 3 when it is completed. On other occasions we enter from box 4 and leave to box 9; or from box 6 and leave to box 8; or from box 9 and leave to box 11. In other words, we transfer to a different place after every execution of the subroutine. This is a general characteristic of the use of subroutines and we must provide a standard procedure for instructing the machine where to find the next instruction after the subroutine is completed.

A subroutine as flexible as this may be worth preserving in the program library. Then, whenever a program involves the calculation of cosh x, we can incorporate the subroutine without having to reprogram it. We will not be in a position to appreciate the versatility of a subroutine library until we have techniques for "translating" subroutines to arbitrary locations in the memory. Those methods will be developed in a later chapter; but we can discuss standard entry and exit procedures at this point.

4-4. Subroutine Entry and Exit

To make the situation more specific, let us suppose that each box of the flow diagram of Fig. 17 occupies a definite block in storage as follows:

Box 1: 215–228
Box 3: 229–243
Box 4: 244–261
Box 6: 262–280
Box 8: 281–297
Box 9: 298–353
Box 11: 354–375

Furthermore, the instructions of the cosh subroutine occupy the locations from 381 to 386.

According to the flow diagram, we want to go from 228 (end of box 1) to 381 and then from 386 back to 229 (beginning of box 3); then from 261 (end of box 4) to 381 and from 386 to 298 (beginning of box 9); etc.

Clearly, the instruction in 386 must be a transfer order, but on different occasions its C address will vary. If we are going to box 3 it will read

386: 0 203 000 229

but if we are to go to box 9 it must read

386: 0 203 000 298

One way to modify this address would be to use a SET C order at the end of box 1:

```
227:  0 005 386 229   SET C part of 386 to 229
228:  0 203 000 381   TRANSFER to 381
```

The instruction in 227 changes the C address of the instruction in 386 to 229. The next instruction transfers to the subroutine. When we reach the instruction in 386 we will transfer to 229.

Of course, we would then need another pair of instructions to go from box 4 to the subroutine and then back to box 9:

```
260:  0 005 386 298   SET C part of 386 to 298
261:  0 203 000 381   TRANSFER to 381
```

Although this procedure would work, it is not as convenient as it might be. For one thing, the programmer must not only keep in mind the location of the first instruction of the subroutine, but also the location of the last instruction of the subroutine, in this case, 381 and 386, respectively. Furthermore it takes two instructions to enter the subroutine each time.

To improve this situation, we adopt a new instruction: TRANSFER to SUBROUTINE: Op 2 = 204. This is the instruction with which we enter a subroutine. Its C address is the location of the first instruction of the subroutine; the B address is the location of the first instruction to be executed following the subroutine. Thus to go from box 1 to the subroutine and from the subroutine to box 3, we use the instruction

228: 0 204 229 381

Similarly, to go from box 4 to the subroutine and from there to box 9, we write

261: 0 204 298 381

Before examining the operations this instruction actually performs in detail, we will introduce two rules to be followed in writing the sub-

routine itself. Suppose that the first instruction of a subroutine is located in α and the last instruction is in β. Then the instruction in α must be

$$\alpha: \quad 0 \ 005 \quad \beta \quad (000)$$

and the instruction in β must be

$$\beta: \quad 0 \ 203 \ 000 \ (000)$$

The instruction in α is SET C, and the one in β is a TRANSFER. The C addresses have been put in parentheses because they will be modified when the program is executed. When used in this way we often refer to the instruction in β as the *link* or the *exit* and we speak of the instruction in α as *storing* or *setting the link*.

In our specific problem $\alpha = 381$ and $\beta = 386$.

Now let us see what

$$0 \ 204 \quad B \quad C$$

accomplishes. First it changes the C part of the instruction in location C to B; then it goes to C for the next instruction. Symbolically, we can represent the operation as

$$\overline{C}_c = B; \quad \text{TRANSFER to C}$$

Consider the specific problem of going from 228 to 381 and from 386 to 229. The sequence of events is this. The instruction in 228,

$$228: \quad 0 \ 204 \ 229 \ 381$$

changes the C address of the instruction in 381 to 229 and then transfers control to 381. The instruction in 381 is now

$$381: \quad 0 \ 005 \ 386 \ 229$$

and when it is executed it changes the C address of the instruction in 386 to 229. The machine then performs the instructions of the subroutine until it comes to

$$386: \quad 0 \ 203 \ 000 \ 229$$

which sends it to 229.

This may seem like a rather complicated business: the order in 228 changes the order in 381 which changes the order in 386. There are, however, certain advantages.

The usefulness of a standard procedure such as this is particularly

apparent when using subroutines from a program library. If every library subroutine is written in the standard way, the exit instruction and the instruction for setting the link are part of the subroutine. Then one can always enter a library routine in the standard way, by using the TRANSFER to SUBROUTINE instruction.

Returning to the flow diagram in Fig. 17 we can accomplish the proper program entries and exits by using the TRANSFER to SUBROUTINE instructions as follows:

	Enter from	Exit to
228: 0 204 229 381	Box 1	Box 3
261: 0 204 298 381	Box 4	Box 9
280: 0 204 281 381	Box 6	Box 8
353: 0 204 354 381	Box 9	Box 11

To be sure the reader understands the operation of the machine, let us follow the flow diagram from the end of box 6 to box 11, looking at the contents of locations 381 and 386 immediately after different instructions are carried out.

Order just executed	Contents of 381	Contents of 386
280: 0 204 281 381	0 005 386 281	0 203 000 ...
381: 0 005 386 281	0 005 386 281	0 203 000 281
.....	.....	.....
386: 0 203 000 281	0 005 386 281	0 203 000 281
281:	0 005 386 281	0 203 000 281
.....	.....	.....
353: 0 204 354 381	0 005 386 354	0 203 000 281
381: 0 005 386 354	0 005 386 354	0 203 000 354
.....	.....	.....
386: 0 203 000 354	0 005 386 354	0 203 000 354
354:	0 005 386 354	0 203 000 354
.....	.....	.....

The instruction in 381 is changed when the TRANSFER to SUBROUTINE instructions in 280 and 353 are executed. The subroutine is entered at 381 and the execution of 381 alters the contents of 386. The first time 386 is executed we go to box 8, the second time to box 11.

4-5. A Typical Subroutine

Let us look now at the subroutine for cosh x, constructed in accordance with the entry and exit rules:

```
381:  0 005 386 (000)   SET link
382:  0 301 000  387    t = e^r
383:  4 388 000  000    p = 1/p
384:  1 387 000  000    p = t + p
385:  4 000 389  000    p = p/2
386:  0 203 000 (000)   TRANSFER (link)
387:  0 000 000  000    Temporary storage
388:  1 000 000  050    CONSTANT
389:  2 000 000  050    CONSTANT
```

Once again, we have used the symbol p for location 000. The two C addresses in parentheses will be modified during execution of the program.

Note that we have explicitly included the constants and also one temporary storage location in the block from 381 to 389. This was done to make the subroutine completely self-contained.

It is usually wise to keep the constants packaged with the subroutine block, particularly in programs that are to become part of the library. When using a subroutine from the library you don't want to have to be bothered with storing additional constants.

Of course, it is inefficient to store the same constant in two or three places when it is used in several different subroutines. (A constant like 1 or π is encountered often.) If storage space is at a premium duplication of constants ought to be avoided; but one must be very careful. If block 3 depends upon a constant stored in block 8, block 11 depends upon a constant stored in block 1, etc., the blocks would be so interdependent that changing one may cause the whole program to fall apart. On the other hand, if each block is self-contained, a new block can be substituted without simultaneously affecting the rest of the program. The decision between compactness and flexibility must be made by the programmer.

Temporary storage locations need not be part of the subroutine block, provided we adopt as a convention that the first ten locations in storage (001 to 010) are to be used exclusively for this purpose. If the programmer avoids putting anything in these locations that has

to be preserved from block to block, this part of the memory can be used by all subroutines. Of course, if more than ten temporary locations are necessary, other provisions must be made for the additional locations. In any event, the description of the subroutine—and an adequate description ***must*** be available for every program in the library —should contain the addresses of any temporary storage locations used outside of the subroutine block itself.

In our example there was only one argument x and one result cosh x, so it was convenient to store them both in 000. If there is more than one argument, or if more than one number is the result of the subroutine, it might be convenient to use the conventional temporary storage block for them.

If 000 is used for storing an argument, that value is lost as soon as an arithmetic operation is performed. In our example it was not necessary to use x after the first step so we did not encounter any difficulty. However, if the argument is needed at several points in the program, it must be stored in the temporary storage block or moved to some other location by using a MOVE instruction as the very first operation following the order that sets the link.

4-6. Generalizing a Subroutine

The most useful subroutines in the library are the ones that will handle the widest variety of situations. A routine for solving a system of precisely five simultaneous linear equations is far less useful than one that will handle any number of equations from two to thirty. This flexibility is somewhat offset by the fact that a general routine of this kind may not be as efficient as one written to accommodate five equations, no more and no less. If an installation is expected to run four equations one day, twenty the next, etc., flexibility is to be preferred. On the other hand, if a large fraction of the time is spent solving systems of a fixed size, it is worthwhile to develop a specific program to deal with that number of equations.

Sometimes with very little additional effort a program can be generalized to include other useful cases. For example, the formula for sinh x is

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

which differs from cosh x only in having a minus sign in the numerator. We can generalize our previous subroutine to compute either one.

Referring to our program in Sec. 4-5, we note that by changing the instruction in 384 to

$$384: \quad 2\ 387\ 000\ 000 \quad p = t - p$$

we will calculate sinh x instead of cosh x. The following program will serve as a general subroutine for both functions:

377:	0 005 386 (000)	SET link for cosh x
378:	9 001 390 384	Change order in 384
379:	0 203 000 382	TRANSFER
380:	0 005 386 (000)	SET link for sinh x
381:	9 001 391 384	Change order in 384
382:	0 301 000 387	$t = e^x$
383:	4 388 000 000	$p = 1\ p$
384:	(0 000 000 000)	$p = t \pm p$
385:	4 000 389 000	$p = p\ 2$
386:	0 203 000 (000)	TRANSFER
387:	0 000 000 000	Temporary storage
388:	1 000 000 050	CONST.
389:	2 000 000 050	CONST.
390:	1 387 000 000	CONST. ($p = t + p$)
391:	2 387 000 000	CONST. ($p = t - p$)

If we enter at 377, the subroutine computes cosh x; if we enter at 380, it computes sinh x. Both versions of the instruction in 384 are stored as constants: one in 390 and the other in 391. Depending upon the entry point, either the addition or subtraction order is placed in 384 by a MOVE instruction. (Note that

$$9\ 001 \quad B \quad 384$$

rather than

$$9\ 000 \quad B \quad 384$$

was used so that x in 000 would be preserved.)

The TRANSFER instruction in 379 was necessary in order to skip over the instructions that set up the sinh x program.

We had to add six memory locations to generalize this program, whereas it would have taken nine to write a separate sinh x subroutine. We have, therefore, not saved very much in the way of storage. In

longer subroutines, however, generalization can be much more efficient than duplication.

4-7. A Complete Example

The following subroutine will calculate the Planck function

$$\pi B_\lambda = \frac{c_1 \lambda^{-5}}{e^{c_2/\lambda T} - 1}$$

where $c_1 = 3.7403 \times 10^{-5}$

$c_2 = 1.43868$

and we must specify λ in cm and T in °Kelvin. If we enter the subroutine with λ in 001 and T in 002 the result will appear in 000. Location 003 is used as temporary storage.

100:	0 005 110 (000)	SET link	
101:	0 352 001 000	$p = \log_{10} \lambda$	
102:	3 111 000 000	$p = (-5) \cdot p$	
103:	0 351 000 000	$p = 10^p$	
104:	3 000 112 003	$t = p \cdot c_1$	
105:	3 001 002 000	$p = \lambda \cdot T$	
106:	4 113 000 000	$p = c_2 / p$	
107:	0 301 000 000	$p = e^p$	
108:	2 000 114 000	$p = p - 1$	
109:	4 003 000 000	$p = t / p$	
110:	0 203 000 (000)	TRANSFER (link)	
111:	−5 000 000 050	CONST. (−5)	
112:	3 740 300 045	CONST. (c_1)	
113:	1 438 680 050	CONST. (c_2)	
114:	1 000 000 050	CONST. (1)	

4-8. Code Words in General Subroutines

In Sec. 3-7 we discussed a program for evaluating the summation

$$c = \sum_{i=1}^{4} a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

This could be easily generalized to a program for computing

$$c = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

So few instructions are involved that it would scarcely pay to make this into a subroutine, but we will use it to illustrate some points that become practical in larger routines.

Suppose that we want a general subroutine to evaluate the sum c, for which n is arbitrary and limited only by the storage capacity of the machine, and the a's and b's are located in arbitrary blocks in the memory. Just before entering the subroutine, we want to specify the numerical value of n and the location of a_1 and b_1, the first values of a and b, respectively. On one occasion, n may be 10, the a's may be located in 401 to 410 and the b's in 547 to 556. Another time, n may be 7, the a's may be in 215 to 221 and the b's in 202 to 208, and so on.

Let us look at the program developed in Sec. 3-7.

```
        2  313  313  313   c = 0
α:     −3  301  305  000   p = a_i · b_i
        1  000  313  313   c = p + c
        0  110  004   α    LOOP
```

You will recall, the sum was accumulated in 313, a_1 is in 301, b_1 is in 305, and $n = 4$. Symbolically, we can write

```
        2   c    c    c
α:     −3  a_1  b_1  000
        1  000   c    c
β:      0  110   n    α
```

To set up a program for dealing with any specific case, we must put the addresses a_1 and b_1 and the value of n (in fixed point) into the proper instructions. This can be accomplished by using SET A and SET B instructions. For example, we could require that before entering the subroutine we must give three orders as follows:

```
0  500   α    a_1   SET A part of α to a_1
0  050   α    b_1   SET B part of α to b_1
0  050   β    n     SET B part of β to n
```

This makes it the responsibility of the person using the subroutine to set things up properly in advance.

Alternatively, we can make it the responsibility of the programmer

who *writes* the subroutine to create a program that sets things up automatically; the one who *uses* the subroutine has to supply only the minimum basic data: namely, a_1, b_1, and n. As we shall see, this requires a bit of manipulation within the subroutine, but it has the advantage of making the subroutine as easy as possible to use.

Before entering the subroutine, the programmer must set up the following *code words* in locations 001 to 003:

001: 0 000 000 a_1
002: 0 000 000 b_1
003: 0 000 000 n

He then enters with the standard TRANSFER to SUBROUTINE instruction and the subroutine takes over from there.

The subroutine begins, as usual, with an instruction that sets the link

0 005 γ (000)

It then proceeds to construct the instructions necessary to calculate the summation. If we had an instruction that would replace the A part of the instruction in α by the C part of the instruction in 001 we could use it here to advantage. But since there is no instruction of this kind, we must resort to more devious means.

We use two instructions to convert the A and B parts of 001 to an instruction for modifying the A address of α:

0 500 001 500 SET A part of 001 to 500
0 050 001 α SET B part of 001 to α

When these instructions have been executed, location 001 contains

001: 500 α a_1

The next two instructions convert 002 to an instruction to modify the B address of α:

0 500 002 050
0 050 002 α

Finally, we use two more instructions to transform 003 into an instruction for modifying the B address of β:

0 500 003 050
0 050 003 β

Then we transfer to 001 and execute 001, 002, and 003. These instructions modify α and β to the proper form necessary to compute the sum. We then transfer to the instruction that clears the sum location and go on from there.

The complete subroutine can be written as follows, beginning in 111:

111:	0	005	124	(000)	SET link
112:	0	500	001	500	Modify instruction in 001
113:	0	050	001	121	
114:	0	500	002	050	Modify instruction in 002
115:	0	050	002	121	
116:	0	500	003	050	Modify instruction in 003
117:	0	050	003	123	
118:	9	001	125	004	MOVE
119:	0	203	000	001	TRANSFER to 001
120:	2	004	004	004	Clear c
121:	−3	(000)	(000)	000	$p = a_i \cdot b_i$
122:	1	000	004	004	$c = p + c$
123:	0	110	(000)	121	LOOP
124:	0	203	000	(000)	TRANSFER (link)
125:	0	203	000	120	CONST.

At the end of the program the sum is in 004 and 000.

Let us consider a specific case. Suppose that we want to enter the subroutine from 253 and go to 254 when we leave; $n = 10$, a_1 is in 351, and b_1 is in 417.

We proceed as follows:

247:	2	001	001	001	Clear 001, 002, 003
248:	2	002	002	002	
249:	2	003	003	003	
250:	0	005	001	351	SET $\overline{001}_c = 351$
251:	0	005	002	417	SET $\overline{002}_c = 417$
252:	0	005	003	010	SET $\overline{003}_c = 010$
253:	0	204	254	111	TRANSFER to SUBR.
254:	.	...	...	...	

where, as before, $\overline{001}_c$ means "the C part of the contents of 001".

We will examine the step-by-step action of the program by writing the instruction executed on the left and the changes it causes on the right.

	Executed				Result				
247:	2	001	001	001	001:	0	000	000	000
248:	2	002	002	002	002:	0	000	000	000
249:	2	003	003	003	003:	0	000	000	000
250:	0	005	001	351	001:	0	000	000	351
251:	0	005	002	417	002:	0	000	000	417
252:	0	005	003	010	003:	0	000	000	010
253:	0	204	254	111	111:	0	005	124	254
111:	0	005	124	254	124:	0	203	000	254
112:	0	500	001	500	001:	0	500	000	351
113:	0	050	001	121	001:	0	500	121	351
114:	0	500	002	050	002:	0	050	000	417
115:	0	050	002	121	002:	0	050	121	417
116:	0	500	003	050	003:	0	050	000	010
117:	0	050	003	123	003:	0	050	123	010
118:	9	001	125	004	004:	0	203	000	120
119:	0	203	000	001					
001:	0	500	121	351	121:	−3	351	(000)	000
002:	0	050	121	417	121:	−3	351	417	000
003:	0	050	123	010	123:	0	110	010	121
004:	0	203	000	120					
120:	2	004	004	004	004:	0	000	000	000
121:	−3	351	417	000					
122:	1	000	004	004	} LOOP				
123:	0	110	010	121					
...	.	...	...	...					
124:	0	203	000	254	TRANSFER (link)				
254:	.	...	...	...					

Note that if we had not first cleared 001 to 003 to zero, we would not be sure Op 1 = 0 in each of these locations.

Location 004 is used for two purposes. It contains the instruction for transferring back to 120 and later on it is used to build up the sum.

As mentioned before, this particular example is a case of the tail wagging the dog. Many more instructions were needed to manipulate the addresses than were used in the computation. But, inevitably, generalization leads to a lot of address modification and correspondingly longer subroutines.

4-9. Program Write-ups

When a program is to become part of a library, an adequately written description is absolutely necessary if it is to be of continuing usefulness. Although most programmers give lip service to this necessity, and rant and rave when others provide them with slipshod write-ups, very few of them will take the time and trouble to describe their own programs adequately.

The usual excuse is the pressure of other work. Once a program has been checked out it is finished as far as they are concerned—on to new and more exciting problems! Of course, no one will pretend that writing a program description is exciting; and it takes time to do a good job. We can only emphasize that the time is worthwhile and without spending that time much of the previous programming effort will be wasted.

It is certainly not sufficient to rely on asking the programmer personally how a problem is to be run. Considering the rate of turnover in computing installations, it is quite likely that the original programmer has moved on to greener pastures. Furthermore, if trouble arises in running the program, it is not at all certain that the original programmer will remember why he used a particular trick when he programmed the problem a year or two ago.

Finally, if there is to be meaningful interchange of library programs between installations, decks of punched cards or reels of tape are not sufficient—a complete description must be provided as well.

A complete program description consists of four parts:

Part one—the information necessary to use the routine. This includes the setting of the switches on the console; the form in which data are to be recorded on cards or tapes; whether the arithmetic is fixed or floating; the number of storage locations occupied; the form of the output; etc. This is the part most frequently referred to and it should be concisely written. Some groups have adopted a standard form for this part so that such things as the switch settings can be seen at a glance.

Part two—the flow diagram and a description of the computing method or algorithm. It is not sufficient to be told that a program will "integrate differential equations" or "solve linear systems". We must also know by what method. This part is particularly useful if

the program is to be modified slightly, or if unexpected errors turn up.

Part three—a complete listing of the program. At least one complete print-out of the program should be available.

Part four—test programs. It is very convenient to have a test example available. For someone unfamiliar with the operation of a program, a test example is a simple way to be sure the instructions are being correctly followed. Furthermore, if machine error is suspected, a sample problem can be used to check the operation.

The parts have been listed in order of priority. A complete listing and test problems are not essential but are useful. Without part one, of course, the program is useless, and part two extends the usefulness considerably.

It is a waste of space to retain programs that do not have write-ups. Programmers who do not mind seeing the results of their efforts relegated to the waste basket are welcome to ignore this section.

4-10. Maintaining a Library

A library is not of much use unless it is easily accessible. In addition to the problem of adequate description, there is also a lack of communication between the person who originates the problem and the library which may contain a program to solve it. Part of the difficulty is in classifying programs so that the programmer can tell precisely what each routine can do. The situation is not improved by programmers who assign cryptic titles to their routines that give no indication whatever of their use.

At Indiana University, we have found that it is very helpful to assign the task of part-time librarian to an experienced programmer. The librarian is responsible for maintaining an index of the available routines and for exchanging programs with other installations. It is also part of the job to be acquainted with the function of most of the programs in the library and the librarian can often come to the rescue if the classification of a subroutine is vague or ambiguous.

Problems for Chapter 4

The operations necessary to solve these problems are described in the preceding chapters. Before actually testing a program on a machine, how-

ever, it would be wise to study some of the testing procedures described in Chap. 5.

1. Use the subroutine for sinh x and cosh x to construct a program for calculating

$$f = \frac{\sinh (x + y)}{\cosh x \cosh y}$$

2. (*a*) Write the program for $\tanh^{-1} x$ (Prob. 5, Chap. 3) in the form of a subroutine.

(*b*) Use this subroutine to construct a table of $\log_{10} (\tanh^{-1} x)$ with x varying from 0.01 to 0.99 in intervals of 0.01. In standard table notation, $x = .01(.01).99$.

3. Stirling's formula may be used to approximate $n!$ for large n:

$$n! \approx \sqrt{2n\pi}\ (n\ e)^n$$

Write a subroutine that is entered with n in 000 and replaces it with $n!$ evaluated by this method.

4. The relations between rectangular coordinates (x,y,z) and spherical coordinates (r,θ,φ) are

$$x = r \sin \theta \cos \varphi$$

$$y = r \sin \theta \sin \varphi$$

$$z = r \cos \theta$$

Given r, θ, and φ in 001, 002, and 003, respectively, construct a subroutine that will replace them by x, y, and z. (Assume θ and φ are expressed in radians.)

5. The relation between energy E, rest mass m, and momentum p, according to special relativity, is

$$E = mc^2 \left[\left(1 + \frac{p^2}{m^2 c^2} \right)^{1/2} - 1 \right]$$

where $c = 2.99791 \times 10^{10}$ cm/sec (the velocity of light). Write a subroutine that is entered with m and p in 001 and 002, respectively, and places E in 000.

6. A set of n numbers is stored in a block from a_1 to a_n. Both n and a_1 are arbitrary. Write a subroutine which will put the largest of these (in absolute value) in location 000.

5

Testing Programs

5-1. General Comments

Very few programs work correctly the first time they are tried on a machine. In some cases the machine will stop dead trying to execute an instruction that is not in its vocabulary; or a programming error in logic will send it into an infinitely repeating loop; or if it does get as far as producing results, they are so fantastically large or small that they are clearly unreasonable. At times like these, when the machine has its moment of triumph over man, the programmer demonstrates the range and versatility of his vocabulary, and minors had best be excluded from the room.

Knowing how to track down the errors in a program is fully as important as writing the program to begin with. One must simply expect to make errors and it is imperative to approach the machine prepared to detect them and eliminate them. The efficient use of machines in "code checking" or "debugging" problems requires careful preparation in advance.

If the machine stops, or loops, or produces nonsense, the programmer should not stare at it dumbfounded; nor should he become panic-stricken and start pushing buttons on the console. (Above all, he should not call for the repairman, since it's unlikely the machine is at fault.) Instead he should try to decide which of the methods outlined in this chapter are most likely to help straighten things out.

It should be emphasized that one does not generally locate errors

then and there at the machine. You are under pressure and you have a limited time. The next fellow is already pacing back and forth, waiting his turn and giving you dirty looks. Every second counts. But by using the proper instructions you can make the machine produce a record containing all the information you need to locate the errors; and by studying this record in the less frenzied atmosphere of your office you are more likely to find the errors.

The best programmer is not only the one who makes the fewest mistakes but also the one who uses the least machine time in locating errors when they do occur.

Some debugging methods involve inserting one or two instructions. Others, however, require the use of small "debugging programs" that have to be stored in the memory. It is a wise precaution to set aside a section of the memory for whatever testing programs may prove to be necessary. If you avoid using this block of locations in normal programming it will be available when needed during testing. Some installations have standardized their debugging programs so that they always occupy the same region of the memory, frequently the last 50 or 100 addresses.

5-2. Checking in Blocks; Sample Calculations

Large, complex programs should be checked in sections before being put together. A good flow diagram is very helpful in this respect. First one box is completely checked; then another is attached and checked; and so on. In this way one is sure of the result up to a point and errors are localized in the part that is newly attached. Beginners would do well to code check blocks of only about one hundred instructions at a time.

For testing purposes, sample calculations must be carried out by hand. Simple data can be used, but not so simple that they are not an adequate check. For example, the test value $x = 0$ is not very informative in testing a polynomial program. Both positive and negative numbers should be used whenever appropriate, and they ought to cover the expected range in magnitude. In this way you might uncover attempts to take square roots or logarithms of negative numbers, or situations in which the floating-decimal number range is exceeded.

The same general method of calculation should be used in the hand

computation as is used by the machine wherever feasible, so that you know not only the final answer but intermediate results as well. This is important because it is necessary to know precisely where in the program the machine result began to deviate from the correct answer. For example, if a polynomial is evaluated by the machine in the form

$$f(x) = (((x + a_1)x + a_2)x + a_3)x + a_4$$

that is the way the hand calculation ought to be carried out, too.

Care in selecting test examples will avoid the embarrassment of unexpected errors in production runs.

5-3. Memory Dumps

The crudest procedure for locating an error is to "dump the memory"—that is, to make a listing of the contents of every storage location. Presumably by examining this listing and comparing what is in a location with what *ought* to be there, it is possible to locate errors.

In the Bell code, we can accomplish a memory dump with two instructions:

```
998:  0 410 000 997
999:  0 000 000 000
```

Both of these can be punched on one card. This card is followed by a transfer card starting the program in 998 and both cards are loaded in the standard way. The first instruction punches the contents of locations 000 to 997; then the instruction in 999 causes the machine to stop.

The one advantage of a memory dump is that it can be carried out by a machine operator who knows nothing at all about the program. In some installations it is standard procedure to dump the memory if there is any kind of program failure during testing or production. It is quick and it does give some information, but there are better techniques.

The principal failing of the memory dump is that it gives the status of all memory locations at one instant; but it does not tell how they got that way. It is a static view of the problem rather than a dynamic one.

Furthermore, it contains a lot of irrelevant data. It indiscriminately

produces a listing of the entire memory, even though the activity of the program may be restricted to one small block.

When all else fails, a memory dump is the quickest way to get at least a hint of what went wrong.

5-4. Snapshots

It is more satisfactory to take several views of appropriate parts of the memory at different times during a computation. This can be accomplished by liberally interspersing PUNCH instructions in the program. It is descriptive to call these print-outs "snapshots".

For example, part of the calculation may involve the computation of functions l, m, and n. These are not final answers in themselves but we want to be sure they have been computed correctly. If they are stored in 201, 202, and 203, respectively, we may put one PUNCH instruction in the program immediately following the computation of the last of them:

0 410 201 203

At some later point in the program it might be desirable to check that the instructions in 308 to 311 have been correctly modified by the program. At this point we can put the instruction

0 410 308 311

into the program.

It is convenient to print or punch these partial results soon after they are calculated because they also serve as landmarks. If the machine stops, owing to a programming error at some point, this output will indicate, for example, that the error occurred *after* the computation of l, m, and n, but *before* the computation of q, r, s, and t.

Consider the following example:

$$f = \frac{(2x + 3\cos\theta)(y + 2\cos\theta)}{(x + 3\cos\theta)(2y + 2\cos\theta)}$$

080:	0 400 101 103	READ x, y, θ
081:	0 304 103 201	$k = \cos\theta$
082:	1 000 201 202	$m = 2\cos\theta = k + k$
083:	1 000 201 203	$n = 3\cos\theta = m + k$
084:	0 410 201 203	PUNCH k, m, n
085:	1 101 203 204	$q = x + n$

086: 1 101 204 205 $r = x + q$
087: 1 102 201 206 $s = y + m$
088: 1 102 206 207 $t = y + s$
089: 0 410 204 207 PUNCH q, r, s, t
090: 3 205 206 208 $u = r \cdot s$
091: 3 204 207 209 $v = q \cdot t$
092: 0 410 208 209 PUNCH u, v
093: 4 208 209 104 $f = u\ v$
094: 0 410 101 104 PUNCH x, y, θ, f
095: 0 203 000 080 TRANSFER

As you will note, the instruction in 087 is in error: the code gives $y + k$ instead of $y + m$ as required by the program. This will show up when q, r, s, and t are punched by the instruction in 089.

The example is an exaggeration because *every* partial result has been punched. In practice it might not be necessary to be quite so cautious.

The PUNCH instructions in 084, 089, and 092 are needed only during the testing process. Once the program is working correctly, they are superfluous. In fact, since punching (or printing) is slow relative to computing time, it would be very inefficient to leave these instructions in the program.

If we replace them by zeros,

0 000 000 000

the machine would stop. An instruction with both Op 1 and Op 2 equal to zero is usually used only at the end of a program.

We can, however, substitute the operation Op 2 = 454. This instruction is called "No operation", or "No op". Regardless of the B or C address, this instruction is ignored and the machine proceeds to the next instruction. To convert the program we have just written from a test program to a production program, we merely load the instructions

084: 0 454 000 000
089: 0 454 000 000
092: 0 454 000 000

The last card is followed by a transfer card that begins the program in 080.

If for any reason it is necessary to restore the testing instructions, they must be reloaded:

084: 0 410 201 203
089: 0 410 204 207
092: 0 410 208 209

It is very much easier to write the program originally with locations reserved for snapshots than to have to squeeze them in afterward.

5-5. Inserting Instructions

If we discover that an instruction has been omitted in the program or if we wish to insert some test instructions, we must break the sequential flow, transfer to another part of the memory to execute the new instructions, and then return to the main program.

For example, suppose that the program contains the instructions

075: 1 402 403 504
076: 4 503 505 506

and we want to insert the instruction

3 401 403 505

between them. Rather than assign the new instruction to 076, which would mean moving the instruction that was in 076 to 077 and moving all subsequent instructions one location higher, we replace the instruction in 075 by a TRANSFER to an unused portion of the memory:

075: 0 203 000 891

The first instruction in the new region must be the order that was previously in 075. The next instruction is the new one and the following instruction is again a TRANSFER back to 076:

891: 1 402 403 504
892: 3 401 403 505
893: 0 203 000 076

This procedure may be used to insert any number of instructions. To introduce 6 new instructions between the order in 115 and the one in 116 we proceed as before. We move the order from 115 to a new region and put a TRANSFER order in its place. The first instruction in the new region is the one that was in 115. We follow this with the 6 new instructions. Finally we end with a TRANSFER back to 116.

It thus takes 8 new locations to interpolate 6 new instructions and, in general, it takes $n + 2$ locations to insert n new instructions.

5-6. Tracing

When a program is traced, the machine produces a record of each instruction *as it is performed*, including the data used and the results obtained. It provides a *dynamic* picture of what is happening in great detail.

When using the Bell method on a basic IBM 650 it is necessary to load a small deck of cards, called the "trace deck", to set the machine up for tracing. In that case, all *normal* PUNCH instructions are suppressed. Instead, a card is punched before each instruction is executed, containing the following information:

Cols. 7–9	location of instruction
10	6 (because there are six words on this card)
11–21	original instruction
22–32	instruction as modified for execution
33–43	contents of loop box
44–54	$\overline{A}$ if A $\neq$ 000, zero if A = 000
55–65	$\overline{B}$
66–76	contents of 000 before instruction is executed
77–79	problem number

Note that this card contains the contents of 000 *before* the instruction was executed. The contents of 000 *after* the instruction is executed will appear on the next card. As usual, a transfer card begins the program. To return operation to the "punch mode" another small deck must be loaded.

The trace output of the polynomial program of Sec. 3-5 is given in Table 5, using the data $x = 2$, $a_1 = -1$, $a_2 = 5$, $a_3 = 3$, $a_4 = 1$. Irrelevant information has been indicated by dots.

Tracing is an important technique in testing programs, but it must be used in moderation. Output is a relatively slow operation, and if the machine must produce a card, or print a line, for every order executed, it will be going at a snail's pace. Furthermore in a long program, with many loops, several thousand instructions may be performed and an indiscriminate trace will give an enormous amount of output, only a small part of which may be needed.

Table 5

Location	Original instruction	Modified instruction	Loop box	$\overline{A}$ if A ≠ 000 Zero if A = 000	$\overline{B}$	$\overline{000}$
100	0 400 201 201	0 400 201 201	0 000 000 000	. . . . .	. . . . .	. . . . .
101	4 201 201 000	4 201 201 000	0 000 000 000	2 000 000 050	2 000 000 050	. . . . .
102	3 000 201 000	3 000 201 000	0 000 000 000	0 000 000 000	2 000 000 050	1 000 000 050
103	−1 000 301 202	1 000 301 202	0 000 000 000	0 000 000 000	−1 000 000 050	2 000 000 050
104	0 010 004 102	0 010 004 102	0 000 000 000	. . . . .	. . . . .	1 000 000 050
102	3 000 201 000	3 000 201 000	0 000 001 000	0 000 000 000	2 000 000 050	1 000 000 050
103	−1 000 301 202	1 000 302 202	0 000 001 000	0 000 000 000	5 000 000 050	2 000 000 050
104	0 010 004 102	0 010 004 102	0 000 001 000	. . . . .	. . . . .	7 000 000 050
102	3 000 201 000	3 000 201 000	0 000 002 000	0 000 000 000	2 000 000 050	7 000 000 050
103	−1 000 301 202	1 000 303 202	0 000 002 000	0 000 000 000	3 000 000 050	1 400 000 051
104	0 010 004 102	0 010 004 102	0 000 002 000	. . . . .	. . . . .	1 700 000 051
102	3 000 201 000	3 000 201 000	0 000 003 000	0 000 000 000	2 000 000 050	1 700 000 051
103	−1 000 301 202	1 000 304 202	0 000 003 000	0 000 000 000	1 000 000 050	3 400 000 051
104	0 010 004 102	0 010 004 102	0 000 003 000	. . . . .	. . . . .	3 500 000 051
105	0 410 201 202	0 410 201 202	0 000 000 000	. . . . .	. . . . .	. . . . .
106	0 203 000 100	0 203 000 100	0 000 000 000	. . . . .	. . . . .	. . . . .

For these reasons, *selective* tracing is to be preferred. By using a few new instructions it is possible to select a small portion of the program for tracing, while the rest of the program is executed at its usual speed.

To make the best use of a selective trace you must have some idea of the general region of the program likely to be in error. It may be that the logic in part of the computation is particularly complex and you suspect the error may lie there; or every part of a program but one may have been checked out on another occasion and you are sure the error is in the new section; or a few "snapshots" have indicated that all was correct up to a point and not beyond. When the area to be traced has been selected, it must be bracketed by instructions that begin and end the tracing operation.

In the Bell code, the instruction Op 2 = 450 starts the trace and Op 2 = 451 stops it. The B and C addresses of these instructions are ignored. When the machine is operating in the "punch mode" neither instruction has any effect.

When the "trace deck" has been loaded the machine automatically starts tracing with the first instruction executed. If the part of the program we want to trace lies further along, we must begin the program with the instruction

```
0 451 000 000   Stop trace
```

Immediately preceding the first instruction to be traced we put in the order

```
0 450 000 000   Start trace
```

Following the last instruction to be traced we place the instruction

```
0 451 000 000   Stop trace
```

Either we can put these instructions into the program when it is first written, in regions where we expect trouble, or they can be inserted by the techniques of Sec. 5-5.

Consider the flow diagram of Fig. 12. If we want to trace only the branch related to imaginary roots (boxes 4b, 5b, and 6b) we can modify the program as follows. We will assume a "Stop trace" order was given at the very beginning of the program.

Then we introduce new instructions as follows:

```
116:  0 203 000 991   TRANSFER
991:  0 450 000 000   Start trace
```

```
992:  0 350 407 000 }
993:  0 203 000 117 }  box 4b
117:  0 300 000 000 }
118:  4 000 405 408 }
119:  9 001 408 402   box 5b
120:  0 203 000 994   TRANSFER
994:  2 306 408 404   box 6b
995:  0 451 000 000   Stop trace
996:  0 203 000 121   TRANSFER
121:  NEXT INSTRUCTION
```

Special techniques for tracing loops will be discussed in Sec. 5-11.

5-7. Stop Codes

There are several reasons we might want the machine to stop and several ways we might cause it to do so.

Suppose that, by mistake, control is transferred during operation to a part of the memory that should not be used in the problem at hand. If there are valid instructions in that part of the memory, the machine will go blithely on executing them. We would prefer that all computations halt, however, as soon as the program strays beyond the intended bounds. Remembering that the instruction

0 000 000 000

causes the machine to stop, we can "clear the memory to zero" before loading the program. In that way, every location in which no new information is specifically stored will contain a STOP instruction, and as soon as a programming error causes the machine to try to execute one of these orders, it will stop.

In the Bell scheme, a special card, called "Memory reset", loaded immediately before the program, clears the memory. Some types of machines will clear the memory when a single button is pressed.

The "all zeros" STOP instruction is not very flexible because it is difficult to start the computation again once it has been stopped. A more useful stop code is Op 2 = 200, the CONDITIONAL STOP. The instruction

0 200 B C

causes the machine to stop, if a particular switch on the console, called

the Programmed Stop Switch, is in the Stop position. The contents of memory location B will be displayed on the console lights. Computation will resume with the instruction in C if the Start button is pressed; in this respect this is also a form of TRANSFER instruction.

Several things are worth noting about this instruction. First of all, it can be ignored simply by flipping the Programmed Stop Switch to the Run position. The advantage of this is that instructions of this kind can be interspersed in the program for testing purposes and then effectively removed by a switch, without bothering to load any new cards.

Secondly, $\overline{B}$ will be displayed on the console lights, permitting a quick check on the contents of some selected storage location. (We will have more to say later on about the evils of relying on the console lights for program checking, but limited use of this procedure is all right.)

Finally, the program can be resumed after this kind of stop, simply by pressing a button.

CONDITIONAL STOPS are useful in diagnosing errors. For example, we may know that the series expansion used to represent some function is not valid if the argument is greater than or equal to 1. Therefore, before proceeding with the computation, we perform a test to see if the argument is too large. If it is not, we go on; if it is, we stop. We choose the B address so that a code number on the display lights tells us the kind of error encountered; and the C address is the location we want to go to when we resume. A series of instructions of this type is given below:

```
562:  2 000 564 000   TEST
563:  0 201 565 567   TRANSFER on sign
564:  1 000 000 050   CONST.
565:  0 200 566  α    COND. STOP
566:  0 000 000 073   Code word
567:  . ... ... ...   Go on
```

The instruction in 562 subtracts 1 from the argument, which we assume is in 000. If the result is negative, the argument is inside the range and the computation goes on from 567. If the result is positive, we go to 565, which causes a stop. The console lights give the contents of 566, which contains the code word

0 000 000 073

If the machine stops with this number displayed on the console lights, it indicates to the operator that the limit on the argument has been exceeded. When we start again we will go back to α, which presumably gives us an opportunity to correct the situation.

A CONDITIONAL STOP can also be used to keep track of your location in the flow diagram when testing a complex program. Suppose the flow chart consists of fifteen boxes, and the paths between them are quite complex. If a CONDITIONAL STOP is put at the end of each box, the machine will stop before going on. As a further aid, we could store in locations 981 to 995 the numbers

981: 0 000 000 001
982: 0 000 000 002
... .
995: 0 000 000 015

When the instruction

0 200 986 C

is placed at the end of the sixth box, the number

0 000 000 006

will appear on the console lights, indicating that box 6 has been completed. The C address should be the first instruction of the next box.

This instruction can also be used instead of a PUNCH instruction to examine an occasional partial result. In that case, the B address would be the location of the number we want to look at. It is not very efficient, however, to halt computation to read console lights, and punching or printing is usually preferable.

5-8. Comparison Methods

Some errors are caused by incorrect or unintentional modification of instructions or constants. If an error of this type is suspected it would be wise to compare the contents of the memory after the program has been operating a while, with the program as loaded originally.

The "brute-force" method is to dump the memory and compare it with a print-out of the original program.

It would be preferable to do this automatically and to select for

printing only those storage locations that have been modified. This requires a special comparison program that is loaded in either the standard region reserved for "debugging routines" or any other part of the memory not used by the program being tested. (A typical comparison program was written for IBM 650 machine-language programs at the University of Michigan. The author does not know of a similar routine for Bell code programs, although one could easily be written.)

A comparison test is run in several stages. First a standard code word is stored in every memory location. Then the program to be tested is loaded in the normal way and run as far as desired.

Following this, the comparison routine is loaded and the original program is placed in the input unit. Under the control of the comparison routine, the instructions on the input cards (or tape) are examined one at a time. Consider an instruction normally stored in 732. The comparison routine examines the contents of 732 and compares it with the instruction on the original card or tape. If there is no difference, there is no output. If they differ, however, the output unit indicates the location of the instruction, the original version, and the modified version. When the comparison routine is through with a particular storage location it replaces its contents by the standard code word.

After the entire input program is compared in this way, the comparison routine goes through the memory to see if there are any locations that still do not contain the standard code word. For example, a location used for temporary storage would not contain information loaded with the original program, but its contents are changed by the operation of the program so that location will not contain the code word. This information also appears in the output of the comparison routine.

A common error made by beginners is to erase program steps by storing data in the same location. An error of this kind, called "overwriting", is easily detected by using a comparison test.

5-9. Address Search

If it is suspected that there is an erroneous reference to a particular address, it is advisable to try an address search to find every reference to that address in the program. (A typical program of this kind was

also written at the University of Michigan for 650 machine-language programs. Here again it would be simple to write a program that would search Bell code programs.)

To run an address search, we read the search program into the debugging region. The address we are to search for is fed into the input and the search is begun. The search program systematically examines every instruction in the memory to see if the test address is part of it. Every time an instruction is found containing the test address, the search program produces output giving the instruction and its location. Incidentally, the search program cannot distinguish between data and instructions and if a constant or intermediate result in storage accidentally has the right combination of digits in the right place it will appear in the output.

5-10. Testing Programs at the Console

To the uninitiated, the term "electronic computer" often conjures up the view of an enormous console of buttons and flashing lights, at which the operator sits, playing programs on it like Bach fugues on a baroque organ. In practice, however, it is wise to use the console with caution. As we pointed out in the first chapter, the more human intervention there is, the less efficient the operation of a program will be, because the computer must wait while the human operator makes up his mind. Furthermore, there is as much of a chance of *introducing* an error by console manipulation as of finding and correcting one. Think twice before pushing a button or flipping a switch; in the rush of a code check, it is all too easy accidentally to push the button that clears part or all of the machine and erases the program or the results of many hours of calculation.

Having issued this warning, let us go on to see what we can accomplish by using the console intelligently.

Some machines have switches that permit a choice between alternative paths in the program. These are particularly useful in testing because one path may be used to print or punch partial results. By flipping a switch on, we can produce results for checking; in normal operation the switch would be off and the printing or punching instructions would be bypassed. We will have an example of this in the next chapter.

We have already mentioned the Programmed Stop Switch on the

IBM 650. This controlled the operation of the CONDITIONAL STOP instructions in the program. It is sometimes convenient, however, to stop the calculation when the program reaches a particular location in the memory. This can be achieved, on the 650, by setting the Address Selection Switches to the desired address and setting the operation switch to the Address Stop position.

This device can be used to localize the area of the program in error. Suppose the machine stops unexpectedly, and you would like to know how far it actually proceeded correctly. You can begin the computation again and set the Address Selection Switches to the location of a landmark near the beginning of the program. It will stop either when it reaches this point or before. If it successfully reaches this point, you can set the switches for the next milestone and let the computation proceed. Eventually the erroneous instruction will cause a stop before the location on the selection switches is reached and therefore you can tell approximately where the error lies.

At this point one ought to relinquish the machine and think for a while. Then one should use one of the other techniques, such as selective tracing or snapshots to pinpoint the error. To continue to use the Address Stop is wasteful of machine time and very poor technique.

Most machines have a mechanism for sending information from the console directly to a location in the memory. This can be used to correct an occasional erroneous instruction. It is a time-consuming operation, however, and should be avoided if there are more than a few changes necessary. The alternative is to punch the corrections on cards or tape and load these in the standard way. An advantage of using cards or tape is that a record of the change is available. Once again, in making changes at the console it is easy to erase something important accidentally.

5-11. Tracing Loops

The information needed to detect errors in loops can usually be obtained from only one or two repetitions of the loop. Tracing a loop is a slow process and it is of doubtful value to let the trace run on and on through dozens and perhaps hundreds of repetitions of the loop. A special trace instruction in the Bell code has been designed specifically to permit limited tracing of loops. Op 2 = 452 is called

"Start trace and erase". Its function is to start tracing, but before the tracing actually begins, this instruction is replaced by the "No operation" code (Op 2 = 454). The next time this instruction is reached, it has no effect.

The "Start trace and erase" instruction must be used in conjunction with the normal "Start trace" and "Stop trace". To trace a loop only twice, the following scheme is used (Fig. 18).

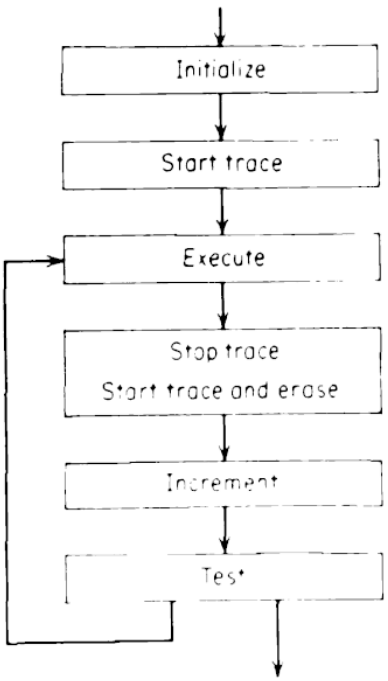


FIG. 18

Following initialization, we start the trace with the normal "Start trace" instruction. After tracing the execution box or boxes we stop the trace; but we start it again immediately by using the "Start trace and erase" instruction. We trace the increment and test boxes and, if the loop is repeated, we continue to trace the execution section a second time. Once again we arrive at the "Stop trace"; but this time the next instruction is "No operation". Therefore, even though the loop may be executed a hundred more times, trace cards are not punched and the operation continues at high speed.

If we desire to start tracing again after the loop has been satisfied, we place a new "Start trace" instruction at the beginning of the next box.

Note that a "Start trace" instruction has no effect if the machine is already tracing; and a "Stop trace" instruction has no effect unless the machine is tracing.

5-12. Testing Programs Containing Subroutines

If a program to be tested contains one or more long subroutines, and if the subroutines have already been tested separately, it is possible to save time when computing a test example by "short-circuiting" the subroutine box. By this we mean that we don't actually execute the subroutine block but we set things up to appear as if we do.

Suppose the program computes θ, then finds the associated Legendre

function $P_2^1(\cos\theta)$ by means of a subroutine, then uses this to find another function z. By working out a test problem by hand, we find that θ turns out to be 50°. By consulting a table we find that $P_2^1(\cos 50°) = 1.47721$. Therefore, instead of executing the subroutine for finding $P_2^1(\cos\theta)$ for a general value of θ, we substitute a few instructions that punch or print θ to be sure it is right and then introduce the known value of $P_2^1(\cos 50°)$.

Suppose the actual subroutine for evaluating the associated Legendre function begins in 451; it is entered with θ in 000 and when it is completed $P_2^1(\cos\theta)$ is in 000. We write a new short subroutine, as follows:

```
451:  0 005 454 (000)   Store link
452:  0 410 000  000    PUNCH
453:  9 000 455  455    Substitute
454:  0 203 000 (000)   TRANSFER
455:  1 477 210  050    CONST.
```

We enter just as in any subroutine; this tests the entry part of the regular program. Then we punch the contents of 000 to make sure θ was 50°. We have stored $P_2^1(\cos 50°)$ in 455 and the order in 453 moves it to 000. (Since the A address already indicated this number was to be moved to 000, we just used the original location, 455, as the C address.) Finally, we leave by the standard TRANSFER instruction.

Once the program is working correctly with the subroutines short-circuited, it should be tested a final time with the subroutines operating normally.

5-13. Testing a Subroutine

In order to test a subroutine, a separate program should be written to simulate all situations likely to be encountered when the subroutine is used.

If the subroutine calculates a function for which tables exist in the literature, the test program might use the subroutine to generate a table for comparison. For example, consider the cosh x subroutine programmed in Sec. 4-5. A testing program might be written that would compute cosh x, where x goes from x_0 to x_f in intervals of h:

```
900:  0 400 910 912   READ x_0, h, x_f
901:  9 000 910 913   MOVE x_0 to 000 and 913
902:  0 204 903 381   TRANSFER to subroutine
903:  9 000 000 914   MOVE cosh x to 914
904:  0 410 913 914   PUNCH x, cosh x
905:  2 913 912 000   Test
906:  0 201 907 908   TRANSFER on sign
907:  0 000 000 000   STOP
908:  1 913 911 913   Increment x
909:  0 204 903 381   TRANSFER to subroutine
```

Or it might be adequate to test the subroutine for several selected values, individually read in

```
900:  0 400 906 906   READ x
901:  9 000 906 906   MOVE x to 000
902:  0 204 903 381   TRANSFER to subroutine
903:  9 000 000 907   MOVE cosh x to 907
904:  0 410 906 907   PUNCH x, cosh x
905:  0 203 000 900   TRANSFER
```

Just as in the case of other programs, PUNCH instructions can be interspersed within the subroutine for testing purposes.

Again we emphasize that test problems should cover the entire range of expected values for all quantities involved. This is particularly true of subroutines intended for the program library.

Problems for Chapter 5

Locate your errors in the problems of the previous chapters by using the techniques described in this one.

6

Automatic Programming

6-1. General Concepts

As we have seen in previous chapters, there's a lot of routine "bookkeeping" involved in the programs we have discussed. Before a problem can be run on a computer it must be expressed in a code the machine can understand and we must assign storage locations to all the variables and intermediate results involved. This is not only a bothersome chore but it is also a procedure subject to frequent human errors.

This unsatisfactory situation developed because it was natural for machine designers to think in terms of the electronic circuits they could economically build, rather than in terms of the people who were to use the computers. So, in the past, the scientist has been presented with machines with languages all their own and it was up to him to figure out how to make them do his computations.

Yet why should a scientist be faced with a new language every time he works with a new machine? Every new computer has its own set of rules concerning what is and what is not a valid instruction and every new language is another obstacle between the scientist and the solution to his problem.

Why not plan machines from the user's viewpoint rather than from that of the machine designer? What sort of language is most desirable for research problems? These questions are being considered

today, and from discussions of this kind a more suitable scientific machine language will probably arise.

Several properties of this language are already apparent. Clearly it would be more convenient if the arithmetic operations were represented by the usual symbols $+$, $-$, $\times$, and $\div$, rather than by numeric codes. Furthermore, the scientist or engineer is also accustomed to using symbols for his variables rather than specific addresses in the memory. He would much prefer to write

$$y = a + x$$

instead of something like

1 604 517 211

Finally, we have noted that there is much more to programming than just the arithmetic operations. The logical operations must also be part of this language. It ought to be possible to write loops without worrying about initializing, incrementing, and testing; we must have the option of selecting alternative paths on different occasions; and it should be easy to capitalize on an existing library of subroutines.

There is still a considerable gap between the basic language of machines and this algebraic language. To bridge this gap, however, the techniques of automatic programming have been devised. It is the aim of these procedures to have the computer itself take care of the routine labor, leaving only the essential elements to the human programmer.

Automatic programming is carried out in two phases. First the program is written in a language not far removed from algebra and fed into the machine. This program is analyzed under the control of a translating routine which transforms the pseudo-algebraic program into a sequence of machine operations. These machine-language steps are not executed at this time but are punched on cards or written on tape as output. Thus, in the first phase, the machine is used merely as a translator.

In the second phase, the machine-language program is run. The output of the first phase, together with the data, becomes the input of the second phase.

The translation process is carried out only once. The machine-language program produced during the first stage is used from then on in every production run. In this respect the procedure is quite

different from an interpretive system, such as the Bell code, in which each pseudo-code instruction is translated every time the problem is run.

The chief advantage of automatic programming is that it simplifies programming. The techniques are easy to learn because they are based on familiar algebraic notation; errors are less frequent because the routine bookkeeping is cut down.

There are, however, some important criticisms. First of all, as we have already noted, an extra pass through the machine is needed to translate the pseudo-algebraic program. In some schemes, more than one extra pass is involved. Therefore, although programming and debugging time is decreased, additional machine time is necessary to translate the program.

Another objection is that the programmer is very remote from the actual operation of the machine when using automatic techniques. This is a definite disadvantage when correcting errors or making slight modifications, because one must either struggle to understand and alter the program produced by the machine, or retranslate the whole program. The latter procedure is certainly less strenuous for the programmer, but again it takes machine time.

Yet another serious question is whether the program, as devised by the machine, is as efficient as it would be if done "by hand" by an experienced programmer. Does it take longer to run and are things packed in storage in the best possible way? The efficiency of the final program depends upon the quality of the translating routine. As better translators are written, the programs they produce will consequently be better.

If we are dealing with a program that will take hundreds of hours to run, efficiency is of paramount importance. In that case one can still take advantage of the simple procedures of automatic programming by using the program produced by the machine as a first step, to be refined later by an expert.

All these criticisms notwithstanding, the future of automatic programming seems assured. For the research scientist or engineer it provides a rapid method of programming in a language that is easy to learn; and it is a step on the way toward establishing a universal language for all machines.

6-2. IT, FORTRAN, AND FORTRANSIT

In keeping with our general plan to use the IBM 650 for examples, we will discuss automatic programming in terms of this machine.

A procedure of this kind for the 650 was pioneered by A. J. Perlis, J. W. Smith, and H. R. VanZoeren at the Carnegie Institute of Technology. Officially called the "Internal Translator", it is also generally known as the "Carnegie Tech Compiler", or IT.

In a parallel development, IBM devised an automatic programming scheme for their 704, called FORTRAN (for FORmula TRANslation). Realizing that it would be at least a step in the right direction to have two computers that respond to the same language, IBM combined FORTRAN with the work of Perlis and his associates, and produced FORTRANSIT, a language compatible with both the 704 and the 650.

The language of FORTRANSIT is very similar to that of FORTRAN. The principal differences are due to the fact that the 704 is much more versatile than the 650 and therefore FORTRAN can do correspondingly more complicated things than FORTRANSIT. The latter is essentially a restricted version of the former; anything that can be done in FORTRANSIT can be programmed in FORTRAN in virtually the same way, but the reverse is not true.

Among other things, FORTRAN offers greater variety in format for input and output, but this versatility is actually a source of confusion to the neophyte. It is a frustrating thing to have computed the answers correctly only to find them printed on the output page in an all but incomprehensible jumble! It will be to our profit to content ourselves with the simple input and output facilities of the 650.

We will concentrate on the FORTRAN-FORTRANSIT language, but many of the concepts can be carried over bodily to the Carnegie Tech Compiler. The principal reason for our choice is the applicability of this language to two machines, the 650 and the 704. Other advantages are the ease of programming computations involving arrays such as vectors or matrices, and the variety of symbols available for representing variables.

However, FORTRANSIT is not without its drawbacks. The current procedure for using FORTRANSIT on the 650 is cumbersome. The translation process requires three passes through the machine and

a fourth finally to perform the computation. Furthermore, since it is something of an afterthought, and a curious amalgam of FORTRAN and IT, it is full of peculiar restrictions. There are also several versions of FORTRANSIT available, depending upon the auxiliary equipment attached to the 650.

Therefore, we will make a compromise: we will not discuss all the detailed procedures for using FORTRANSIT on the 650. The steps may be found in programming manuals issued by IBM and they will probably be improved upon in the near future. Instead, we will concentrate on the basic concepts of FORTRANSIT as an example of an automatic programming scheme.

In the sections that follow, the reader will recognize topics we have discussed before, such as loops, branches, and subroutines. We will assume that the reader is familiar with these concepts from previous chapters, and we will only be concerned with how they may be carried over to another language.

6-3. Symbolic Variables

In algebra, we customarily use a single letter to represent a variable. Of course, we do not restrict ourselves to the roman alphabet, and in various disciplines, π, Ω, and $\aleph$ have standard meanings. Furthermore, we may adorn these symbols with garlands of subscripts to give them different meanings.

When the machine assists us in programming, we may also represent the variables by symbols. In translating the symbolic program to the machine language, the computer keeps track of these symbols. The first time a symbol appears, the machine assigns a location in storage to it; and every subsequent time that symbol appears, the computer replaces it in the machine-language program by the same address.

When programming symbolically for machine translation, however, we cannot use subscripts or superscripts in the usual way. In order for information to be fed into the machine, it must be prepared on cards or tape and there are no facilities for representing something above or below the line. Furthermore, we are restricted to the roman alphabet, arabic numerals, and a few other symbols, and there is no distinction made between capital and small letters. One way we overcome this limitation is to use symbols consisting of several letters combined with

numbers, such as AD3, or X4AF. We can also construct symbols so as to suggest other notations that might be more familiar; for example, if some quantities are usually denoted by β or γ we can spell out the symbols BETA and GAMMA.

In the FORTRANSIT language, a symbol may not consist of a combination of more than five letters and numbers: thus, EPSIL is acceptable, but EPSILON is not.

For reasons we have described earlier, most of the computations in scientific work are carried out in floating-point variables; but there is a definite need for fixed-point operations, too. The first letter of a symbol is used to distinguish between variables that are to be manipulated by floating-point arithmetic and those that are to be in fixed point. Fixed-point variables must begin with the letters I, J, K, L, M, or N; symbols beginning with other letters represent floating-point quantities. Several examples are given below:

Fixed point	Floating point
IX2	B8
NU	TEX
M	X17
K3P	ALPHA
JAG81	Z

Note that IX, IX1, and IX2 are three different fixed-point variables; and B, B8, and B19 are three different floating-point variables. Similarly, if the variable HERE is once accidentally misspelled HEAR, the machine will treat HEAR as a completely different variable and will assign it to a new storage location.

6-4. Constants

Just as there are fixed- and floating-point variables, there are also fixed- and floating-point constants, that is, quantities whose numerical values do not change during a series of computations.

In FORTRANSIT, the distinction is very simple: floating-point constants are written with a decimal point, whereas fixed-point constants are not. Fixed-point constants are always integers, but floating-point constants may contain decimal fractions. Thus, 715 is a

fixed-point constant, but 715. is a floating-point constant. Similarly, 85.372, −114.5, and 0.035 are all floating-point constants.

To increase the flexibility, there is a second mode of representation for floating-point constants, involving both positive and negative powers of ten. The two parts of a number in this notation, mantissa and exponent, are separated by the letter E, which precedes the exponent. The mantissa need not be normalized. The following representations are all equivalent:

$$\begin{aligned} &54.7\text{E} + 02 = 5.47\text{E} + 03 = 5.47\text{E}03 \\ &= 5.47\text{E}3 \qquad = 54700.\text{E} - 1 = 5470.\text{E}00 \\ &= 5470. \end{aligned}$$

In FORTRANSIT, this representation can only be used in programming and cannot be used on data cards (see Sec. 6-10). It is most convenient to use the exponent representation when dealing with very large or very small numbers, since it is somewhat easier to write 5.7662E − 6 than 0.0000057662.

A constant containing more than eight significant digits will be truncated to eight for use in the calculation, so it is useless (and it gives a misleading impression of the accuracy) to write 3.1415926536 for the value of π.

6-5. The Concept of a "Statement"

FORTRAN and FORTRANSIT programs are written as a series of statements. Statements are generalized instructions, considerably more flexible than any we have previously discussed.

In the Bell code, every instruction contains precisely 10 digits plus sign (the word size of the 650); FORTRANSIT statements are of variable length.

In the Bell code, we are restricted to one arithmetic operation at a time, so that to perform the computation

$$y = 3x + b$$

we first had to multiply x by 3 in one instruction, and then add b to the product in a second. In FORTRANSIT we can use a single statement to program this entire computation and, indeed, much more complicated ones.

There are statements of different types to permit us to perform all

the kinds of operations we have already discussed. There are statements for performing arithmetic; statements for branching and looping; and input and output statements.

To be fed into the machine, FORTRANSIT statements are punched on cards: one letter, number, or special symbol to a column. In addition to the usual numerical and alphabetical representation, the FORTRANSIT language involves some new symbols which require up to *three* holes in one column:

Symbol	Punches
/	0-1
=	8-3
(	0-8-4
)	12-8-4
*	11-8-4
.	12-8-3
,	0 8-3

Key punches can be obtained to punch these holes automatically.

A blank statement card is illustrated in Fig. 19.

A zero in the first column indicates this card is to be translated. Any other symbol will cause the card to be ignored during translation, but this column cannot be left blank.

In order to refer to a particular statement in the course of a program, that statement must be identified with a unique nonzero statement number. The number is only an identification and the statements need not be numbered in any particular order. Although every statement *may* be numbered, only certain statements *have* to be. From the point of view of the internal operation of FORTRANSIT, it is an advantage to number the statements only when necessary and to keep the numbers as low as possible.

The statement number is punched in columns 2 to 5. Each of these columns must contain a numerical punch; therefore statement number 14 is identified as 0014. If the statement does not have a unique number, these columns are punched with zeros.

A single statement may be so long that it requires more than one card to punch it all. In that case it may be continued on as many as nine subsequent cards. The statement number is repeated in col-

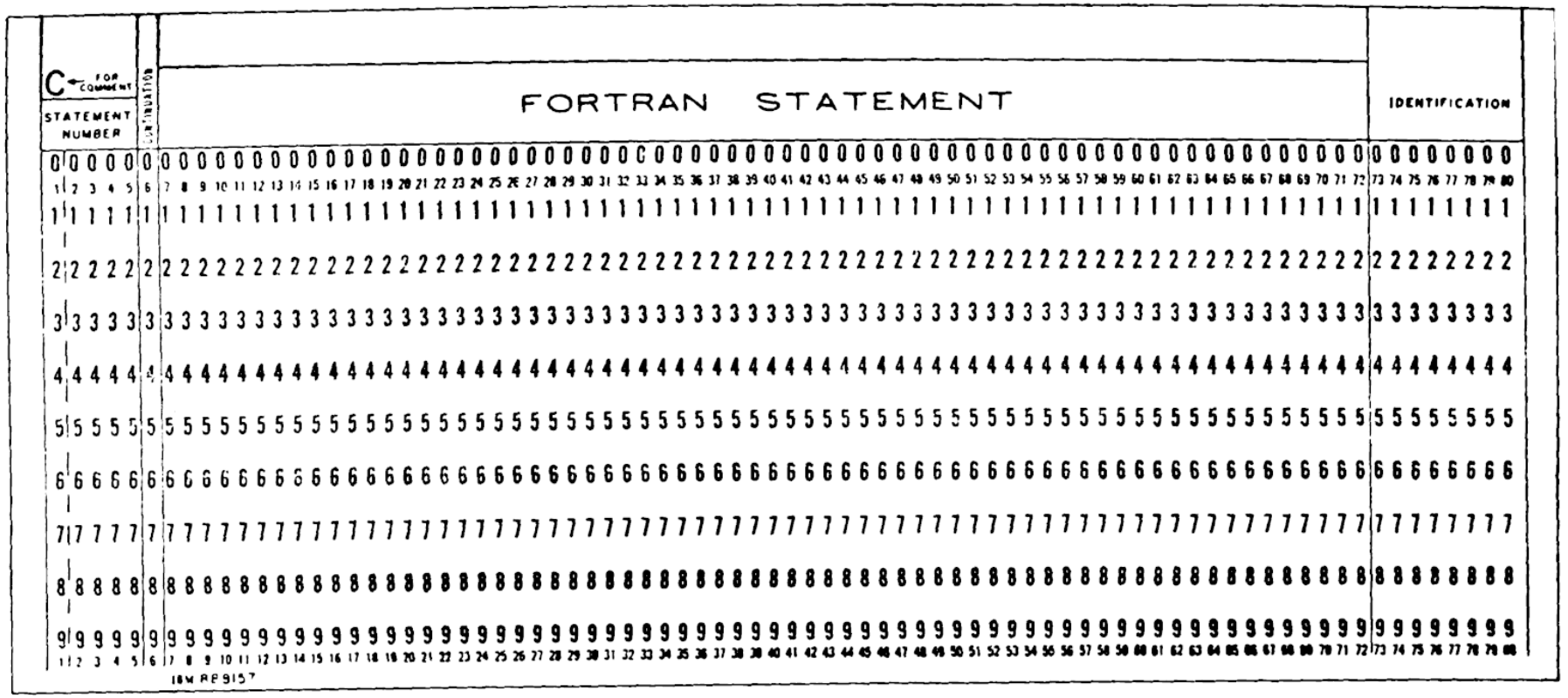


FIG. 19. (*Reproduced by permission of International Business Machines Corporation.*)

umns 2 to 5 on each card; the first card will contain a zero in column 6; the second a 1 in column 6; etc. If only one card is necessary, that card must contain a zero in column 6.

If a statement cannot be punched on a total of ten cards, it must be broken up into two or more statements. Although one statement can involve several cards, it is not permissible to punch two statements on a single card, no matter how short they may be. A new statement must always begin on a new card.

The statement itself is punched beginning in column 7. The number of columns that may be used in the statement field varies depending upon the particular version of FORTRANSIT. In FORTRANSIT II(S), for example, columns 7 to 36 are used for the statement and the rest of the card is ignored.

Blank columns in the statement field are ignored, so statements can be spread out for ease in reading the printed version, or compressed for economy of space.

6-6. Source Program and Object Program

The deck of statement cards (preceded by function title cards described in Sec. 6-9) forms the "source program". This is the deck that is fed into the machine for translation. The sequence of the statements in the source program determines the order of execution (not the statement numbers).

At the end of the assembly or translation phase (which in FORTRANSIT involves three passes through the machine), the programmer is presented with a machine-language deck called the "object program". This is the deck that is used in running the problem.

If the programmer does not wish to be bothered with learning the machine-language code, he does not need to, because it is not necessary for him to understand the instructions punched on the object program deck.

Nor does he have to concern himself with the various decks involved in the assembly operation. For purposes of input and output he will need the correspondence table (Sec. 6-10), but that is all.

Debugging can be done effectively by making use of the conditional punch statement (Sec. 6-10) and the program can be corrected or

modified at the source program stage and reassembled. Care in checking the source program will eliminate the need for many reassemblies.

6-7. Arithmetic Symbols

When writing statements, the standard symbols + and − are used for addition and subtraction. However, to avoid confusion with the decimal point or the letter X, an asterisk * is used for multiplication and the slant / is used rather than ÷ to represent division. Furthermore, since we cannot punch superscripts on cards, we introduce a new symbol, the double asterisk **, for exponentiation. (In FORTRANSIT, only fixed-point exponents are handled automatically.)

Thus

$$A + B,\ A - B,\ \text{and}\ A / B$$

mean just what you would expect, while

$$A * B \text{ is used instead of } A \times B \text{ or } A \cdot B$$

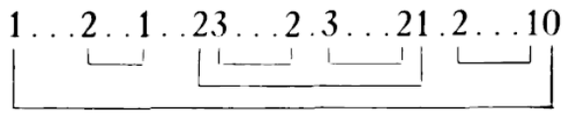
and $$A ** L \text{ represents } A^L$$

The usual abbreviation of algebra, in which AB means A times B, cannot be used in this language because AB can be the symbol for a single variable. It is therefore necessary to write the multiplication symbol * explicitly every time multiplication is intended.

Although in algebraic notation we customarily use a variety of brackets, such as [ ] and { }, as well as ( ), to punctuate arithmetic expressions, we are limited in FORTRAN and FORTRANSIT to the pair ( ). A little thought will convince the reader that any arithmetic expression can be written with just the one pair of parentheses without ambiguity and therefore this is not a serious limitation, although from the point of view of reading an unfamiliar equation, it is sometimes confusing to figure out which open parenthesis is the mate of a closed one. A general scheme for finding pairs is to begin at the left and number each parenthesis, increasing the count by 1 for each open parenthesis (and decreasing by 1 for each closed one). If the parentheses have been used correctly, the number of the last parenthesis must be zero, and the mate to a particular open parenthesis is the first closed one with a smaller label. For example, the scheme

$$(\ldots(\ldots)\ldots((\ldots)\cdot(\ldots))\cdot(\ldots))$$

would be labeled and paired as follows:



The use of several lines for expressions, such as

$$\frac{3x + 2}{4y + 3} + \frac{7x + y}{2y + z}$$

or

$$\cfrac{1}{1 + \cfrac{1}{x + \cfrac{1}{y}}}$$

cannot be achieved on punched cards. In this language, everything must be strung out on a single line. As a result, there are many more parentheses than we normally encounter in algebra. The first expression has to be written

```
(((3. * X) + 2.) / ((4. * Y) + 3.)) + (((7. * X) + Y) / ((2. * Y) + Z))
```

Note that all the variables and constants are in floating-point form and that labeling the parentheses according to the previous rule gives a total of zero, indicating that every parenthesis has a mate.

Actually, not all the parentheses we have used in this example are necessary, because FORTRANSIT has a built-in hierarchy of operations. If an expression without parentheses contains several arithmetic operations, the translating routine will group the factors so that exponentiation will have highest priority; next comes multiplication or division, and finally addition or subtraction. Therefore

```
A * B + C
```

will be interpreted as

```
(A * B) + C
```

and

```
A + B * C
```

will be treated as though parentheses divided it into

```
A + (B * C)
```

Similarly

```
A * B ** E
```

becomes

```
A * (B ** E)
```

and $A / B + C ** D + E$

becomes $(A / B) + (C ** D) + E$

When both multiplication and division occur, or both addition and subtraction, the grouping is taken from the left, as in

$$A * B / C$$

which is interpreted as

$$(A * B) / C$$

or $A / B * C$

which becomes

$$(A / B) * C$$

The beginner would do well to use parentheses liberally if there is any doubt at all in his mind.

6-8. Arithmetic Statements

An arithmetic statement is an explicit equation containing just one variable on the left-hand side and any valid combination of variables on the right. Thus

$$(3.0 * X) + Y = (0.2 * Y) + Z$$

is not acceptable, but the following are:

$$Y = (Z - (3.0 * X)) / 0.8$$

or $$Z = (0.8 * Y) + (3.0 * X)$$

or $$X = (Z - (0.8 * Y)) / 3.0$$

When the program is executed, the contents of the storage location assigned to the variable that appears on the left will be replaced by the numerical value of the expression on the right. Therefore, the three preceding equations will have different meanings to the machine, even though algebraically they are equivalent. In the first equation the variable Y takes on a new numerical value, while X and Z are unchanged; in the second, Z is changed; and in the third, X.

In order for the computation to be carried out correctly, all the variables that appear on the right must have been assigned appropriate values by preceding statements, or must have been read in as data. For example, in the statement

$$Z = (0.8 * Y) + (0.3 * X)$$

the values of X and Y must have been established earlier in the program. The machine will use whatever numbers happen to be in the storage locations associated with X and Y, and if these have not been properly set up, the results will naturally be incorrect.

Here are a few examples of statements in floating-point arithmetic. The equation

$$\frac{1}{f} = \frac{1}{p} + \frac{1}{q}$$

must first be written as an explicit equation for f:

$$f = \frac{1}{\frac{1}{p} + \frac{1}{q}}$$

We can program this as a series of statements:

```
R = 1. / P
S = 1. / Q
T = R + S
F = 1. / T
```

But if we do not need the reciprocals of P and Q later on, there is no need to assign separate symbols to them. In that case, it is easier to write the simple statement

```
F = 1. / ((1. / P) + (1. / Q))
```

Note that all variables and constants are in floating-point form. Also every open parenthesis has a mate.

The "trapezoidal rule" for integrating a function between the points (x_1,y_1) and (x_2,y_2) is

$$\text{Area} = \frac{(x_2 - x_1)(y_2 + y_1)}{2}$$

The corresponding statement is

```
AREA = ((X2 - X1) * (Y2 + Y1)) / 2.
```

The polynomial

$$y = a_0 + a_1x + a_2x^2 + a_3x^3$$

can be written

```
Y = (((((A3 * X) + A2) * X) + A1) * X) + A0
```

A typical fixed-point equation might be

$$J = (2 * K) + 3$$

This can also be written

$$J = 2 * K + 3$$

because of the hierarchy rule which places multiplication before addition. Note that 2 and 3, written without decimal points, are fixed-point constants. The reader is reminded that

$$J = 2K + 3$$

is not an equivalent statement, because multiplication must be explicitly indicated.

In FORTRAN (but not FORTRANSIT), all the quantities that appear on the right-hand side must be variables, constants, and functions in the same arithmetic. That is, mixed expressions involving fixed and floating quantities on the same side of an equation are usually not allowed. For example,

$$Y = (7 * X) + 2$$

would be incorrect because 7 and 2 (without decimal points) are fixed-point constants, and X is a floating-point variable.

On the other hand

$$Y = (7. * X) + 2.$$

is correct. Similarly,

$$Y = 2. * (Y + M)$$

is not correct in FORTRAN, because Y is a floating-point variable and M is a fixed-point variable.

There are two exceptions to this rule. One pertains to "subscripts" in arrays and we will come to that later in this chapter. The other exception is that fixed-point *exponents* are permitted in floating-point expressions, as in

$$y = x^3 - 7$$

which may be written

$$Y = (X ** 3) - 7.$$

Note that the exponent 3 is in fixed form (without the decimal point), but the constant 7. must be in floating-point form.

In both FORTRAN and FORTRANSIT, however, it *is* possible to have a fixed-point variable on the left, even though the expression

on the right is in floating-point form, and vice versa. The result will be computed in the arithmetic on the right and then converted to the arithmetic of the left-hand side before being stored.

When a floating-point number is converted to fixed form, only the integer part is retained. The decimal fraction is ignored; the answer is not rounded—it is truncated. That is, 9.7416 becomes 9 and not 10.

The equations

$$Y = 2 * I + 3$$

and

$$J = (2.6 * A) + 3.851$$

are examples of conversions of this kind.

Although the mixed equation

$$Z = (2.0 * I) + (7.0 * X)$$

is not acceptable in FORTRAN because the fixed variable I and the floating-point variable X appear on the same side of the equation, the equivalent result can be correctly obtained by first transforming I to a floating-point variable Y and then using Y in the calculation:

$$Y = I$$

$$Z = (2.0 * Y) + (7.0 * X)$$

As in our previous discussions, equations like

$$I = I + 2$$

and

$$Y = (0.3 * Y) + 8.0$$

in which the same variable appears on both the left- and right-hand sides, are acceptable. In each case the contents of the storage location assigned to the variable on the left will be replaced by the numerical value of the expression on the right.

6-9. Special Functions

A number of subroutines for evaluating frequently encountered functions may be incorporated in a FORTRAN or FORTRANSIT program. A typical library of FORTRANSIT subroutines for scientific calculations would surely include the following:

FUNCTION	SYMBOL
sine x	SINF(X)
cosine x	COSF(X)
$\log_e x$	LOGF(X)
e^x	EXPF(X)
$\sqrt{x}$	SQRTF(X)

Note that a symbol is associated with every function. Each symbol consists of two parts: a name containing four or five letters and ending in F (for "function"), and an argument enclosed in parentheses. The name may contain numbers, but it must begin with a letter. For example, G3AF is an acceptable function name. The function name with the F removed cannot be the same as the name of any variable used in the program.

The argument is not restricted to X, as given in the table above, but can be any expression in the proper arithmetic mode. For example

$\sqrt{3x + y}$ becomes SQRTF(3. * X + Y)

or $\log_e (\sqrt{x} - z)$ becomes LOGF(SQRTF(X) − Z)

In FORTRAN the result of the operation of the subroutine on the argument will be a floating-point number *unless* the first letter of the name of the function is an X. If the first letter is X, the result will be a fixed-point number. Thus MAXF represents a function with a floating-point result; but XMAXF represents a function with a fixed-point result. The name does not indicate, however, whether the *argument* must be in floating point or fixed; the subroutine description must inform the programmer of that.

In FORTRANSIT, a set of "subroutine title cards" identifying the functions to be used in the program must precede the statement deck before translating. Thereafter, the statements may include functions along with variables and constants. For example, the equation

$$c = (a^2 + b^2 - 2ab \cos \theta)^{1/2}$$

becomes

C = SQRTF((A ** 2) + (B ** 2) − (2.0 * A * B * COSF(THETA)))

and

$$y = be^{-t}$$

becomes

```
Y = B * EXPF(-T)
```

The object program produced by the machine by translating a source program containing references to special functions is not complete. Before it can be used to carry out a computation, it must be supplemented by special decks from the library containing the instructions for evaluating the functions. Details of this process can be found in the FORTRANSIT manual and in the local computing center library regulations.

6-10. Input and Output Statements

The statement

```
READ
```

in the FORTRANSIT source program will cause a group of data cards to be read when the object program is executed. A group may consist of any number of cards; the last card is distinguished by an X-punch in column 10. Reading stops automatically when that X-punch is detected, and the program will go on to the next instruction.

We will discuss only one of the two forms of data cards used in FORTRANSIT: the Type 1 card, distinguished by a Y-punch in column 3. It may contain as many as four variables with twenty columns used for each: ten for identification, and ten for the numerical value.

The columns are assigned as follows:

1–10: identification of first variable
11–20: numerical value of first variable
21–30: identification of second variable
31–40: numerical value of second variable; etc.

If there are fewer than four variables on a card, the next identification field must be punched with ten zeros and the rest of the card may be blank. Thus, if there are only two variables on the card, there must be zeros in columns 41 to 50.

Unfortunately, in FORTRANSIT it is not possible to identify the variables symbolically on the data cards; a numeric code must be used instead. The ten identification columns are divided as follows: the

first and second columns are punched either 01, to identify a fixed-point variable, or 02 for a floating-point variable.

The next four columns are devoted to the variable identification code. During the translation process, the machine assigns a code to each variable encountered, and at the end it produces a correspondence table giving the code corresponding to each symbol. For example, if the variable BETA is assigned the identification code 0007, this must appear in the appropriate columns on input cards (and it will also be punched automatically in corresponding columns of output cards).

Finally, the programmer may use the last four columns for an arbitrary numeric identification. For example, the fixed-point variable IMP may be assigned the identification code 0004 during assembly and we may want to identify a particular input value as part of run #665. The ten-digit identification for this value of IMP therefore becomes

01 0004 0665

where 01 denotes a fixed-point variable, 0004 is the identification code, and 0665 designates the run number.

Fixed-point data are punched as though the decimal point were just to the right of the last digit. Thus 17 is punched

0000000017

Floating-point data are punched with the exponent to the right, but *not* in the same form as constants are punched on statement cards. The mantissa is normalized with the decimal point to the *left* of the first significant digit and must contain eight digits. The exponent consists of two digits with fifty added to the power of ten to avoid negative exponents. Thus the representation of 1 is

1000000051

and *not* 1.0E00. (Note how this also differs from the normalization used in the Bell method.)

In addition to numerical information, columns 20, 40, 60, and 80 should contain a Y-punch (for +) or an X-punch (for −) to indicate the sign.

The statement

PUNCH, X, I, BETA

will cause a card to be punched with the identification and current numerical values of X, I, and BETA. (Note the commas.) The output card will be of the same form as a Type 1 data card. Columns 1 to 10 will contain the identification of X and columns 11 to 20 will contain X in normalized floating-point form. The next ten columns will identify I, and the following ten will contain the value of this fixed-point variable. Finally, the next twenty columns will contain the identification and numerical value of BETA.

Unlike input cards, however, no more than three variables can be punched on an output card. Furthermore, each output card will contain an X-punch in column 10. Therefore, although output cards from one problem may be used as input cards for another, each output card forms a one-card group for the purposes of input, because of the X-punch in column 10.

If a PUNCH statement is not given a unique statement number in FORTRANSIT, it will become a "conditional PUNCH". When the object program is run, a conditional PUNCH statement will cause output only if the sign switch on the console storage-entry switches is turned to minus. Otherwise, it will be ignored. PUNCH statements assigned unique statement numbers will always cause output regardless of the setting of the sign switch.

The conditional PUNCH statement is the principal debugging aid available to the FORTRANSIT programmer (Sec. 6-20).

6-11. Unconditional Transfer

The statement

GO TO 15

will transfer control unconditionally to statement number 15, and the program will proceed from there.

The general form of this type of instruction is

GO TO *n*

where *n* is a statement number.

In FORTRANSIT, it is not permitted to transfer to a PUNCH statement. Otherwise, there are no restrictions.

6-12. PAUSE AND STOP

It is sometimes convenient to have a command that will interrupt the execution of the program temporarily until the Start button is depressed. The statement

PAUSE

serves this purpose. It stops the machine during the execution of the object program (but it does not affect the source program).

A PAUSE statement can be used as an aid in debugging or as a signal during production runs. It is a sort of milepost; when it is reached you know where you are in the program.

When the Start button is depressed, the machine proceeds automatically to the next statement.

The statement

STOP

indicates the end of the program. Pressing the Start key will not cause computation to be resumed. *Every* FORTRANSIT source program must end in a STOP statement because this is also a signal to the translating routine that the end of the program has been reached.

6-13. EXAMPLES OF SIMPLE PROGRAMS

We are now in a position to write a complete program. We can read data, program arithmetic operations, punch the results, and return to the beginning.

For example, the program for computing

$$y = \frac{3x + 7}{4x + 5} + \frac{1}{2}$$

becomes

```
2:  READ
    Y = (((3. * X) + 7.) / ((4. * X) + 5.)) + 0.5
3:  PUNCH, X, Y
    GO TO 2
    STOP
```

We have explicitly given the PUNCH statement a number in order to produce output every time; otherwise it would be a conditional

PUNCH statement and would only produce output when the sign switch on the console is on minus.

Similarly, the example of Sec. 4-7:

$$\pi B = \frac{c_1 \lambda^{-5}}{e^{c_2/\lambda T} - 1}$$

where $c_1 = 3.7403 \times 10^{-5}$

and $c_2 = 1.43868$

becomes, writing B for πB,

```
16:  READ
     B = (3.7403E - 05 / (W ** 5)) / (EXPF(1.43868 / (W * T)) - 1.)
 4:  PUNCH, W, T, B
     GO TO 16
     STOP
```

We have used the symbol W instead of LAMBDA for the variable λ because LAMBDA begins with an L and would therefore be a fixed-point variable. Furthermore, LAMBDA exceeds the five-letter FORTRANSIT limit.

6-14. The Loop Statement

The statement

```
DO 17 I = 1, 5
```

will cause the statements following this one, up to and including statement number 17, to be repeated five times. If the fixed-point variable I occurs in these statements, its value will be 1 the first time through, 2 the second, etc., and finally, 5 the last time.

In this example, 17 is the *goal* of the DO statement; all statements between the DO and 17 are called the *range* of the DO; and I is the *index* of the DO.

Note the location of the comma in this statement. The statement is incomplete without it.

If the fixed-point variables K2 and MY have been previously defined in the program

```
DO 42 JS = K2, MY
```

means, "Do all statements between this one up to and including state-

ment 42. The first time the fixed-point variable JS will have the value K2, and it will be increased by 1 each time. The loop will be terminated after it has been executed with JS = MY". The goal of the DO is 42, and the index is JS.

The fixed-point variable need not be increased by only 1 each time. If it is to be increased by 3, we may write

```
DO 19 I = 1, 10, 3
```

This will cause the loop to be executed first with I = 1, then with I = 4, then 7, and finally 10. On the other hand,

```
DO 19 J = 1, 10, 4
```

would cause the loop to be executed with J = 1, 5, and 9. The loop would not be executed with J = 13 because this would exceed 10, the specified upper limit of J.

Similarly,

```
DO 85 K = I, J, M
```

is acceptable if I, J, and M have been established earlier in the computation. I cannot be less than 1.

Consider the following problem: read N groups of cards, where N is defined earlier in the program. The following instructions will accomplish this:

```
    DO 5 I = 1, N
5:  READ
```

This is simply a counting loop, in which the index is used to count up to N.

The value of

$$K! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots K$$

can be computed by these statements (unless K = 0):

```
    I = 1
    DO 2 J = 1, K
2:  I = J * I
```

Note that it was necessary to initialize I to unity before starting this loop.

The first ten terms of the hyperbolic sine series will be computed by the following loop:

```
S = X
T = X
```

```
      D = 1.
      DO 7 I = 1, 9
6:    T = (T * (X ** 2)) / ((D + 1.) * (D + 2.))
      S = S + T
7:    D = D + 2.
```

In FORTRANSIT, the statement immediately following a DO statement must have a number. We have obeyed this rule in the preceding examples.

A further restriction is that the last statement in the range of the DO cannot be a PUNCH or any kind of transfer statement. Therefore, if we want to punch a result at the end of every performance of the loop we cannot write

```
      DO 6 ........
      ............
6:    PUNCH
```

However, there is a dummy statement in the language just for this kind of situation; the statement

```
CONTINUE
```

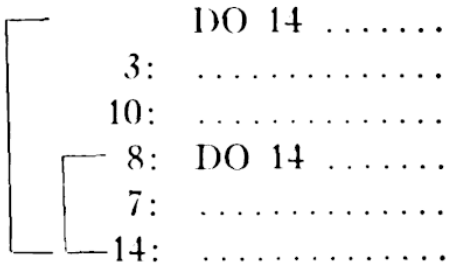
is the equivalent of "No operation". We can achieve the desired result in the example above, by putting a CONTINUE statement at the goal of the DO instead of the PUNCH:

```
      DO 6 ........
      ............
7:    PUNCH
6:    CONTINUE
```

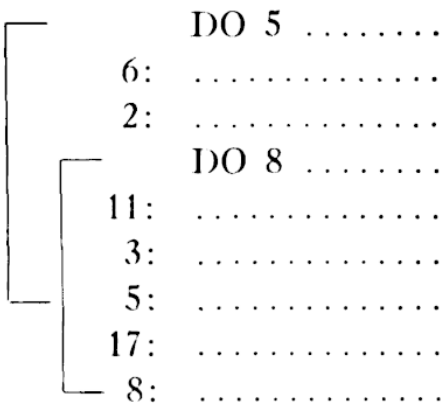
DO statements may be "nested". That is, a DO may contain other DO's within its range. However, the inner loop must lie completely within the outer one. That is, programs like this are valid:

```
      DO 12 ......
7:    ............
11:   ............
      DO 15 .......
6:    ............
4:    ............
15:   ............
9:    ............
12:   ............
```

and also



but not



Furthermore, it is not ordinarily permissible to transfer into the middle of the range of a DO as in the example

The full flexibility of the DO instruction will be evident after we have discussed arrays.

In summary, the general form of a DO statement is

$$\text{DO } n \; i = j, k, m$$

where n is the statement number of the goal; i (the index) is a fixed-point variable; and j, k, and m are either fixed-point variables or fixed-point constants. The smallest permissible value of j is 1 and j, k, and m must all be positive (and the plus sign is not punched). If m is not explicitly given, it is assumed to be one.

6-15. Arrays

One of the principal advantages of the FORTRAN-FORTRANSIT system is the ease with which we can treat arrays. For our purposes, an array is any group of numbers that can be put in order. A one-dimensional array is a list of numbers put in order on the basis of one characteristic. For example, the coefficients of the polynomial

$$y = a_1 + a_2x + a_3x^2 + a_4x^3$$

can be put in the order

$$a_1$$
$$a_2$$
$$a_3$$
$$a_4$$

This particular array contains four *elements*.

Similarly, the x, y, and z components of the vector B form a one-dimensional array of three elements:

$$B_x$$
$$B_y$$
$$B_z$$

Suppose we want to perform a particular calculation for seven different values of a parameter A. We can put them in an arbitrary order, and call the first one A_1, the second A_2, etc. In this way we can form a one-dimensional array of seven elements.

Now consider the three simultaneous linear equations

$$\begin{aligned} 2x - 7y + 2z &= -8 \\ x + y - 4z &= 15 \\ 3x - 3y + z &= 7 \end{aligned}$$

There are three equations for the three unknowns: x, y, and z. Each unknown has a different coefficient in each equation: the coefficient of y in the first equation is -7, in the second, $+1$, and in the third, -3. The coefficients of the unknowns can be written as a two-dimensional array with nine elements arranged in three columns and three rows:

$$\begin{matrix} 2 & -7 & 2 \\ 1 & 1 & -4 \\ 3 & -3 & 1 \end{matrix}$$

This array is two-dimensional because the position of any element is given by two factors. One factor is the number of the equation; this determines the row. Thus any coefficient in the first equation must appear as an element in the first row. The second factor is the variable x, y, or z; this determines the column. Thus all the coefficients of x appear in the first column. We see, therefore, that the coefficient of z in the second equation is in the second row, third column; and the coefficient of y in the first equation is in the first row, second column.

We can assign two numbers to each element in a two-dimensional array. By convention the first number denotes the row and the second the column. An array of the kind we have just been discussing contains nine elements, labeled as follows:

$$\begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

FORTRANSIT can deal with arrays of one or two dimensions. However, instead of using the conventional subscript notation in which A_3 denotes the third element in the A array, the subscript must be given in parentheses: A(3). Otherwise, the labeling scheme is the same. In a two-dimensional array the subscripts are separated by commas: B(2,3) is the element of the B array in the second row and third column.

Suppose the C array contains 2 rows and 2 columns. The elements would be labeled in FORTRANSIT as follows:

C(1,1) C(1,2)
C(2,1) C(2,2)

Any symbol that can be used to represent an ordinary variable can be used to represent an array. When it designates an array it is called a *subscripted* variable and the quantities written within the parentheses are called the subscripts. Within a single program, the same symbol cannot be used to represent both a subscripted and a nonsubscripted variable.

As before, if the first letter of the name of a subscripted variable is I, J, K, L, M, or N, the *elements* of that array are fixed-point numbers; otherwise they are in floating point.

The subscripts must be integers, but they can contain certain com-

binations of fixed-point constants and fixed-point variables. The following types of subscripts are permitted:

1. A single, unsigned, fixed-point constant, such as 3
2. A single fixed-point variable, such as I
3. A fixed-point variable plus or minus a fixed-point constant, such as J + 2, or K − 3
4. A fixed-point constant times a fixed-point variable, such as 3 * NU
5. A fixed-point constant times a fixed-point variable plus or minus another fixed-point constant (written without parentheses), such as 2 * J + 1, or 5 * KAP − 2.

The following examples are acceptable subscripted variables:

GAM(2 * J) PI(3,K) M(2 * I + 1, 2 * J + 1)

GAM represents a one-dimensional floating-point array; PI is a two-dimensional floating-point array; and M is a two-dimensional fixed-point array.

The programmer must clearly understand the distinction between BI and B(I). BI represents a single variable that has no particular relation to B1, B2, or B3. B(I), on the other hand, represents the Ith element in the B array. In particular, the sequence of statements

```
I = 2
B(I) = X
```

will set B(2) equal to the value of X; but the sequence

```
I = 2
BI = X
```

will only affect the value of variable BI, and not the variable B2.

6-16. DIMENSION Statements

It is necessary to inform the machine, during the translation phase, of the number of elements a given array may contain, so that the correct number of storage locations will be reserved. This is done by preceding the first statement referring to an array by a DIMENSION statement that gives the name of the array and the maximum value of the subscripts. Several arrays may be specified in one DIMENSION statement, separated by commas. For example, if AX is to represent a one-dimensional array of five elements, and KN3 is a two-

dimensional array with four rows and three columns, the program must contain the statement

```
DIMENSION AX(5), KN3(4,3)
```

before any other statement involving AX or KN3.

It is a wise precaution to put all DIMENSION statements at the very beginning of the source program.

6-17. Loops Involving Arrays

In Chap. 3 we discussed a summation loop of the type

$$s = \sum_{i=1}^{8} a_i b_i$$

We can easily compute this sum by combining the concept of an array with a DO statement.

Let us define the elements of arrays A and B as follows:

a_1 = A(1)	b_1 = B(1)
a_2 = A(2)	b_2 = B(2)
etc.	etc.

The program for our problem then becomes

```
4:  READ
    S = 0.
    DO 3 I = 1, 8
2:  S = S + (A(I) * B(I))
    PUNCH, S
3:  CONTINUE
1:  PUNCH, S
    GO TO 4
    STOP
```

The arrays will be read in on statement 4. The next statement initializes the sum to zero. The DO has as its goal statement number 3 and there are four statements in the range of the DO.

The statement following the DO is given a number, as required. I, the index of the DO, appears as the subscript in both the A and B arrays. Therefore the first time through the loop, statement 2 will be executed as

```
S = S + (A(1) * B(1))
```

and the second time as

$$S = S + (A(2) * B(2))$$

and so forth. The index in this example is used not only to count the number of terms in the sum but also to select the proper elements from the array.

The next statement is a PUNCH without a number; therefore it is a conditional PUNCH. It was included here as a debugging aid. When the program is first being tested it will be run with the sign switch on minus and the partial sums, S, will be punched to check that they are being built up correctly. After we are assured that all is going well, we can switch the sign to plus and the conditional PUNCH will be ignored.

We could not use the conditional PUNCH statement as the goal of the DO so we have used the CONTINUE statement as a dummy.

Statement 1 punches the answer. It is not a conditional PUNCH statement because it has a statement number. The following statement sends the computer back to statement 4, which reads in more data. Finally, the STOP statement indicates the end of the program.

Loop instructions can also be used to create arrays, as in this problem: form a one-dimensional array of ten elements in which the nth element is

$$\frac{\cos nx}{n}$$

Assuming that x has been defined earlier in the problem, the array can be constructed by looping as follows:

```
      DO 7 N = 1, 10
3:    Y = N
7:    C(N) = (COSF(Y * X)) / Y
```

It is necessary to convert the fixed-point variable N to the floating-point variable Y before the computation on the right-hand side of statement 7 can be carried out in FORTRAN.

The following loop will construct a one-dimensional floating-point array consisting of all the odd numbers between 9 and 21:

```
      C(1) = 9.
      DO 4 I = 2, 7
4:    C(I) = C(I - 1) + 2.
```

Note that the subscript on the right in statement 4 is one of the permissible types, involving a fixed-point variable and a fixed-point constant.

6-18. Branching and Terminating Loops

There are two ways of selecting alternative paths in FORTRANSIT. One is on the basis of whether a selected quantity is negative, zero, or positive; the other is on the basis of the value of a positive fixed-point variable.

We will discuss the first type of branch in this section. It is illustrated by the statement

$$\text{IF } ((6. * X) - (2. * Y))\ 34,\ 2,\ 7$$

This statement means the following: if the value of the expression

$$(6. * X) - (2. * Y)$$

is negative, execute 34 next; if it is zero, execute statement 2 next; and if it is positive, execute statement 7.

Suppose, for example, that our flow diagram contains a box to be skipped if the variable Q = 0 (Fig. 20). We have labeled the boxes with the corresponding statement numbers. This flow can be achieved by the following scheme:

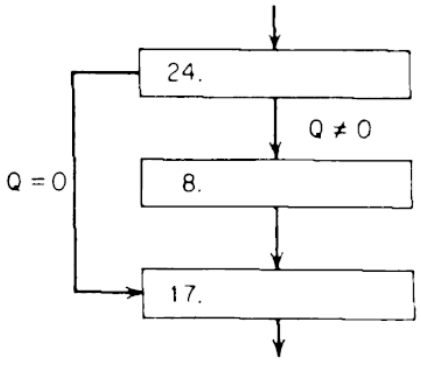


Fig. 20

```
24:  IF (Q) 8, 17, 8
 8:  ..............
17:  ..............
```

The flow diagram of Fig. 21 can be programmed as follows:

```
11:  IF (P − Q) 14, 12, 7
14:  ..................
     GO TO 9
12:  ..................
     GO TO 9
 7:  ..................
 9:  ..................
```

When a loop is to be terminated on the basis of the magnitude or sign of a result, the IF instruction is used to make the decision, just the way the TRANSFER on sign or exponent instructions were used in Sec. 3-9. For example,

```
IF (Y - 0.001) 8, 8, 3
```

will send the program back to statement 3, unless Y is less than or equal to 0.001, in which case it goes on to 8.

As we might expect, the effect of the statement

```
DO 4 I = 1, 8
```

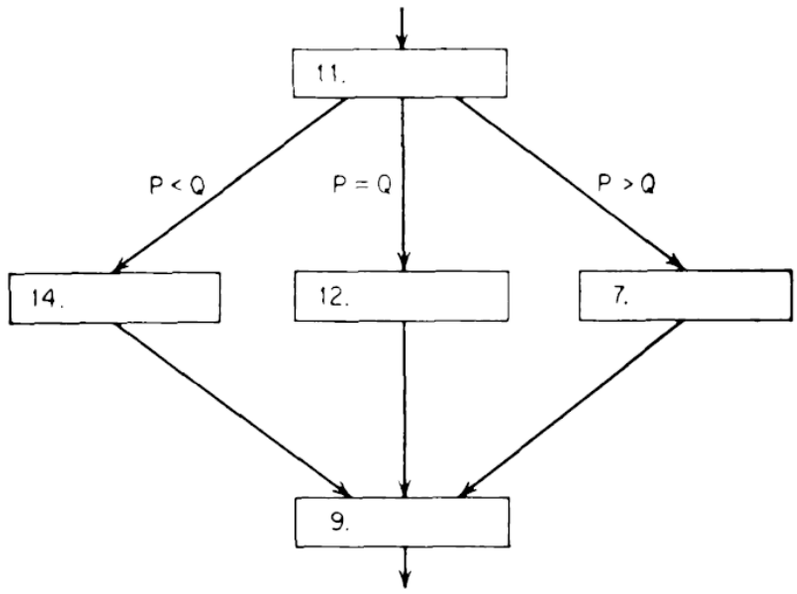


FIG. 21

can be duplicated in three statements: one to initialize I, one to increment it, and an IF statement to test if the loop has been satisfied:

```
      I = 1
2:    .....
      .....
4:    .....
      I = I + 1
      IF (I - 8) 2, 2, 6
6:    .....
```

An IF statement cannot send the program to a PUNCH statement. Here again the CONTINUE statement can be used to avoid difficulties.

6-19. Branching Out of a Subroutine

As we have already seen in Sec. 4-3, it may be necessary to evaluate the same function several times during the course of a program. If the routine for this calculation is not available in the FORTRANSIT library of special functions, it can be written as a series of FORTRANSIT statements and treated as a subroutine.

One of the problems we discussed in Sec. 4-4 was the difficulty of subroutine exit. You will recall that each time the subroutine box is completed, we must go somewhere else.

FORTRANSIT provides a simple method of taking care of multiple exits. There is another kind of GO TO statement, called the "computed GO TO". The statement

GO TO (7, 18, 8, 41), K

will cause a transfer to statement 7 if K = 1, to statement 18 if K = 2, to statement 8 if K = 3, and to 41 if K = 4.

Suppose we want to program the flow diagram of Fig. 22. Let us say that the subroutine box begins with statement 15. The last statement of the subroutine must be

GO TO (6, 4, 11), M

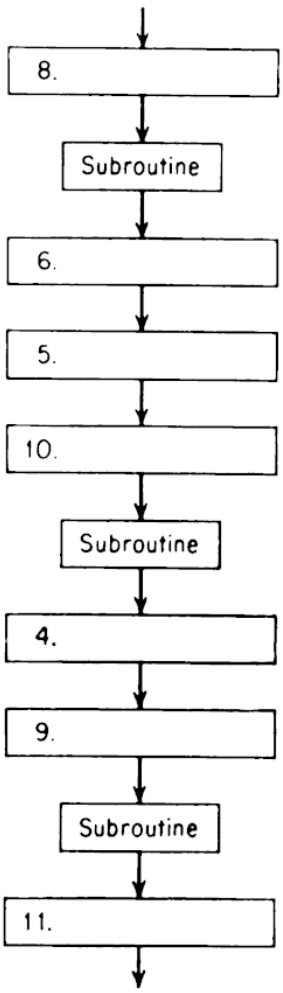


Fig. 22

Then the program will look like this:

```
 8:  . . . . . .
     M = 1
     GO TO 15
 6:  . . . . . .
 5:  . . . . . .
10:  . . . . . .
     M = 2
     GO TO 15
 4:  . . . . . .
```

```
 9:  . . . . . .
     M = 3
     GO TO 15
11:  . . . . . .
     etc.
```

As you see, before entering the subroutine, the variable M is given the value corresponding to the desired exit. When the subroutine is completed, the value of M selects the exit.

The general form of a statement of this type is

$$\text{GO TO } (n_1, n_2, n_3, \ldots), i$$

where n_1, n_2, n_3, etc., are statement numbers, and i is a fixed-point variable.

The fixed-point variable need not be assigned by a simple statement such as

$$J = 2$$

It can also be computed during the calculation, as in the statement

$$J = 2 * K + 1$$

That is why this is called a "computed GO TO" statement.

Here again, you cannot transfer to a PUNCH statement directly.

It is possible to leave the range of a DO, transfer to a subroutine, and transfer back into the DO provided the subroutine does not affect the value of the index of the DO.

6-20. Debugging

Preparation for debugging a FORTRANSIT program should be made in the original source program. Conditional PUNCH statements should be introduced at every likely place. The beginner usually errs by not using this instruction liberally enough.

Conditional PUNCH statements are used to obtain snapshots, as previously described in Chap. 5. PAUSE statements are used in much the same way as programmed stops in the Bell code.

These two types of statements provide virtually the only debugging aids in FORTRANSIT. They are all that are really necessary, however.

Remember that every reassembly takes time so it is wise to check

and double-check every statement to be sure that it is of the correct form. Watch out for commas, extra parentheses, and taboos (such as a transfer to a PUNCH statement). Care in checking the source program pays off in a minimum of debugging time and the need for few reassemblies.

Problems for Chapter 6

1. Program the following problems using FORTRANSIT:

(*a*) Chap. 2, Probs. 2, 5, 6, 7, and 8, assuming that there are subroutines available for square root, logarithm, cosine, etc.

(*b*) Chap. 3, Probs. 2, 4, 5, and 6

(*c*) Chap. 4, Prob. 6.

2. Write and test a FORTRANSIT program for evaluating the determinant of a matrix of arbitrary size, up to 20×20. Include instructions for reading in a code word describing the size of the matrix and instructions for reading in the elements plus some identification. Punch the determinant and the identification at the end of the program.

3. Write and test a FORTRANSIT program for finding the real roots of a polynomial of arbitrary degree up to 10.

4. Write a FORTRANSIT subroutine for carrying forward the solution of a set of simultaneous ordinary differential equations by an increment h in the argument, using the Runge-Kutta method. Test by integrating any second-order equation with known solution one hundred steps forward.

PART TWO

Advanced Programming

7

Arithmetic Instructions in Machine Language and SOAP

7-1. Introduction

We come at last to a description of the basic language of the machine, the series of commands in terms of which all programs must eventually be expressed in order to be run on the computer. As usual, although most of the techniques will be quite general, we will specifically use the IBM 650 for examples.

This part of the book is intended for the reader who has *already tried* the techniques of the preceding chapters and found them inadequate for his needs. He may have discovered that his program took too long in the Bell interpretive code, or exceeded the storage capacity; or he may wish to correct or alter the object program produced by FORTRANSIT without using the additional machine time necessary for reassembly. For that reader the machine language is a necessary tool. If the problem cannot be done in machine language, it simply cannot be done on the computer at all.

We have also reached the point beyond which the idiosyncrasies of the IBM 650 become important in programming. Since not every reader will be interested in the details of 650 operation, we have in-

dicated by a star (⋆) those paragraphs or entire sections which may be omitted if the reader wants only a general introduction to machine-language programming.

7-2. Word Size, Storage Capacity, and the Basic Registers

The IBM 650, as pointed out earlier, has a word size of 10 decimal digits plus sign. Both data and instructions must be expressed in words of this size.

The arithmetic of the *basic* 650 is *fixed point*. Auxiliary equipment can be obtained to permit floating-point operations as well, but we will not consider those instructions.

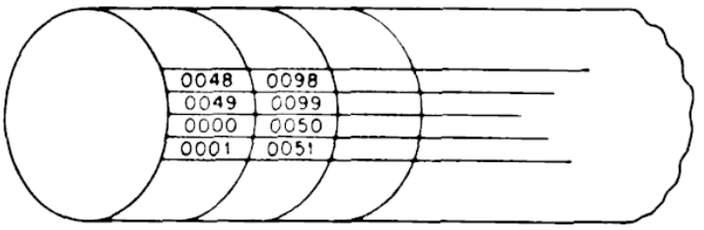


Fig. 23

The storage of the basic machine is limited to 2,000 words with addresses from 0000 to 1999.

⋆The information is stored in the form of magnetic bits on a magnetic drum that rotates at 12,500 rpm. Storage locations are arranged in forty bands, each containing fifty locations, and are usually labeled by the first address in the band: thus we speak of the zero band, the 1500 band, the 1950 band, etc. Location 0049 is followed by location 0000; location 0050 is in the next band down the drum (Fig. 23).

⋆Information can be transmitted to or received from the drum through a set of "read-write heads" under which the drum rotates. Only one storage location in a particular band is under a read-write head at one time; when location 0024 is under a read-write head, location 0074 in the next band and locations 0124, 0174, 0224, etc., up to 1974 are also under read-write heads. As the drum rotates, every storage location in a band is brought, in turn, under the read-write head of that band.

Most of the manipulation of numbers takes place in a register

called the "accumulator". This is actually two registers, called the "upper accumulator" and the "lower accumulator"; in some operations these are treated separately, and in others they are coupled.

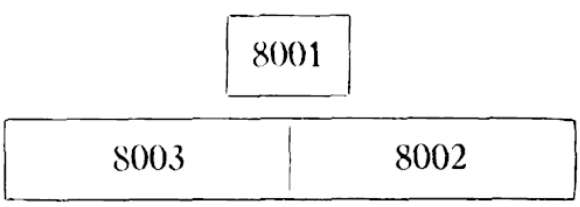
In order to transmit information from the drum to the accumulator, or from the accumulator to the drum, the numbers must first pass through a register called the "distributor".

The distributor, the upper accumulator, and the lower accumulator each contain 10 decimal digits plus sign. In all instances except one, however, both upper and lower accumulator will have the same sign. The exception will be found under the discussion of division.

Each of the registers has an address, as follows:

distributor:	8001
lower accumulator:	8002
upper accumulator:	8003

Distributor, upper, and lower accumulator form the "arithmetic unit" of the 650 and it is often convenient to regard them as though they were arranged as follows:



The only other address acceptable to the basic 650 is 8000, which belongs to the storage-entry switches on the console. Numbers dialed into these switches can be used as instructions or data, although all addresses in the 8000's are subject to certain restrictions.

The program register, to which instructions are sent in order to be executed, does not have an address.

7-3. Functions of the Accumulator during Arithmetic Operations

There are many parallels between the operation of the accumulator of an electronic computer and the dials on a desk calculator. On a desk machine, sums, differences, products, and quotients must be built up in the dials. Similarly, on an electronic computer, these quantities are built up in the accumulator. It is impossible, for ex-

ample, to add two numbers stored in the memory, without going through the accumulator.

Every arithmetic operation consists of two factors: addend and augend, minuend and subtrahend, multiplier and multiplicand, or dividend and divisor. In desk computer calculations, one factor comes from the keyboard during the execution of the operation; on an electronic computer, one factor usually comes from the memory. The location of the second factor depends upon the nature of the operation. This is true on a desk calculator as well as an electronic machine.

There are usually two sets of dials on a desk calculator: one row has room for as many digits as there are columns on the keyboard, and another row has twice as many. The double row is necessary because, for example, the product of two 10-digit numbers can be a 20-digit number.

On the 650, when the 20-position accumulator is used as a unit, it acts very much like the double row on the desk calculator; products are built up there, and that is where the dividend is placed before division.

When the accumulator is split into an upper and lower part, sometimes one and sometimes the other half will act like the single row of dials on the desk machine. For example, the multiplier must always be in the upper accumulator, but quotients appear in the lower. In spite of these minor differences, however, anyone familiar with the operation of a desk computer can easily become accustomed to the behavior of the accumulator on an electronic machine during arithmetic operations.

On both the desk machine and the electronic calculator, arithmetic operations must be carried out in a series of steps. For example, addition is broken up as follows on a desk machine:

1. Clear dials.
2. Enter one factor on keyboard and transfer to dials.
3. Enter other factor on keyboard and add to the number in the dials.
4. Write down sum.

Of course, in an electronic computation, both the addend and the augend are initially stored in the memory. Steps 1 and 2 above can be combined on an electronic computer because a single command

will clear or reset the accumulator and transfer a number to it from the memory. The steps therefore become:

1. Reset accumulator and add one factor to it from the memory.
2. Add other factor to the accumulator without clearing.
3. Store sum by sending it from the accumulator to the memory.

Each of these steps requires a separate instruction. Therefore, in 650 machine language, it takes three instructions to accomplish fixed-point addition, whereas floating-point addition can be done in the Bell code by one:

1 A B C

or in FORTRANSIT, by the single statement

C = A + B

7-4. The Form of a 650 Instruction Word

An instruction word for the IBM 650 consists of ten digits plus sign; the digits are arranged as follows:

xx	xxxx	xxxx
Operation Code	Data Address	Instruction Address

Every order has a two-digit numerical code associated with it and these two digits occupy the first two locations in an instruction word.

The next four digits are called the "data address" or D-address, because in many (but not all) of the orders, this part of the word gives the address from which data are to be taken or to which data are to be sent.

Finally, the "instruction address", or I-address, normally gives the location from which the *next* instruction is to be taken. Thus, for example,

65 0300 1723

is an instruction which clears the entire accumulator, adds the contents of storage location 0300 to the lower part, then proceeds to location 1723 for the next instruction.

⋆7-5. Optimization

In sequential programming, the instructions are placed in successive storage locations in the memory. This was the procedure we followed in discussing the Bell code.

From the point of view of a magnetic drum machine like the 650, however, sequential programming is not very efficient. To understand the reason you must know a little more about the internal operation of the machine. When the computer receives an instruction, a finite length of time must elapse before it is ready to receive data; and a finite length of time must follow *that* before it is ready to receive a new instruction. During all that time the drum is rotating under the read-write heads and if the appropriate address containing the data or the next instruction is not under the head when needed, the machine waits until it is.

The time necessary for successive storage locations to pass under the read-write head is called "one word time". Since there are 50 words in a band and the drum rotates at 12,500 rpm, there are $12{,}500 \times 50$ word times in one minute. Consequently, one word time equals 60/625,000 seconds, or 0.096 millisecond.

If it would take one rotation of the drum, or fifty word times, for the machine to execute an instruction and be ready for the next one, sequential programming would be acceptable. But suppose it takes only five word times; then while the instruction in 0025 is being executed, 0026, 0027, 0028, 0029, and 0030 are passing under the read-write head. If the next instruction is stored in 0026, the machine must execute 0.9 of a rotation before location 0026 is again in a position to be read. During all that time the machine is idle.

An "optimized" program is one in which storage locations for both data and instructions are planned so that they will come under the read-write head as soon as possible after the machine is ready for them. By choosing the D- and I-addresses in accordance with certain rules, a great deal of time can be saved in running the program as compared with sequential programming.

Optimizing "by hand" is tedious business. Rough optimization can be achieved by spacing data and instructions several locations apart around the drum; but detailed optimization requires a knowledge of the exact number of word times needed by every type of instruction.

However, the computing machine itself can be used to optimize the program. This requires an extra pass through the machine, but it is worth it. One such procedure, known as SOAP, will optimize a program almost as well as it can be done by hand and with far less strain on the programmer. SOAP also has other important advantages to be discussed in the next section.

The author strongly recommends that optimizing by machine be used by all "amateur" programmers. Time spent in learning the word times associated with different codes is time wasted unless you plan to make a career out of the 650.

It should be pointed out that optimizing is not a problem in machines with magnetic core or electrostatic storage because every location in the memory is available on an equal basis.

7-6. An Introduction to SOAP

Although we may not wish to depart too radically from the actual operations the machine performs, we may still program in terms of symbols instead of numerical codes and memory addresses, by using the method known as SOAP (for Symbolic Optimal Assembly Program). In this process, as in FORTRANSIT, the programmer prepares a deck of cards in a symbolic language and the machine is used to translate it into machine codes and memory addresses. The symbolic language of SOAP, however, does not make use of algebraic notation. One instruction in the SOAP code becomes precisely one instruction in the machine-language deck and only those operations that can be performed in the machine language can be performed in SOAP and with exactly the same restrictions.

For every two-digit operation code in the machine language, there is a corresponding three-letter SOAP code. In the translating process, the letters are replaced by the appropriate numbers. The advantage of the letter code, however, is that it is easy to remember. For example, the operation code 24 means "Store the distributor", but the number 24 contains no clue as to its meaning. On the other hand, it is easy to associate the SOAP code STD with "STore Distributor".

In all our discussions of machine codes we will also give the SOAP code (specifically, the SOAP II code).

We will postpone a discussion of symbolic addresses in SOAP until we have learned a little more about machine programming.

7-7. The Location of the Decimal Point

In arithmetic operations involving fixed-point numbers it is convenient to regard the decimal point as always being to the immediate left of the first digit:

This implies that every number used in calculation must be less than 1 in absolute value.

When the two parts of the accumulator are coupled, the decimal is assumed to lie to the left of the upper accumulator. When it is split (as it would be following division), each half becomes a number with ten decimal places.

The advantage of adopting this standard location is that it removes the confusion of assigning the decimal point following multiplication and division. Every schoolboy knows that the product of two numbers, each with 10 decimal places, is a number with 20 decimal places. Since it is also true that the product of two numbers, each less than 1, is itself less than 1, we easily see that the product of two 10-digit numbers, with decimal point assigned by this convention, will fill the 20-digit accumulator, and can never exceed it.

Similarly, if the quotient of two numbers is less than 1 (and we must be sure of this ahead of time; otherwise the machine will not carry out the division), adopting this convention for both dividend and divisor will assure that the decimal point will be at the far left in both quotient and remainder.

Of course, the numbers normally encountered in computations do not always have the decimal point so conveniently located. The process of adjusting the decimal point to conform to the convention is called "scaling" and we will discuss it in Sec. 7-19. In describing the arithmetic operations, however, we will assume for the time being that all numbers have already been scaled, and that the decimal point is at the far left in every factor involved.

7-8. The Addition Pattern

Let us see the steps involved in using the lower accumulator to add two fixed-point numbers, located in 0072 and 0084, respectively; the sum is to be stored in 0136.

In the first step, we reset the entire accumulator and bring the contents of 0072 to the lower part. This is done by the command "Reset and add to the lower accumulator", or "Reset Add Lower" as it is more frequently called:

machine language:	65
SOAP:	RAL

The data address of this instruction is the location of the information to be brought to the lower accumulator: in this case, 0072. The I-address will be the location of the next instruction. Let us temporarily ignore optimization (Sec. 7-5) and place the orders in sequential memory locations. If we put the RAL instruction in 0101, we can put the next instruction in 0102. The complete command is then

0101: 65 0072 0102

It is important to note that any instruction that clears the accumulator clears the *entire* accumulator and not just the lower or upper half. After the execution of an RAL instruction, the upper accumulator contains ten zeros.

When the number in 0072 is brought to the lower accumulator, it passes through the distributor and replaces whatever was there before.

The second step is to add the contents of 0084 to the lower accumulator without resetting. The appropriate command is "Add Lower":

machine language:	15
SOAP:	ALO

In our problem, the instruction becomes

0102: 15 0084 0103

(We are continuing to store instructions sequentially.)

Here again the number in 0084 passes through the distributor, where it replaces the number that was previously there. Addition does not take place in the distributor.

Finally, we want to store the sum in location 0136. The order that will accomplish this is "Store Lower":

machine language:	20
SOAP:	STL

The data address indicates the location to which the contents of the lower accumulator is to be sent. The instruction we want is therefore

0103: 20 0136 0104

In going from the lower accumulator to the memory, the information passes through the distributor, where it again replaces the number that was previously there.

The complete sequence needed to perform the addition and storage is

```
0101:  65 0072 0102  (RAL)
0102:  15 0084 0103  (ALO)
0103:  20 0136 0104  (STL)
```

Note that in instructions such as RAL and ALO, in which information comes *from* the memory *to* the accumulator, the D-address specifies the location *from which* data is to be taken. On the other hand, in an instruction like STL, in which information goes *from* the accumulator *to* the memory, the D-address specifies the location *to which* the information is to be sent.

To add four numbers, it is not necessary to store the intermediate sums, but only the final one. The appropriate sequence would be

```
RAL
ALO
ALO
ALO
STL
```

It is necessary to *reset* and add at the beginning so as to clear away anything that might be left in the accumulator from previous operations. All subsequent additions take place without resetting; and finally the sum is stored. *Storing does not clear the accumulator*, so any subsequent operation will probably begin with an instruction that resets the entire accumulator.

For convenience, we will sometimes refer to a sequence of instruc-

tions beginning with an operation that resets the accumulator and ending with one that stores the result as a "sentence".

7-9. Addition, Subtraction, and Storage Codes

The following table contains all the fixed-point addition and subtraction codes valid on the basic 650. They are given in two forms: SOAP and machine language:

SOAP	Machine language	Name
AUP	10	Add Upper
SUP	11	Subtract Upper
ALO	15	Add Lower
SLO	16	Subtract Lower
AML	17	Add Absolute Magnitude Lower
SML	18	Subtract Absolute Magnitude Lower
RAU	60	Reset Add Upper
RSU	61	Reset Subtract Upper
RAL	65	Reset Add Lower
RSL	66	Reset Subtract Lower
RAM	67	Reset Add Absolute Magnitude Lower
RSM	68	Reset Subtract Absolute Magnitude Lower

We have used "Add Upper" as an abbreviation for "Add to the upper accumulator"; similarly, "Reset Add Absolute Magnitude Lower" is a shortened version of "Reset and add the absolute magnitude to the lower accumulator". In each case, the D-address specifies the location of the data.

SLO is similar to ALO except that the number in the location specified by the D-address is *subtracted* from the contents of the lower accumulator.

Again it should be pointed out that RAU, RSU, RAL, RSL, RAM, and RSM are all resetting instructions and clear the *entire* accumulator.

As is seen in the table, there are parallel instructions for both upper

and lower accumulator with one exception: addition and subtraction of *absolute magnitudes* can only take place in the lower accumulator.

We may store the result from either the upper or the lower accumulator:

SOAP	Machine language	Name
STL	20	Store Lower
STU	21	Store Upper

The D-address of STL or STU must be a memory address and cannot be the address of upper or lower accumulator or distributor.

7-10. Format of a SOAP Card; Absolute Addresses

SOAP instructions that are to be translated by the machine must be punched, one to a card, in a special format which makes use of columns 41 to 72. The card is broken into nine fields as follows (Fig. 24):

Column(s)	Function
41	Type
42	Sign
43–47	Location
48–50	Operation Code
51–55	D-address
56	Tag for D-address
57–61	I-address
62	Tag for I-address
63–72	Remarks

For the time being we will only be concerned with four fields:

43–47	Location
48–50	Operation Code
51–55	D-address
57–61	I-address

giving the location of the instruction and its three parts: operation, D-address, and I-address. Note that the form of a SOAP instruction is the same as that of a machine-language instruction, but the operation code and the addresses take 3 and 5 locations, respectively, instead of 2 and 4.

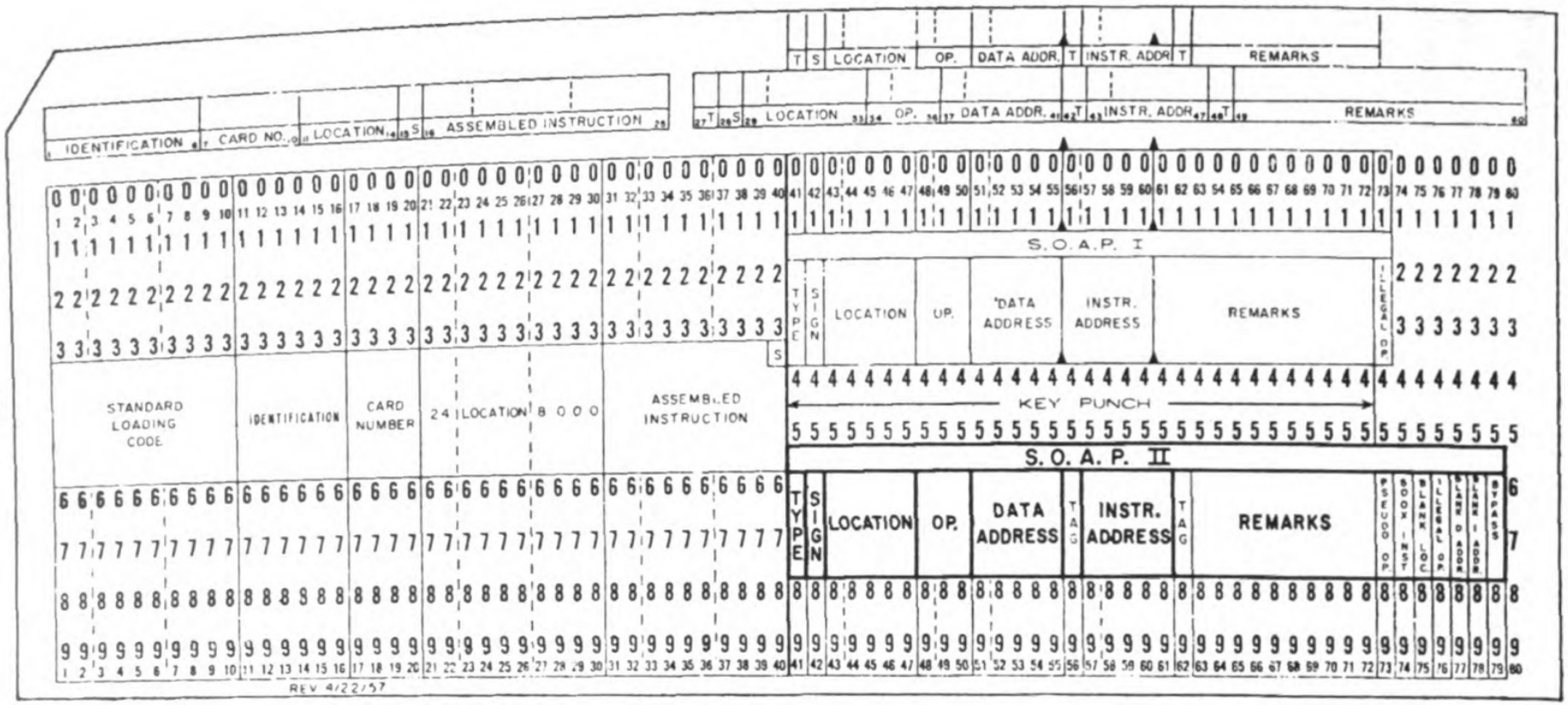


FIG. 24. (*Reproduced by permission of International Business Machines Corporation.*)

The operation code on a SOAP card may be given in either the 3-letter SOAP code (such as RAU) or the 2-digit machine-language code (60). If we use the 2-digit code, however, column 48, the first column of the operation code field, must be left blank. This is a characteristic of SOAP: whenever we use an operation code or address in the machine language, the first column in the corresponding field must be left blank. In the text, we will indicate this blank by two vertical lines || ; for example, we will write ||60 to emphasize that the first column of the operation field is blank.

We may specify the Location, D-address, or I-address on a SOAP card as regular machine addresses; these are often called *absolute* addresses, which means that the process of translation from SOAP to machine language leaves them unchanged.

To punch the instruction

0105: 65 1920 0803

in absolute form on a SOAP card, columns 43, 48, 51, and 57 must be left blank:

COL.	41	42	43 44- -47	48 49, 50	51 52- -55	56	57 58- -61	62	63- -72
	TYPE	SIGN	LOCATION	OPERATION CODE	DATA ADDRESS	TAG	INSTRUCTION ADDRESS	TAG	REMARKS
			0 1 0 5	6 5	1 9 2 0		0 8 0 3		

In the text we will indicate this as follows:

||0105 ||65 ||1920 ||0803

The format of the machine-language cards produced as the output of a SOAP assembly will be discussed in Sec. 8-6.

7-11. SOAP Addresses

More often than not we will not use absolute addresses but one of two alternative designations, called symbolic and regional addresses. In these cases, the first column of the address field ***must not*** be left blank.

A symbolic address contains from one to five numeric or alphabetic characters of which the first is normally a letter. Q, A3, ETA, PI, or B7XQ4 are all acceptable symbolic addresses. The first time a symbol

is encountered in assembling a SOAP program, it is assigned an optimized storage location by the machine and that address is never used for another variable unless the machine is explicitly instructed to the contrary. Every time that symbol appears as either the Location, D-address, or I-address in a SOAP instruction, it is replaced by the same address in the machine-language deck.

Wherever possible, in succeeding sections, we will use symbolic addresses that are similar to the names of the variables involved. Thus, the storage location containing the variable x can be given the symbolic address X; and the instruction RAU X will bring the numerical value of the variable x to the upper accumulator.

Symbolic addresses are assigned to the first storage location available in keeping with certain rules of optimization. However, it is often desirable to reserve a set of *sequential* storage locations—for example, for storing arrays. Such blocks of locations are called "regions" and they are particularly useful in operations involving loops. A region is identified by a single letter; thus we can have a B region, an F region, a P region, etc.

The programmer himself selects the addresses to be set aside for each region. He specifies these addresses at the beginning of the SOAP deck by using a regional specification card. This card is identified by a pseudo-operation code in columns 48 to 50:

REG

Columns 41 to 47 are blank on this card. Column 51 contains the letter identifying the region; columns 52 to 55 contain the address of the first word in the region and columns 58 to 61 the address of the last word.

REG is called a "pseudo-operation" because it has no machine-language counterpart. It serves a purpose only during the SOAP assembly process when it identifies a block of addresses as a "region". We will encounter other pseudo-instructions in Sec. 8-12. Note that the Location field on an REG card is left blank.

Upon encountering the REG card in the assembly process, this specific block of addresses is reserved and the identifying letter is recorded.

Suppose we decide to define the L region as the block from 0832 to 0865. The regional specification card will be punched as follows:

COL	41	42	43 44- -47	48 49, 50	51 52- -55	56	57 58- -61	62	63- -72
	TYPE	SIGN	LOCATION	OPERATION CODE	DATA ADDRESS	TAG	INSTRUCTION ADDRESS	TAG	REMARKS
				R E G	L 0 8 3 2		0 8 6 5		

From then on, we may use *regional* addresses to specify Location, D-addresses, and I-addresses in the L region. A regional address consists of the alphabetic character defining the region, followed by four digits referring to an address *relative to the beginning of the region.* Thus the first word in the L region is given the regional address L0001. Similarly, L0005 is the regional address of the fifth word in the L region. When this address is encountered in the assembly, the machine refers to the information previously supplied on the regional specification card and will replace L0005 by an absolute address. In our example, the L region extends from 0832 to 0865; therefore, L0005 would be replaced by 0836.

7-12. Blank Addresses

Unless the normal flow of a program is interrupted, the Location of an instruction will correspond to the I-address of the preceding instruction. In a machine-language code (that has been optimized and therefore does not have sequential addresses), we might have, for example,

```
0701:  60 0503 0806
0806:  10 0732 0745
0745:  21 0516 0921
```

When using SOAP, we can use symbolic addresses, as described in the preceding section, so the above program might be written for machine assembly in the form

Location	Operation	D-address	I-address
LBQ	RAU	B	ST23
ST23	AUP	X3T	PENTA
PENTA	STU	REL6	Q

Note that we have repeated the symbolic I-address of each instruction as the Location of the next. This program would assemble correctly, but it is needlessly cluttered with symbols for I-addresses and Locations. More often than not, even in long complex programs,

I-addresses are repeated as the Locations in the instructions that immediately follow. To capitalize on this, and to cut down on unnecessary symbols, the SOAP scheme permits us to use blank addresses in the following way: either the D-address or the I-address of a SOAP instruction may be left blank provided the Location of the next instruction is also left blank. When the blank D- or I-address is encountered during assembly, the machine assigns to it an optimized absolute address and *also* assigns the same absolute address to the Location of the next instruction. (Except in the case of pseudo-instructions, it is an error to leave both the D- and I-address in the same instruction blank or to assign a symbol to the Location if the previous D- or I-address has been left blank.)

Our example can now be written

LOCATION	OPERATION	D-ADDRESS	I-ADDRESS
LBQ	RAU	B	
	AUP	X3T	
	STU	REL6	Q

We have assigned a symbolic address to the Location of the first instruction on the assumption that we want to refer to that instruction at some other point in the program. Furthermore, having assigned a symbolic address to both the D-address and the I-address of the last instruction, we cannot leave the Location of the next instruction blank.

(Blank addresses may be used for D-addresses as well as I-addresses because D-addresses are used in branching instructions to specify alternative path routes; see Sec. 8-9.)

In subsequent sections we will leave Locations and I-addresses blank unless there is a specific need to do otherwise.

7-13. 8000 ADDRESSES IN ADDITION AND SUBTRACTION

The quickest way to double a number is to add it to itself. If the number is already in the lower accumulator we can double it by adding the contents of the lower to the lower. We can accomplish this by using the "Add Lower" instruction with the addresses of the lower accumulator (8002) as the D-address:

ALO ||8002

(As usual, the number passes through the distributor on the way to the accumulator and replaces whatever was previously there.)

The upper accumulator can be used in the same way:

```
AUP ||8003
```

If we want to clear the lower accumulator but leave the upper unchanged, we must use the instruction

```
RAU ||8003
```

The "Reset Add Upper" instruction clears the entire accumulator, but the D-address, 8003, causes whatever was in the upper accumulator prior to clearing to be returned to the upper accumulator. Note that STU ||8003 is not a valid instruction.

If we want to erase the upper and preserve the lower, we can use the instruction

```
RAL ||8002
```

Numbers will be moved from upper to lower and vice versa by the instructions

```
RAL ||8003
```

and

```
RAU ||8002
```

As has been mentioned several times, the distributor contains the last number that passed through it on the way to or from the memory. It occasionally happens that the number needed in an arithmetic operation is already in the distributor, as well as somewhere in the memory. It is always more efficient to use the distributor address (8001) when calling for this information because the distributor is immediately accessible, whereas a location on the magnetic drum is not.

Therefore, to find $3x$, it is better to use the sequence

```
RAL  X
ALO ||8001
ALO ||8001
```

than

```
RAL  X
ALO  X
ALO  X
```

In the former case we take advantage of the fact that the RAL instruc-

tion brings x to the distributor on the way to the accumulator, and we call on x from the distributor in the succeeding instructions.

The distributor address can be used for the data in connection with any of the addition or subtraction codes. The programmer must be very careful, however, that the number he thinks is in the distributor has not been replaced by a previous instruction, such as STU.

7-14. The Coupled Accumulator in Addition and Subtraction

Although we speak of adding or subtracting in the upper or lower accumulator, the two parts are actually coupled in these operations; that is, there is a carry from the lower accumulator into the upper.

Consider the following situation. Suppose the upper accumulator is zero and the lower contains ten 9's:

$$+\ 00000\ 00000\ 99999\ 99999$$

If we add a 1 in the rightmost position, it will propagate a carry up the line and into the upper:

$$+\ 00000\ 00001\ 00000\ 00000$$

Similarly, if the upper is zero and the lower contains ten 9's, subtracting 1 from the rightmost position of the *upper* will produce the following result:

$$-\ 00000\ 00000\ 00000\ 00001$$

There is no way to keep the other half of the accumulator inactive. Actually, we must interpret addition and subtraction as follows: the instruction ALO, for example, adds to the 20-digit accumulator a 20-digit number of which the leftmost 10 digits are zero. Similarly, AUP adds to the 20-digit accumulator a 20-digit number of which the rightmost 10 digits are zero. Subtraction must be visualized in the same general way.

The programmer must be careful that he does not leave information in half of the accumulator under the mistaken impression that it will not affect his results in the other half. It is also unwise to leave information in part of the accumulator expecting to find it unchanged several instructions later. As a case in point, suppose that we have computed a number in the lower accumulator so that the coupled

accumulator contains

$$+\ 00000\ 00000\ 24171\ 21836$$

We will need this number at a later stage, but we want to do another computation in the meantime. What we should do is store the lower in some location of the memory, but you might expect that we can use the upper for the next series of operations and return to find the previous result in the lower. This is not the case, as we shall see.

Suppose that the operations in the upper add 2 and subtract 3 in the rightmost position. At successive stages the accumulator will look like this: first we add 2:

$$+\ 00000\ 00002\ 24171\ 21836$$

Then we subtract 3; but since both halves of the accumulator are coupled, the result will be

$$-\ 00000\ 00000\ 75828\ 78164$$

Thus instead of getting

$$-\ 00000\ 00001$$

in the upper, and preserving

$$+\ 24171\ 21836$$

in the lower, the upper contains all zeros, and the number in the lower is the complement of the number previously there.

To avoid difficulties of this kind, it is advisable not to use the accumulator for temporary storage of information and to begin every sequence of computations with an order that resets the entire accumulator.

7-15. Overflow during Addition and Subtraction

The sum of two numbers, each less than 1, can, of course, be greater than 1. Therefore, even though the addend and augend each contain 10 digits with decimal point at the far left, the sum may be an 11-digit number.

If the summation is performed in the lower accumulator and the sum exceeds 1, the carry will propagate into the upper accumulator. If the programmer overlooks this and stores only the lower accumulator, he will have an erroneous result.

In performing the computation

$$y = a + b - c - d + e$$

as long as the computation takes place in the *lower* accumulator, the programmer need not be concerned if $a + b$ or any other partial result exceeds 1, as long as y is less than 1. The carry will go over into the upper during the computation, but it will eventually be removed by subsequent operations.

This is not the case if the computation is carried out in the upper accumulator. If we add a to b in the upper and the sum exceeds 1, the carry to the left is lost. This condition is called "overflow" because the number has flowed out of the accumulator. When it occurs, the machine detects it and "sets" an internal overflow circuit. Depending upon whether the overflow switch on the console has been turned to Stop or Sense, the machine will either stop or go on with the overflow circuit set.

If overflow is not expected, the programmer would want an overflow to stop the machine to indicate an error. For example, if the computation of

$$y = a + b - c - d + e$$

is carried out in the upper and there is an overflow at any stage, the final result will be incorrect.

However, it is also possible to create an overflow intentionally and to use the overflow condition to be a signal of special circumstances. A particular branch instruction (Sec. 8-9) permits the programmer to select alternative paths depending upon whether the overflow circuit has been set.

7-16. Multiplication

The product of two 10-digit numbers, each less than 1 in absolute value, can be a 20-digit number, and it must also be less than 1.

The multiplication sequence consists of several steps. First the multiplier is brought to the *upper* accumulator, usually with the instruction RAU, using the location of the multiplier as the data address. The next instruction is "Multiply":

machine language:	19
SOAP:	MPY

in which the location of the multiplicand is the D-address. The 20-digit product will be built up in the coupled accumulator. If the significant digits in both multiplier and multiplicand are as far to the left as possible, the 10 most significant digits of the product will be in the upper accumulator and can be stored from there by the instruction STU. A complete multiplication sentence would be

```
RAU  A
MPY  B
STU  C
```

(If all 20 digits of the product are to be preserved, the lower accumulator must be stored in another location by using the instruction STL.)

⋆On the 650, the contents of the lower accumulator just prior to the execution of the MPY operation will be added to the part of the product built up in the upper accumulator. That is, if the contents of the upper and lower are initially

```
Upper:  A
Lower:  B
```

the instruction

```
MPY  X
```

will produce the result

$$AX + B$$

Although this is occasionally useful, it causes trouble for the beginner more often than not. He frequently neglects to clear the lower accumulator when setting up the multiplier in the upper. As a result, the products are not correctly computed.

⋆A particularly common error is made when computing

$$Y = A \cdot B \cdot C \cdot D$$

The sequence is usually begun correctly with the instructions

```
RAU  A
MPY  B
```

At this point the 20-digit product $A \cdot B$ fills the accumulator. We wish to use this product as the new multiplier, and since it is already in the upper accumulator, the beginner goes blithely on to another multiplication

```
MPY  C
```

What he has neglected to note, however, is that the part of the previous product that was located in the lower accumulator will be added to the high-order digits in the new product and produce an erroneous result.

⋆The correct way around this difficulty is to reset the accumulator, preserving only the contents of the upper half, by using the instruction

```
RAU ||8003
```

before giving another multiplication instruction.

⋆The entire sequence becomes

```
RAU  A
MPY  B
RAU  ||8003
MPY  C
RAU  ||8003
MPY  D
STU  PROD
```

7-17. Division

In division, a 20-digit dividend can be divided by a 10-digit divisor, to yield a 10-digit quotient plus a 10-digit remainder. To begin the division sequence, the 20-digit dividend is brought to the coupled accumulator (for example, by the instructions RAU and ALO). If the dividend contains only 10 digits, the lower accumulator is left blank.

There are two division instructions:

machine language:	64 and 14
SOAP:	DVR and DIV

In both cases, the D-address is the location of the divisor, and the I-address is, as usual, the location of the next instruction. Furthermore, in both cases, the 10-digit quotient appears in the *lower* accumulator after the division instruction has been executed. The difference between the two instructions is that in the case of DIV the remainder is preserved in the upper accumulator; but DVR clears the upper accumulator to zero and the remainder is destroyed.

Division with the remainder preserved is the only case in which the two halves of the accumulator can be of different signs, because

the remainder will have the sign of the dividend, which need not be the sign of the quotient.

At the end of the division operation, the divisor is in the distributor.

Whenever division is performed, care must be taken to assure that the quotient is less than 1 in absolute value; otherwise the machine will stop on overflow, *whether or not* the overflow switch is on Stop or Sense. We will return to this point when we discuss scaling.

7-18. Multiplication and Division by 10: Shifting

As we have pointed out, we ordinarily regard all factors involved in multiplication and division as being less than 1 in absolute value. The exception is multiplication or division by 10, which can be accomplished without using the normal multiplication or division instructions.

Consider a 20-digit number in the coupled accumulator:

$$+\ 00008\ 71244\ 32614\ 98825$$

Ten times this would be

$$+\ 00087\ 12443\ 26149\ 88250$$

In other words, multiplication by 10 can be achieved by shifting each digit one place to the left. Similarly, multiplication by 10^3 can be accomplished by shifting left 3 places.

The instruction

machine language:	35
SOAP:	SLT

is the left-shift operation. The data address serves a new purpose in this command; it is not the address of anything, so the term is really a misnomer. Instead, it specifies explicitly the number of places the information in the accumulator is to be shifted to the left. D-addresses for shift instructions are usually given in absolute form.

Thus

```
SLT  ||0005
```

will cause the contents of the accumulator to be shifted left 5 places. (The I-address, as usual, refers to the location of the next instruction.)

The D-address in shift orders is interpreted "modulo 10", which

means that a D-address of 0012 or 0022 or 0032 will be equivalent to a D-address of 0002.

In a left-shift operation, zeros are automatically supplied at the right-hand side; information shifted out of the accumulator at the left is lost. That is, after shifting

06251 83324 57009 01627

three places to the left, the accumulator will contain

51833 24570 09016 27000

and the three digits 062 will have been lost.

Shift operations, incidentally, do not affect the overflow circuit, even though information passes out of the accumulator to the left.

Division by positive powers of 10 (or equivalently, multiplication by negative powers of 10) can be achieved by shifting to the right. In this case, digits will be lost at the right end; that is, the effect of executing one right shift on the number

00000 00057 23481 19657

will be

00000 00005 72348 11965

and the 7 is lost.

The instruction

machine language:	30
SOAP:	SRT

is the command for right shift, and here again the D-address specifies the number of places the information in the accumulator is to be shifted.

When performing a right shift using the instruction SRT, the digits shifted out of the accumulator are lost and the remaining digits are unaffected. Frequently, however, we might want the last remaining digit to be "rounded", that is, to be increased by 1 if the last digit shifted out of the accumulator is 5 or greater. This can be accomplished by the instruction

machine language:	31
SOAP:	SRD

If SRD is used instead of SRT, the result of the previous shift example would be

00000 00005 72348 11966

because the digit lost was a 7. (This instruction will always round "upward" in absolute magnitude, rather than "even" as some people prefer, although the latter procedure could be programmed.)

It must be noted that a left shift of 5 followed by a right shift of 5 is *not* usually equivalent to no shift at all, because information once lost by shifting out of either end of the accumulator cannot be regained. For example, if the accumulator initially contains

02654 11182 96713 31148

and we shift left 4 and then right 4, the final contents of the accumulator will be

00004 11182 96713 31148

Shifting operations are used not only to multiply and divide by powers of 10, but also to move information around in the accumulator. This is an important function of the shift operations particularly in packing and unpacking information (Probs. 1 and 2, Chap. 8).

On binary machines, shift operations correspond to multiplication and division by 2 instead of 10.

Yet another shift operation plays a role in programming floating-point computations on a fixed-point machine. That is "Shift left and count":

machine language:	36
SOAP:	SCT

The purpose of this command is to shift the contents of the entire accumulator left until a digit other than zero is in the leftmost position, as in a normalized floating-point mantissa. The computer also counts the number of shifts necessary to achieve this and records the result in the rightmost two digits of the lower accumulator. This can then be used to adjust the exponent of a computed answer.

Thus, if the number initially in the accumulator is

00071 86225 19248 81417

the contents following the order

SCT ||0000

will be

71862 25192 48814 17003

where the last digit, 3, indicates that three left shifts were necessary to normalize the answer.

If no shifts are necessary, the rightmost two digits will be replaced by 00 and the corresponding two digits of the original number will be lost. If one shift is all that is necessary, the rightmost two digits will be replaced by 01, and one of the digits of the original number will be lost. Otherwise, no digits will be lost.

If, after ten shifts, there is still a zero in the leftmost position, the overflow circuit will be set. That is, the result of a shift and count executed on the number

00000 00000 00007 16223

will be

00007 16223 00000 00010

and the overflow circuit will be set.

The situation is somewhat more complex if the D-address of the SCT instruction is not zero. If there is no left shift, then, as before, the rightmost two digits will be replaced by 00. Other cases might best be described by an example. If the instruction

SCT ||0007

is given, and the accumulator contains

00000 51162 73846 11311

the result will be

51162 73846 11311 00008

The 8 in the rightmost digit is the sum of the number of left shifts (5) and the 10's complement of the D-address ($10 - 7 = 3$).

Here again, the *total* counted will not exceed 10. Any attempt to exceed it will set the overflow. That is,

SCT ||0004

applied to

00000 79741 62358 11003

will result in

07974 16235 81100 30010

and the overflow circuit will be set.

7-19. Scaling Variables and Constants

If the range of values of a variable is known, it is possible to multiply the variable by a constant to create a *scaled* variable with the

decimal point at the far left. For example, if x is known to vary from $+150$ to -80, the scaled variable X defined by the relation

$$X = 10^{-3} x$$

will lie between $+.150$ and $-.080$, and will therefore always be less than 1 in absolute value. X is the scaled variable corresponding to x and 10^{-3} is the "scale factor". On decimal machines, powers of 10 are usually used as scale factors; on binary machines, powers of 2 (because of the role played by shift operations in scaled equations).

Even though a number is already less than 1, it can be scaled so as to eliminate unnecessary zeros following the decimal. For example, if y varies from $+.0000720$ to $-.0000950$, we can define

$$Y = 10^{4} y$$

and Y will still be less than 1 in absolute value.

Constants must also be scaled so that they can be stored with the decimal located in the standard position. If an equation calls for multiplication by 34.8, we must rewrite it so as to perform multiplication by .348. In the next section we will see how all equations must be rewritten to involve only scaled variables and scaled constants.

7-20. Scaling Equations

Consider the equation

$$z = x + y$$

If it is known that x ranges from $+85$ to -115, the obvious scale factor is 10^{-3} and we define

$$X = 10^{-3} x$$

Similarly, if y always lies between $+1000$ and -800, we might define

$$Y = 10^{-4} y$$

The sum of x and y has a possible range of from $+1085$ to -915, and therefore, we define

$$Z = 10^{-4} z$$

Consider the original equation. We can write it so that every factor is always less than 1 by multiplying throughout by 10^{-4}.

$$10^{-4} z = 10^{-4} x + 10^{-4} y$$

Now we substitute the scaled variables for the original variables and obtain

$$Z = 10^{-1} X + Y$$

Note there is a power of 10 involved in the first term on the right because the scale factor for x is not the same as the scale factors for y and z. But as we have seen in Sec. 7-18, multiplication by powers of 10 is easily accomplished by shifting. Therefore, the program for our computation can be written as follows:

```
RAL X
SRT ||0001
ALO Y
STL Z
```

In the first step we bring X to the lower accumulator; in the second we multiply 10^{-1} by shifting; next we add Y; and finally we store the result.

Consider the computation

$$y = 3.82\, ax$$

If a is always less than 5 in absolute value, and x is always less than 20, y cannot exceed 382 and the following scaled variables are appropriate:

$$A = 10^{-1} a$$
$$X = 10^{-2} x$$
$$Y = 10^{-3} y$$

Multiplying the original equation by 10^{-3} we obtain, after rearranging the right-hand side,

$$10^{-3} y = 3.82 \cdot (10^{-1} a) \cdot (10^{-2} x)$$

or

$$Y = 3.82\, A\, X$$

Note, however, that the constant has not been scaled. Remember that we will actually only be able to perform multiplication by .382 instead of 3.82. We can get the equivalent result by rewriting the equation in the form

$$Y = 10 \cdot (.382) \cdot A \cdot X$$

Here again, multiplication by 10 becomes a shift operation in the program. Calling C the address of .382, we can write the program as follows:

```
RAU  A
MPY  X
RAU  ||8003
MPY  C
SLT  ||0001
STU  Y
```

(The necessity for the instruction RAU ||8003 before the second multiplication was explained in Sec. 7-16.)

It was emphasized in Sec. 7-17 that the factors in division must always be arranged so that the quotient is less than 1 in absolute value; otherwise the machine will stop on division overflow.

Suppose we want to compute

$$z = \frac{y}{x}$$

Although both y and x may be properly scaled, the quotient may exceed 1. Suppose we know that the largest value of y is .9000 and the *smallest* value of x is .0001; consequently the quotient may be as large as 9000.

If we define the scaled variables

$$\begin{aligned} \mathrm{X} &= x \\ \mathrm{Y} &= y \\ \mathrm{Z} &= 10^{-4} z \end{aligned}$$

we obtain the scaled equation

$$\mathrm{Z} = 10^{-4} \frac{\mathrm{Y}}{\mathrm{X}}$$

This can be programmed as follows:

```
RAU  Y
SRT  ||0004
DVR  X
STL  Z
```

By shifting the dividend to the right *before* division, we assure the fact that the quotient will be less than 1.

Care must be exercised in dealing with equations involving roots, as in this example:

$$y = a + \sqrt{x}$$

Let

$$\begin{aligned} \mathrm{A} &= 10^{-1}\, a \\ \mathrm{X} &= 10^{-1}\, x \\ \mathrm{Y} &= 10^{-1}\, y \end{aligned}$$

Multiplying the equation by 10^{-1} we obtain

$$10^{-1}\, y = 10^{-1}\, a + 10^{-1} \sqrt{x}$$

which can be rewritten

$$10^{-1}\, y = 10^{-1}\, a + \sqrt{10^{-2}\, x}$$

and finally becomes

$$\mathrm{Y} = \mathrm{A} + \sqrt{10^{-1}\, \mathrm{X}}$$

We have barely hinted at the complexity of scaling. After a few tussles with an obstinate decimal point, the programmer will easily understand why floating-point arithmetic is so popular.

8

A Continuation of Machine-language and SOAP Instructions

8-1. The Format of Data Cards

When the program has been assembled and is ready to run, we must punch the data on cards or tape. The 650 can read as many as ten words from a single card. Information from any one of the 80 columns can be brought to any one of these ten words, depending upon the way the control panel is wired. However, rather than have to wire a new control panel for every computation, computing centers usually wire a few standard boards permanently. It is desirable to use standard card formats whenever possible and to use programming, rather than board wiring, to redistribute the information in the memory.

One of the most popular card formats consists of 8 words of 10 digits each. The 80 columns of the card are punched as follows:

Columns 1–10: word 1
11–20: word 2
21–30: word 3
. . . . . .
71–80: word 8

X and Y punches are used to indicate negative and positive numbers, respectively, and are punched in columns 10, 20, . . . , 80 for the corresponding words.

Another common format contains ten 8-digit words. Since the standard 650 word size is 10 digits, the two remaining locations in each word must be filled with zeros (usually by permanent wiring on the standard board).

If the data consist of only a few significant digits, two or more numbers can be "packed" into a single 10-digit word. They will be transferred from the card to the machine as a 10-digit word, which must subsequently be "unpacked" by an appropriate program.

At least four card formats are available for each control board. Three are wired in by hand, and a fourth is permanently wired internally. Which of these formats will be called upon is determined by Y-punches on the card, as established by local convention. For example, a Y-punch in column 3 may mean that the card is to be read in the ten 8-digit word format.

The programmer should inquire at his own installation concerning standard boards and card formats.

The internally wired circuit is always the eight 10-digit word format. It is also activated by an appropriate Y-punch, for example, a Y-punch in column 1. Such a card is called a *load card* and it plays an important role in reading operations.

Cards of different formats may be mixed together provided each is properly identified by the presence or absence of Y-punches.

8-2. The Read Band

Information cannot be transferred from the card to any arbitrary location on the 650 drum; only certain addresses can be used. The reader will recall that the 2000-word drum on the 650 is subdivided into forty 50-word bands. Within each band there is a block of ten consecutive locations which will accept information from cards. These blocks are called "read bands" and contain the addresses

0001–0010
0051–0060
0101–0110

0151–0160

. . . . . .

1951–1960

*In actual operation,[1] the information passes from the card to a read buffer and from there to the read band. For the purposes of programming, however, we can neglect the read buffer entirely.

Each block of ten words in a read band acts as a unit. When the machine is instructed to read information from the card to, say, the band beginning with 1951, the information in all ten words of this band will be replaced by new information taken from the card. If fewer than ten words are read from the card (as, for example, when using the eight 10-digit word format), either the ninth and tenth words of the read band can be wired on the control panel so as to contain zeros, or they will receive invalid information. (The latter situation will lead to trouble if the program accidentally refers to these locations.)

It is a common programming error to assume that the ninth and tenth words of a read band are not affected when only eight words are read from the card. These two locations of a read band should never be used as permanent storage, because they will be erased whenever the read band is used.

It is customary to select one band to be used as the read band throughout a program. Frequently the band from 1951 to 1960 serves this purpose, and in SOAP programs it is convenient to call this the R region. In that case we must use a regional specification card (REG) to designate locations 1951 to 1960 as the R region, and thereafter the regional address R0001 will indicate absolute location 1951, etc.

8-3. The Read Instruction: Load Cards

The command that transfers information from a card to a read band is

machine language:	70
SOAP:	RCD or RD1

(We will employ RCD throughout.) The D-address of this instruction selects the particular read band. Any one of the 50 addresses in

[1] In this chapter, as in the previous one, parts that pertain to special aspects of the 650 are marked with a star and may be skipped if desired.

a band can be used as the D-address in an RCD instruction, and it will activate the corresponding read band. For example, information will be read into locations 1951 to 1960 on any of the following D-addresses: 1950, 1953, 1959, 1975, 1999, etc. Similarly the read band from 1601 to 1610 will accept information if the D-address of an RCD instruction lies between 1600 and 1649.

The I-address of an RCD instruction will be the address of the next instruction, *unless* the card that is read is a load card. In the latter case, the D-address will not only specify the read band to be used, but it will also be the address of the next instruction. RCD is a form of branch instruction: on a load card, the machine will go to the D-address for the next instruction; otherwise it will go to the I-address. For example,

RCD ||1956 ||1800

will cause the information on the card to be read into the read band from 1951 to 1960. The format will be determined by Y-punches on the card. If the card is a load card, the next instruction will be taken from 1956; if it is not, it will come from 1800.

This is the reason that any D-address in the 50-word band can be used in an RCD instruction. The read band will be activated by any address in the 50-word band, but the location of the next instruction when a load card is read can be specified arbitrarily.

8-4. Redistributing Information on the Drum

Although information can only be read from a card into a read band, we may want to move it immediately to another part of the memory. This can be accomplished by two new instructions:

machine language:	69 and 24
SOAP:	LDD and STD

called "Load Distributor" and "Store Distributor", respectively.

LDD brings information from any drum address, as specified by the D-address, to the distributor; STD takes the information from the distributor to any drum address, specified by the D-address of that instruction.

Suppose the following variables have been punched in eight 10-digit word format on a nonload card:

Word 1: X
Word 2: B3
Word 3: ETA
Word 4: Y
Word 5: GAMMA
Word 6: PIX
Word 7: MM2
Word 8: Z

and we want to send the data to the appropriate locations reserved for these variables. If the R region designates a read band, the following program would serve the purpose:

```
BOX1    RCD  R0001
        LDD  R0001
        STD  X
        LDD  R0002
        STD  B3
        LDD  R0003
        STD  ETA
        LDD  R0004
        STD  Y
        LDD  R0005
        STD  GAMMA
        LDD  R0006
        STD  PIX
        LDD  R0007
        STD  MM2
        LDD  R0008
        STD  Z          NEXT
```

The RCD instruction will cause the information to be read into the read band and the subsequent LDD-STD pairs will move the data around. (Since this card was not a load card the machine will go to the I-address of the RCD instruction, which was left blank in keeping with the SOAP convention.)

8-5. Initializing Card Reading from the Console

Every computation must begin by loading the program into the memory. Unlike the Bell scheme or FORTRANSIT, there is no

built-in reading routine in the basic machine. However, by pressing the appropriate buttons on the console, we can have the machine execute any instruction that is put in the storage-entry switches. If this is an RCD instruction, it will cause the first card to be read. Then, somehow, this card must provide, among other things, a method for reading in the next card.

8-6. The Single-word Loader

A simple way to load a program is to punch the instructions, one to a card, in the form of "single-word loaders". These are load cards containing not only information to be stored, but also a short program for moving it to an arbitrary location and provision for reading in the next card. The card is punched as follows:

Word 1: 69 1954 1953
Word 2: blank
Word 3: 24 (xxxx) 8000
Word 4: the data to be stored
Words 5 to 8 are blank

Signs are punched in columns 10, 30, and 40. The D-address of word 3 is the location to which the data in word 4 are to be sent.

A typical single-word loader to send the instruction 15 1218 1620 to location 0506 would contain

Word 1: 69 1954 1953
Word 3: 24 0506 8000
Word 4: 15 1218 1620

All other words would be blank.

To use a single-word loader, the storage-entry switches are set at

70 1951 (xxxx)

We will explain the role of the I-address shortly. By pressing the Program Reset and Program Start buttons on the console, we cause this instruction to be executed. Since the operation code is 70 (RCD), the first single-word loader will be read.

The D-address of the instruction in the storage-entry switches is an address in the 1950 band; therefore the read band 1951 to 1960 is activated and accepts information from the card. Since this card is a load card it is in the eight 10-digit word format. The only three words

that concern us, however, are the contents of the first, third, and fourth words in the read band:

```
1951:  69 1954 1953
1953:  24 0506 8000
1954:  15 1218 1620
```

The next instruction will be taken from the D-address in the storage-entry switches (1951) because the card was a load card; therefore the next instruction to be executed is

```
1951:  69 1954 1953
```

This is the first instruction of an LDD-STD pair which brings the word in 1954 to the distributor and then stores it in 0506 when the instruction

```
1953:  24 0506 8000
```

is executed.

Note that the instruction following this is to be taken from address 8000, which is the address of the storage-entry switches. Since these switches still contain the instruction

```
70 1951 (xxxx)
```

it will cause another card to be read. If this card is another single-word loader, the process will be repeated. Thus, the single-word card, together with the instruction in the storage-entry switches, forms a very tight loop that is executed every time a card is read.

Suppose that the entire program is punched on single-word loaders and, after the last program card has been read by the scheme we have just outlined, we want to begin the execution of the program. Let us say that the first instruction of the program is in 0101. This should be placed in the storage-entry switches as the I-address of the RCD instruction. The complete instruction becomes

```
70 1951 0101
```

Suppose we put a blank card after the last single-word loader. When the last single-word loader sends control back to 8000, the instruction there will cause the blank card to be read; but since a blank card cannot be a load card, the machine will go to the I-address of the instruction in 8000, and this will begin the program.

The blank card in this case acts just like a transfer card in the Bell scheme (Sec. 2-15).

Data cards that are to be read in during the computation follow the blank card.

We are now in a position to understand the output of a regular SOAP assembly. Every input card results in one output card and columns 41 to 80 of the input card are exactly duplicated on the output card. The machine-language version of every *regular* SOAP instruction is punched in columns 1 to 40 of the output card in the form of a *single-word loader*, and it is fed into the machine by the method described earlier in this section. The output of pseudo-instruction cards is described in Prob. 3 at the end of this chapter.

8-7. Other Loading Routines

Single-word loaders are very convenient, particularly when testing a program, because instructions can be removed or replaced very easily. But they are not very practical for loading large programs, because card reading is a relatively slow process. Therefore, other loading routines are used in which several instructions are punched on one card. One such procedure involves seven instructions on one card. Up to seven words are punched in columns 11 to 80 and will be stored in successive locations in the memory according to instructions punched in columns 7 to 10. Such a "seven-per-card loader", unlike a single-word loader, does not itself contain the program for moving the data from the read band to another location. Therefore the deck of seven-per-card loaders must be preceded by a short program deck that controls the reading and redistribution of information.

Some installations use a modification of SOAP in which the output of the assembly consists of load cards with 5 instructions on each.

Further details about loading programs may be found in local computing center regulations.

8-8. Punch Bands and the Punch Instruction

Just as there are special read bands, there are also special punch bands on the 650. There are ten locations in a punch band and the last two digits of the addresses run from 27 to 36, or from 77 to 86. Thus, the first punch band on the drum contains addresses 0027 to 0036; and the last, addresses from 1977 to 1986.

Here again, the band acts as a unit and is activated by any address in the 50-word band. Thus any address from 0050 to 0099 will activate the punch band from 0077 to 0086.

Three different card formats can easily be punched by wiring one control panel. Which format will be used depends upon the contents of the last word in the punch band that is activated. Just as Y-punches are used on input cards to select the format, 8's are used in the tenth word of the punch band to control the output format.

There are no internally wired punch formats.

Here again we can select one punch band for use throughout the program and denote it perhaps as the P region by using an REG card.

The instruction that activates the output unit is

machine language:	71
SOAP:	PCH or WR1

(We will use PCH throughout.)

There is no particular advantage in using any address other than P0001 as the data address of the PCH instruction. This is not a branch instruction, like RCD, and the I-address always specifies the address of the next instruction.

If the information has to be rearranged for punching, we again use LDD-STD pairs to do this. For example, to punch X, AA2, and BETA from the first three words of the P band, we can write

```
B6    LDD  X
      STD  P0001
      LDD  AA2
      STD  P0002
      LDD  BETA
      STD  P0003
      PCH  P0001      B7
```

8-9. Branching Codes

All branching codes on the 650 have the following characteristic: when the condition that appears in the name of the instruction is satisfied, the D-address specifies the location of the next instruction; otherwise, it is the I-address as in other instructions.

For example, consider the command "Branch on Minus":

machine language: 46
SOAP: BMI

The instruction

BMI L Q

is interpreted this way: "If there is a negative number in the accumulator, go to L for the next instruction; otherwise, go to Q".

By the same token, the RCD instruction could be called "Branch on Load Card".

The following instructions involve branching based on the number in the accumulator:

SOAP	Machine language	Name
NZU	44	Branch on Nonzero Upper
NZE	45	Branch on Nonzero
BMI	46	Branch on Minus

As you see, it is possible to test the upper accumulator (NZU) or the entire accumulator (NZE) for zero, but not the lower alone.

The instruction "Branch on Overflow,"

machine language: 47
SOAP: BOV

will send the machine to the D-address for the next instruction if the overflow circuit has been set by some previous operation. It does not matter how far back in the program this occurred; the overflow circuit retains a record of an overflow and responds to the BOV instruction. *Once the overflow circuit is tested by a BOV instruction, it returns to the normal condition before overflow, and can only be set again by another overflow.*

⋆A special set of branching instructions on the 650 examines individual digits in the distributor; the machine will branch to the D-address if there is an 8 in the specified location; it goes on its normal way if there is a 9 there, and stops if the digit is neither an 8 nor a 9. For the purpose of this instruction, the digits of the distributor are numbered, right to left, from 1 to 10.

⋆As a typical example, consider: "Branch on 8 in distributor digit 3":

machine language: 93
SOAP: BD3

If the distributor contains

xxxxxxx8xx

(where the digits marked x are immaterial), the machine will go to the D-address for the next instruction. If the distributor contains

xxxxxxx9xx

the machine will go to the I-address; and if it contains

xxxxxxx5xx

it will stop.

★The complete set of branch-on-digit codes is as follows:

SOAP	Machine language	Name
BD1	91	Branch on 8 in digit 1
BD2	92	Branch on 8 in digit 2
BD3	93	Branch on 8 in digit 3
BD4	94	Branch on 8 in digit 4
BD5	95	Branch on 8 in digit 5
BD6	96	Branch on 8 in digit 6
BD7	97	Branch on 8 in digit 7
BD8	98	Branch on 8 in digit 8
BD9	99	Branch on 8 in digit 9
BD0	90	Branch on 8 in digit 10

Note that BD0 is a branch on digit 10, not the nonexistent digit number 0.

★The previous codes can be used to select alternative paths of computation depending upon the format of the input card. The Y-punch that selects the input format can also be used to control a circuit on the control board that automatically stores 8's and 9's in one of the words in the read band. At some later time, this word can be brought to the distributor and tested using a "Branch on 8" instruction. Depending upon the result of this test, the machine will follow alternative paths in the program.

D- and I-addresses in branch instructions may be treated in one of three ways when using SOAP: they may both be explicitly specified, or either one may be left blank. As pointed out in Sec. 7-12, if either one is left blank, the Location of the next instruction must also be

To modify, we store the incrementing constant and add it to the counter every time we go through the loop:

```
UPDA RAL CTR
     ALO INC
     STL CTR  TEST
INC  ||00  ||0000 ||0001
```

While the incremented counter is still in the lower accumulator, we can test it against the maximum value desired (for example, 15), and if it equals this, we leave the loop; otherwise we go back to EXEC:

```
TEST SLO  MAX
     BMI  EXEC LEAVE
MAX  ||00  ||0000  ||0015
```

When a loop involves address modification, it is convenient to use the regional form in SOAP for the addresses that are to be modified. In that way, the locations will lie sequentially in storage and we can proceed through them, one at a time, by adding 0001 to the D- or I-addresses as desired.

If the D-address is to be modified, the constant

```
DMOD ||00 ||0001 ||0000
```

can be stored and used for incrementing; if the I-address is the one that will be altered, the constant

```
IMOD ||00 | 0000 ||0001
```

serves the corresponding purpose.

Address modification is accomplished by bringing the instruction involved to the accumulator and adding the increment to it, just as though the instruction were any other piece of data.

Suppose we want to increment the D-address of the instruction

Loc.	Op.	D-address	I-address
BLOV	ALO	R0001	DENT

Assuming that the incrementing constant DMOD has been stored, we write

```
IC RAL BLOV
   ALO DMOD
   STL BLOV  DL
```

This will modify the instruction and store the new version in the same memory location. The next time through the loop, the instruction performed will be

```
BLOV ALO R0002 DENT
```

Of course, since the instruction has actually been changed in the memory, it must be initialized before we enter the loop again. Initialization is accomplished by storing the correct initial form of the instruction as a constant; then an LDD-STD pair is used to bring it to the proper location. In this case we want to store the initial form of the instruction in a location we will call IBL and we can bring it to BLOV by these instructions:

```
BINI LDD IBL
     STD BLOV EXTR
IBL  ALO R0001 DENT
```

If this method of initialization is used, every instruction that is modified must have an explicit Location, D-address, and I-address; no blank addresses are permissible. The reader should be able to understand why.

An alternative method of initializing instructions makes use of two new commands: "Store Data Address" and "Store Instruction Address":

machine language:	22 and 23
SOAP:	SDA and SIA

SDA modifies the instruction in the distributor by replacing the 4 digits of the D-address part by the corresponding 4 digits of the number in the lower accumulator. The modified instruction is then stored in the location specified by the D-address *of the SDA instruction.*

Suppose the distributor contains

```
24 1475 1922
```

and the lower accumulator contains

```
19 1871 1511
```

(signs are immaterial). If the SDA instruction

```
22 0540 1695
```

is executed, the contents of the distributor will become

```
24 1871 1922
```

and that number will also be stored in 0540. The lower accumulator will not be changed.

If the SIA instruction were given instead,

23 0540 1695

the distributor would contain

24 1475 1511

and it would also be stored in 0540.

Initialization for the previous problem can also be accomplished by using the program

```
IC   RAL  MD
     LDD  BLOV
     SDA  BLOV  EXTR
MD   ||00  R0001  ||0000
```

8-12. More about Pseudo-instructions in SOAP

We have already discussed the instruction REG. This is not a regular machine-language instruction; its sole purpose is to inform the machine, during the SOAP assembly, that a block of addresses in the memory is to be designated a "region". We have therefore distinguished this from other commands by calling it a pseudo-instruction.

In the paragraphs that follow, we will see that it is convenient to be able to communicate other information to the machine during SOAP assembly and we will therefore introduce quite a few new pseudo-instructions. The reader should bear in mind that pseudo-instructions have no machine-language counterparts. They function only during the assembly process and do not become a part of the machine-language program that is used to carry out the actual computation.

SOAP instruction cards containing pseudo-operations have blank Location fields and may have blank D- or I-addresses, depending upon the operation. In the machine-language deck produced by the assembly process, a pseudo-instruction card produces a load card of a different form from that produced by regular instructions (Prob. 3 at the end of this chapter).

8-13. The Pseudo-instructions BLR and BLA

We have already mentioned in Bell programming that there are advantages in using separate blocks of storage locations for each box of the flow diagram. These advantages continue to apply when programming in machine language or SOAP. In the latter case, unless special provision is made, the machine will not restrict the locations assigned during assembly, and the instructions and data may be scattered all over the memory in optimized locations.

Two pseudo-instructions are used to permit us to restrict the availability of blocks of addresses during assembly. One of these is "Block Reservation":

BLR

As with any other pseudo-instruction, this 3-letter code is punched in the Operation Code field of a SOAP instruction card. The D- and I-addresses specify the first and last addresses of the block which is to be reserved, *in absolute form*.

Thus, when the card

BLR ||0500 ||0799

is encountered during the assembly process, the machine will not assign addresses between 0500 and 0799 to symbols encountered from that point on.

If the box we are assembling is to be restricted to the block between 1630 and 1725, inclusive, we must use two BLR cards preceding the instruction cards: one to reserve the drum from 0000 to 1629, and one to reserve it from 1726 to 1999.

If we want to assemble two boxes, one right after the other, but each to occupy a different part of the memory, we can open a new region of the drum after the first box has been assembled, and close the other. To open a block that has previously been reserved, we use the pseudo-instruction "Block Availability":

BLA

again specifying the bounds of the region in the D- and I-address fields in absolute form.

If we had previously restricted the assembly to the band between 1630 and 1725 and we now want to restrict it to the band between

0515 and 0830, we use two cards: a BLR to close one block, and a BLA to open the other:

```
BLR  ||1630  ||1725
BLA  ||0515  ||0830
```

Earlier in this section we showed how to restrict the assembly to the block from 1630 to 1725 with two BLR cards. We can accomplish the same result by using the combination BLR-BLA (in that order) as follows:

```
BLR  ||0000  ||1999
BLA  ||1630  ||1725
```

The first card reserves the whole drum and the second opens only the desired block. Note that the BLR card is essential because at the beginning of a SOAP assembly the entire drum is available for assignment. This procedure is less efficient than the combination of two BLR cards because it takes quite a bit longer to reserve the entire drum than only a small part of it.

It should be pointed out that a program that is restricted to a block in the memory will not be optimized as well as if it had been permitted to use the entire drum; but in most cases the advantages from the point of view of ease of locating errors and replacing blocks outweigh this disadvantage.

8-14. The Availability Table

After a set of instructions has been assembled within a block of storage locations, there may still be addresses within the block that have not been used. Because of the optimization rules, these will be scattered here and there in the block. However, it is important to know precisely where they are since we may need them for corrections or additional instructions if the memory is crowded. Therefore, immediately after a block has been assembled, we instruct the machine to produce a record of the locations that have been used and those that are still available. This is called an "availability deck" and it is produced by a card containing the pseudo-operation "Punch Availability Table":

```
PAT
```

The Location, D-, and I-addresses are blank.

As soon as this card is encountered, the machine will produce the availability deck, which is a pack of fifty cards, each containing a Y-punch in column 41 (the Type column). When listed on the tabulator, these cards give an easily interpreted picture of the drum, with 0's indicating locations that are no longer available, and 1's indicating those that are.

A further property of this deck is that it can be reloaded prior to another assembly and it will re-create the availability conditions that existed when the availability table was first produced. Thus, if we want to pack another box of the flow chart into a block already partly occupied, we need only load the availability deck for the first box before assembling the second and it will restrict further assignments to locations that were previously unoccupied.

8-15. One Symbolic Address with Two Meanings: The Pseudo-operation HED

It is sometimes desirable to use the same symbol to mean different things in different blocks of the program. This happens particularly often when using several subroutines from the library; the symbol X may be used in each one of them to represent a different variable. We usually want to preserve each of them as a distinct quantity assigned to a separate address in storage; otherwise we would destroy the X of one block when we store the X of another.

One way to overcome this is to go through every block to be sure that the same symbol is never used for two different variables; but this is a tedious job and errors often slip through. SOAP provides a more suitable alternative. If we are assembling several boxes at the same time, we precede each with a card punched with the pseudo-operation "Head":

HED

The first column of the D-address field of this card is punched with any valid character; this becomes the "heading" of this block, and it is in force until a new HED card is encountered.

Following a HED card, every symbolic address encountered that contains *fewer than five characters* is automatically expanded to five characters, of which the fifth is the heading. Thus in a block headed by the card

OP. D-ADDRESS
HED B

the symbol

BETA

will be treated as though it were the symbol

BETAB

and

ETA

becomes the new symbol

ETA||B

(where the fourth character is a blank). Similarly, in a block beginning

OP. D-ADDRESS
HED 7

the variable

A

becomes

A|||7

containing three blanks.

Note that the 5-character symbol

AGRIP

does not become

AGRI7 or AGRIP7

but remains unchanged because 5-character symbols are never headed. The advantage of this will be clear later on.

Now there is no danger of giving the same storage location to X in box 5 and X in box 9. If the first is headed with a 5 and the second with a 9, they are treated as two separate symbols

X|||5

and

X|||9

8-16. TWO SYMBOLS WITH THE SAME ABSOLUTE ADDRESS: THE PSEUDO-OPERATION EQU

It may occur that we want two distinct symbolic addresses to be assigned to the same absolute memory address. This might be done

for mnemonic purposes; or it might be necessary when the memory is tightly packed and there is no more room for a new variable. If the variable ETA can be erased by the time the variable Q has to be stored, we can store Q where ETA was. To inform the machine that ETA and Q are to be assigned to the same storage location, we use a pseudo-operation "Equivalence":

EQU

The two symbols that are to be equivalent are then specified as the D- and I-addresses on this card. Thus

Op.	D-address	I-address
EQU	ETA	Q

would accomplish the result we have just described.

Another situation in which an EQU card would be appropriate is this: suppose we want to perform the shift operation

SLT ||0004

but for one reason or another we want to use the symbolic address M instead of the absolute address 0004. We can write

SLT M

provided we have specified earlier the equivalence of M and 0004:

EQU M ||0004

(Note that the address 0004, as used in this shift operation, is not to be interpreted as an address in the memory; it only tells how many places to shift. We therefore do not want to set aside the storage location 0004 for the variable M. Compare this with the SYN card described in the next section.)

8-17. A Symbol with a Preassigned Absolute Memory Address: The Pseudo-operation SYN

A card punched with the pseudo-operation "Synonym"

SYN

contains a symbolic D-address and an absolute drum location for the I-address. Thereafter, whenever that symbolic address is encountered during assembly, it will be replaced by the specified absolute address;

furthermore, the specified drum address will be reserved for that variable.

There are several different circumstances when one might use a SYN card. For example, we might want to be sure that the variables PN, R3, and BNQ are located in a punch band so that they don't have to be moved before output. We might therefore use three SYN cards as follows:

OP.	D-ADDRESS	I-ADDRESS
SYN	PN	‖1826
SYN	R3	‖1827
SYN	BNQ	‖1828

Then again, it might be convenient to assign the location at which the program begins to some absolute location, say 0100, so that we know in advance that we are to set the storage-entry switches to

70 1951 0100

(see Sec. 8-6). To do this we could use the SYN card

SYN BEGIN ‖0100

BEGIN would then be the Location specified on the SOAP card containing the first instruction of the program.

A particularly important use of the SYN card is encountered under the following circumstances. In a particular block let us say we introduced the symbolic variable SIGMA and during the assembly the machine assigned it to the absolute address 1948. At some later time we want to assemble another block of the program, and in this new block we again refer to the same variable SIGMA. In order to be sure that the same address is used for SIGMA in both blocks, we precede the assembly deck of the second block with the card

SYN SIGMA ‖1948

Note that if a header card is used, all symbols that are to be carried from block to block must contain five characters; otherwise they will be given a heading and will be treated as different variables in each block.

8-18. The Two Forms of Subroutines for Use with SOAP

When programming in the basic machine language or SOAP, one depends heavily upon subroutines to relieve the burden of having to

program everything in detail. The basic codes perform only the most elementary operations, but the program library contains subroutines for a very wide variety of special computations.

Library programs for use with SOAP may be in one of two forms: "symbolic" and "relocatable". A symbolic library program consists of a deck of program cards of the same form as we have been discussing. They are assembled along with all the other parts of the program and the machine assigns locations to the symbols as it goes along.

Subroutines in the relocatable form, however, are used in a different way. Relocatable subroutines have been written so as to occupy the storage locations with the lowest addresses, compatible with good optimization. When the program is assembled, however, the entire subroutine can be moved bodily so as to occupy any region of the memory. Thus, although the original routine occupied locations between 0000 and 0185, we can "translate" it so that it will occupy locations between 1100 and 1285. (To preserve optimization, it is frequently necessary to restrict the translation to a multiple of 50 or 100 locations.)

We indicate the amount of translation by preceding the deck of library cards by a card punched with the pseudo-operation "Relocate":

REL

The D-address specifies, in absolute form, the amount of the translation. Thus

REL ||0800

will cause the entire subroutine to be stored 800 locations higher in the memory.

All cards of the relocatable program deck are distinguished by a 2 in column 41. This tells the machine during assembly that all addresses on that card are to be translated by the amount specified on the preceding REL card, unless there is a specific indication to the contrary.

The Location on a Type 2 card must be given in absolute form. It will always be translated. Both the D- and I-addresses are always specified; no blank addresses are used. If the D- or I-address is given in absolute form it will be translated; but if the absolute address is preceded by an F, it will be considered a "fixed" address and will *not* be translated. As an example of a case when we would not want an address translated, consider the shift operation

Loc.	Op.	D-address	I-address
‖0117	SRT	F0007	‖0145

The D-address indicates there is to be a right shift of 7 places, and this is always to be 7 places no matter how the subroutine is translated. By putting an F in front of 0007, we assure that this number will be stored without alteration.

D- or I-addresses on Type 2 cards can also be in symbolic form to provide linkage with other parts of the program. In that case the addresses will be optimized and assigned in the usual way.

The relocatable library programs usually precede all other parts of the program during assembly. Each subroutine is preceded by its own REL card specifying the amount of translation. As the program is assembled, the addresses *after translation* are reserved and will not be assigned to any other variables during the remainder of the assembly.

It is convenient to have block reservation and equivalence statements similar to BLR and EQU, but with addresses that can be translated. The corresponding pseudo-operations are

RBR and REQ

The absolute addresses that are punched on these cards will be modified according to the preceding REL card. Thus, if we have indicated a translation of 250 locations by the card

REL ‖0250

the instruction

RBR ‖0810 ‖0830

will reserve the block between

0810 + 0250 = 1060

and 0830 + 0250 = 1080

There are two advantages to using subroutines in the relocatable form. For one thing, they are usually very tightly optimized, more so than they would be if optimized by machine with the rest of the program. For another, the assembly of Type 2 cards takes place very rapidly because the machine only needs to translate the addresses and it does not have to search for available optimum locations. For these reasons, the longest subroutines are available in relocatable form and should be used that way.

One disadvantage of the relocatable form is that it is harder to understand what is going on if anything goes wrong or if it is desirable to make slight alterations in the procedure. In that case, the symbolic form has the decided advantage.

8-19. Subroutine Entry and Exit

Regardless of whether the subroutine comes from the library or is specially written for a computation, the entry and exit are usually in a standard form. We have previously discussed several times the necessity for a general exit procedure when using a subroutine so that the flow can proceed to any specified point in the program. This problem has a particularly simple solution in the machine language of the 650. Before entering the subroutine, we put the instruction that is to follow the subroutine in the distributor. The first instruction of the subroutine stores the distributor in the last word of the subroutine. Therefore, when the subroutine is completed we automatically execute the desired instruction.

As an example, suppose we have a subroutine that begins in symbolic location SUB3. The first instruction of the subroutine must be

```
SUB3 STD EXIT
```

where the I-address is immaterial. In this case the last instruction of the subroutine will have the symbolic Location EXIT. The original content of EXIT is immaterial, but each time the subroutine is executed EXIT is filled with the instruction that is to follow the subroutine.

If we are to execute the subroutine before going to box 4, we can enter the subroutine as follows:

```
LDD BOX4 SUB3
```

This will bring the first instruction of box 4 to the distributor and transfer control to SUB3, which, in turn, will take that instruction from the distributor to EXIT, where it will be ready when needed at the end of the subroutine.

Note that the LDD instruction involved in the transfer to a subroutine can have a blank D-address provided the instruction to follow the subroutine comes immediately after the LDD instruction and has a blank Location. That is, we can write either

```
        LDD  BOX4  SUB3
BOX4    RAU  Q     STAA
```

or

```
        LDD        SUB3
        RAU  Q     STAA
```

In the latter case, the blank D-address of the LDD instruction and the Location of the RAU instruction that follows will automatically be given the same address.

8-20. No Operation and Halt

The command "No Operation"

machine language:	00
SOAP:	NOP

will do nothing except send the machine to the I-address for the next instruction. The D-address is ignored.

The command "Halt"

machine language:	01
SOAP:	HLT

will cause the machine to stop *if the Programmed Stop Switch is set at Stop*. When the Program Start button is pressed, the machine will go on to the I-address for the next instruction. If the Programmed Stop Switch is on Run, HLT is equivalent to NOP.

HLT can be used as a debugging device to serve as a landmark at crucial points in the program. It can also be used to indicate when the program has strayed from the intended path. By using a special card, prior to loading the program deck, it is possible to put the instruction

01 xxxx 9999

in every storage location, where the D-address will be the location in which the instruction is stored. That is, 0810 will contain

01 0810 9999

(This instruction will always stop the machine because 9999 is not a valid address.) When the regular program is loaded, the HLT instruction will be replaced wherever new information is stored, but it

will remain in all other locations. If control is erroneously transferred to one of these locations, the machine will stop and the program register will indicate the location of the HLT instruction.

Note that clearing the memory to zero will not have the same effect because

00 0000 0000

means "No operation; go to 0000 for the next instruction".

8-21. Some General Procedures for SOAP Assemblies

By using some care, SOAP assemblies can be carried out with a minimum of error and with very good optimization.

The assembly deck usually begins with any relocatable library subroutines that are to be used, each preceded by its own REL card specifying the amount of translation.

The SYN, REG, and EQU cards follow. The SYN cards contain the absolute addresses of any symbols that the programmer wishes to specify, including those that were assigned by the machine during the assembly of other blocks of the program. The REG cards specify blocks of storage addresses in which regional addresses are to be used; and EQU cards are used if different symbols are to have the same absolute locations.

After all these come the individual boxes of the flow diagram that are to be assembled, each preceded by BLA and BLR cards or availability decks from previous assemblies, limiting the parts of the memory that are to be used, and by an HED card, if heading is desired. Within the deck for a particular box of the diagram, the instruction cards usually precede cards containing constants.

The boxes are not necessarily assembled in the order of execution, but rather in the order of *frequency* of execution, so that the parts that are performed most often are assembled first and hence best optimized. (Remember that the storage location assigned to a variable will be optimized with respect to the first appearance of that symbol during assembly.) The final flow of the computation will be controlled by explicitly specified I-addresses and Locations.

At the end of the deck we put a PAT card to produce an availability table. This should be done whether you expect to use it or not

because it cannot be produced without reassembly if it is needed unexpectedly.

Before actual assembly, the deck should be printed on the tabulator and examined for obvious errors, such as blank D- or I-addresses that are not followed by a blank Location.

When the deck has been assembled and is ready to run, it is wise to load first the special card that puts the HLT instruction

01 xxxx 9999

in every location, so that the machine will stop if the program erroneously leaves the proper path. A few additional remarks about debugging will be made in the next section.

When errors are detected, they can be corrected either directly from the console or by loading single-word load cards. Unless there are very extensive errors, it is usually not necessary to reassemble the entire program; but if reassembly is necessary, the output of the previous assembly, plus corrections, can be used as the input of the new assembly because columns 1 to 40 are ignored during assembly and columns 41 to 80 are just a duplicate of the deck previously assembled.

Once the program is running correctly, it can be punched on seven-per-card loaders directly from the drum by using a standard punch-out routine, and these cards can be used for running the program in the future instead of the slow single-word loaders.

8-22. Debugging

All the debugging techniques described in Chap. 5 can be used in testing machine-language programs. There are standard routines available for memory dumps, tracing, address searches, etc. Some of these routines are available in a form to be assembled by SOAP along with the rest of a program; others must be used in special regions of the memory. To use routines of the latter type, we must plan to leave enough storage free after assembly to accommodate the debugging routines. It is sometimes a local convention to reserve the highest addresses for debugging, but every programmer should inquire at his own installation to be certain.

Once again we must emphasize that it is important to be thoroughly prepared for debugging before getting time on the machine.

8-23. Omissions

We have purposely omitted a discussion of 650 commands for table look-up, floating-point arithmetic, built-in loop boxes, core storage, and tapes. With the exception of table look-up, these are all auxiliary features and are not part of the basic 650. While they are beyond the scope of this introduction to programming techniques, the fledgling programmer should not hesitate to learn about them if he thinks they will assist him in his calculations and if they are available on the machine at his disposal.

Problems for Chapter 8

1. Fifteen 5-digit words (all positive) are punched from columns 6 to 80 on a load card. Write a routine for "unpacking" them and storing them individually in successive locations beginning with the address punched in columns 2 to 5.

2. Fifteen 4-digit words are punched in 5-digit fields from columns 6 to 80 on a load card. The first digit of each field is an 8 or 9 depending upon whether the number is positive or negative. Write a routine for unpacking and storing each word individually with proper sign in successive locations beginning with the address punched in columns 2 to 5.

3. A card containing a pseudo-instruction produces a corresponding load card during SOAP assembly. This is not a *single-word* loader, but only contains 00 0000 8000 in the first ten columns. What happens when this card is read with the storage-entry switches set at 70 1951 xxxx?

4. At the end of Sec. 8-11, we described a small program for initializing an address. Why is the following version wrong?

```
IV   LDD  BLOV
     RAL  MD
     SDA  BLOV EXTR
MD   ||00  R0001 ||0000
```

5. Write a program for punching the contents of every read band on the drum, 5 words to a card, and properly identified as to origin.

6. Write a subroutine for adding two 20-digit numbers. The 20 digits of one are to be stored in A0001 and A0002, the 20 digits of the other in B0001 and B0002, the 20 digits of the result in C0001 and C0002. If overflow occurs, replace the sum with twenty 9's.

7. Write a subroutine for performing floating-point addition using only the basic 650 codes. Assume the normalization of the Bell scheme.

8. Write a subroutine to test if a matrix stored in the machine is symmetrical. The subroutine should punch any pairs of elements which do not obey the relation $a_{ij} = a_{ji}$. The matrix can be of arbitrary size from 3×3 to 30×30 and the size is to be specified, prior to entering the subroutine, by a code word stored in symbolic location CWD. Write a program using the subroutine to test it and debug by the methods of Chap. 5.

Glossary of Terms

As in any technical field, programming has a vocabulary all its own. What is more, some terms may mean different things to different programmers; therefore, the best we can hope to do is to define the terms as they are used in this book. In any event, this Glossary is designed as an aid to the beginner rather than as a rigorous lexicon.

accumulator That part of the arithmetic unit in which sums, products, etc., are built up. (Chaps. 7 and 8)

address A number or symbol used to denote the location of information in the machine.

analog computer A device which performs computations in terms of continuous quantities as opposed to the discrete digits used in a digital computer.

arithmetic unit That part of the computer in which the arithmetic computations (as well as others) are performed. On the IBM 650, the arithmetic unit consists of the upper and lower accumulator and the distributor.

assemble (verb) To use the machine to translate a program from a language such as FORTRANSIT or SOAP to the basic language of the machine.

automatic programming A procedure in which the machine is used to translate a program written in a more or less algebraic language (such as FORTRANSIT) to the machine language, removing

much of the routine bookkeeping of programming from the responsibility of the programmer. (Chap. 6)

base (of a number system) See Sec. 2-1.

binary system A number system with base 2.

bit One binary digit.

B-line See *loop box*.

block (noun) A group of successive locations in the memory.

branch instruction An instruction that permits the programmer to choose between alternative paths of computation depending upon the circumstances encountered by the machine during the execution of the program. Synonym: transfer instruction. (Chap. 3)

buffer A place in which information from one part of the machine can be temporarily retained until another part of the machine is ready for it.

clear (verb) To replace the information located in a part of the machine by zeros.

code The basic symbols in terms of which a program must be expressed in order to be accepted by the machine.

code check The act of testing a program on the machine or the time set aside for testing programs on the machine schedule.

command See *instruction*.

console The display of lights, buttons, and switches from which the operation of the machine can be monitored.

control panel Used on some machines (such as the IBM 650) to vary the format in which the computer will accept information or produce results.

control unit The part of the machine which "directs traffic" during a computation. It examines each instruction and supervises its performance before going on to the next.

debug (verb) To find the errors ("bugs") in a program.

digital computer A device that performs computations in terms of discrete numbers, as contrasted to an analog computer.

distributor A part of the arithmetic unit on the IBM 650 through which all information must pass on its way to or from the drum.

dump (verb) To produce a record of all, or a large part, of the memory.

fixed point Arithmetic in which the numbers are written with a

fixed location of the decimal point. Antonym: floating point. (Sec. 2-3)

floating point Arithmetic in which a number is specified in two parts—a mantissa, containing the significant digits, and an exponent, denoting the power of the base by which the mantissa is to be multiplied. Antonym: fixed point. (Sec. 2-3)

flow diagram A boxed outline of the course of a computation, showing the order in which the major steps are to be performed, decisions that must be made, etc.

hexadecimal system A number system with base 16. Synonym: sexadecimal.

initialize (verb) To restore the initial conditions before entering a loop. (Sec. 3-11)

input Used to denote either the part of the machine that accepts information or the information accepted by the machine.

instruction A set of symbols that will cause the machine to carry out a single operation. Part of an instruction is the operation code; the rest contains such information as the address of the data to be used.

interpretive programming A method of writing programs in a pseudo-code. The program is stored in the memory along with an "interpreter" and each instruction is converted to a series of machine-language commands just before it is executed. (Sec. 2-8)

key punch A device for punching information on cards.

link That part of a subroutine that connects it with the rest of a program. (Sec. 4-4)

loading The process of feeding information into the memory via the input unit.

loop A sequence of operations repeated over and over until some criterion is satisfied. (Chap. 3)

loop box A device for modifying instructions. A number stored in the loop box can be added to an instruction just before the instruction is executed, modifying it without changing the instruction as stored in the memory. Synonyms: B-line, indexing accumulator. (Chap. 3)

machine language The basic codes to which the computer responds. (Chaps. 7 and 8)

memory That part of the machine in which information can be retained until needed. Synonym: storage.

object program A term in automatic programming for the machine-language program produced by the machine by translating a source program written by the programmer in a language similar to algebraic notation.

octal A number system with the base 8.

optimize To space instructions or data around the drum memory so that the information is in a position to be accessible when needed. (Sec. 7-5)

output Either that part of the machine which produces a record of results, or the record itself.

overflow The condition that exists when a number exceeds the normal range of the accumulator during a calculation. (Sec. 7-15)

program (*noun*) The sequence of steps the computer is to perform when carrying out a computation.

program (*verb*) To set up the program.

punched card A card on which information is recorded in the form of holes in specific locations.

register Any part of the machine in which information can be retained.

reset See *clear*.

sequential programming Placing the instructions of a program in the memory so that instructions that are performed one after the other have memory addresses in the same order.

sexadecimal system See *hexadecimal system*.

single-address machine A computer in which an instruction normally contains only one address; therefore a single instruction can only be used to bring one factor to the accumulator at a time and a separate instruction is needed to store a result. See also *three-address machine*.

source program An automatic-programming term for the program written by the programmer in a language similar to algebra, to be translated to an object program in the machine language by the computer. (Chap. 6)

statement A generalized instruction used in automatic programming.

storage See *memory*.

subroutine A set of instructions which can be easily incorporated into a larger program and which will perform some special computation, such as the evaluation of a trigonometric function, or the solution of a system of linear equations. (Chap. 4)

symbolic programming A method of programming in which symbols are used instead of explicit numerical codes and addresses; the machine is then used to translate these into the machine language.

tag (*noun*) A symbol attached to an instruction to indicate that it is to be treated in a special way.

three-address machine A computer in which an instruction normally contains three addresses. An addition instruction will contain the addresses of addend and augend and the address in which the sum is to be stored.

trace (*verb*) To produce a record of each instruction as it is performed by the machine, including such information as the source of the instruction and the factors used in the computation.

transfer card A card used to terminate the loading process and to begin the execution of the program. (Sec. 2-15)

transfer instruction See *branch instruction.*

translator A program used to translate from a code created for convenience in programming to the basic language of the machine.

update To modify an instruction so that the addresses are increased by an assigned amount every time the instruction is performed.

word A group of digits normally treated as a unit by the machine.

X-punch A punch in the first row above the zero row on an IBM card; used in scientific computations to denote a minus sign. Synonym: 11-punch.

Y-punch A punch in the second row above the zero row on an IBM card; used in scientific computations to denote a plus sign. Synonym: 12-punch.

Index

Accumulator, 159–161, 219
  (*See also* IBM Type 650)
Address, 21–22, 219
  (*See also* Bell Telephone Laboratories interpretive code; IBM Type 650; SOAP)
Address modification (*see* Incrementing techniques; Loops)
Address search, 113, 215
Address Selection Switches, 115
Alphabet, 33–36
Analog computer, 3, 219
Arc sine *x*, 73–76
Arc tanh *x*, 80, 100
Arithmetic unit, 8, 219
  (*See also* IBM Type 650)
Arrays, 144–149
  (*See also* SOAP, addresses, regional)
Assembly, 219
  (*See also* FORTRAN-FORTRANSIT; SOAP)
Automatic programming, 219
  (*See also* FORTRAN-FORTRANSIT)

B line, 55–63, 221
Bands, 158, 189–190, 195–196
Base, 14–17
Bell Telephone Laboratories interpretive code, 24, 120, 137, 157, 161–162
Bell Telephone Laboratories interpretive code, address 000, 29
  addresses, 24–25
  branching, 63–80
  card format, 36–42
  fixed-point arithmetic, 56–59, 69–74
  floating-point arithmetic, 26–33
  incrementing, 52–54, 68–73
  initialization, 69–73
  input, 33–42
  loading, 39–42
  loops, 51–80, 93–97, 115–116
  MOVE instruction, 62
  no operation, 105
  output, 43
  PUNCH instruction, 43
  punch mode, 107
  READ instruction, 42
  special functions, 47
  stop, 79
    conditional, 110, 115
  subroutines, 84–99
  trace mode, 107
  transfer, conditional, 63, 69, 111
    unconditional, 43, 87–89, 106
  transfer card, 40, 44, 105, 195, 222
Binary computers, shifting in, 182
Binary numbers, 13–17, 220
Binary-octal numbers, 17
Binary point, 15

Bit, 15, 219
Block programming, 84, 102–103, 204–205, 211
Boxes (*see* Block programming; Flow diagrams)
Branching operations, 63–80, 149–152, 196–199, 220
Buffer, 220

Clearing operations, 77, 165–167, 220
Code, 5, 220
(*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT; IBM Type 650; SOAP)
Code check (debugging), 101–118, 138, 150, 152, 213–215, 220
Comparison test, 112
Computer components, 8
(*See also* IBM Type 650)
Console (*see* IBM Type 650)
Control unit, 8, 220

Debugging techniques, 101–118, 138, 150, 152, 213–215, 220
Decimal point, 14, 18–21, 164
(*See also* Fixed-point arithmetic; Floating-point arithmetic)
Differential equations, 153
Digital computer, 3, 220
(*See also* IBM Type 650)
Distributor, 159, 165, 197–198, 220
Double-precision arithmetic, 17, 216

Electrostatic storage, 8, 163
Execution, 69, 200
(*See also* Loops)
Exponent, 19–21, 63–66, 125, 133, 137
Exponential function, 47, 135
Exponential integral function, 80, 82–83

Factorial function, 79, 100, 141
Fixed-point arithmetic, 18, 220
(*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT; IBM Type 650; Scaling techniques; SOAP)
Floating-point arithmetic, 18, 182, 216, 221
(*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT)
Flow diagrams, 44, 81–86, 102, 221
FORTRAN-FORTRANSIT, 122, 157, 161
arithmetic symbols, 129
arrays, 144–149
assembly, 120–123, 128, 137
branching, 149–152
correspondence table, 137
fixed-point arithmetic, 131–134
fixed-point constants, 124–125
fixed-point variables, 123–124
floating-point arithmetic, 131–134
floating-point constants, 124–125
floating-point variables, 123–124
hierarchy of operations, 130
incrementing, 150
initialization, 150
input, 136
loading, 136–137
loops, 140–143, 147–150
mixed equations, 134
mixed expressions, 133
no operations, 142
object program, 128, 222
output, 137–138
source program, 128
special functions, 134–136
statements, CONTINUE, 142
DIMENSION, 147
DO, 140, 152
IF, 149–150
GO TO *n*, 138
computed, 151–152
PAUSE, 139, 152
PUNCH, 137
conditional, 138–139, 148, 152
READ, 136
STOP, conditional, 139
subroutines, 151–152
subscripted variables, 145–149
subscripts, 133, 145
symbolic variables, 123–124
transfer, conditional, 149–152
unconditional, 138

Hyperbolic sine and cosine, 48, 91–93, 117, 141–142

IBM Type 650, 21
accumulator, 159–161, 164–187, 173–177
addresses, 158–159, 161
arithmetic unit, 158–159
branching, 196–199
card format, 188–189
console, 111, 114–115, 138, 159, 177, 192–194, 209, 213, 215–216, 220
control panel, 188, 220
data address, 161
distributor, 159, 165, 197–198, 220
8000 addresses, 159, 173–175
fixed-point arithmetic, addition, 159–160, 165–168
division, 179–180
multiplication, 177–178
subtraction, 165–168
incrementing, 201–203
initialization, 202–203
input, 188–195, 198
instruction address, 161
instruction word, 161
load card, 189–195, 203
loading, 188–195, 209, 213, 221
loops, 200–203
machine language, 00 (NOP), 213
01 (HLT), 213, 215
10 (AUP), 167
11 (SUP), 167
14 (DIV), 179
15 (ALO), 167
16 (SLO), 167
17 (AML), 167
18 (SML), 167
19 (MPY), 177
20 (STL), 168
21 (STU), 168
22 (SDA), 202, 216
23 (SIA), 202
24 (STD), 163, 191, 196, 200
30 (SRT), 181
31 (SRD), 181
35 (SLT), 180
36 (SCT), 182
44 (NZU), 197
45 (NZE), 197
46 (BMI), 197
47 (BOV), 177, 197
60 (RAU), 167
61 (RSU), 167
64 (DVR), 179
65 (RAL), 165–167
66 (RSL), 167
67 (RAM), 167
68 (RSM), 167
69 (LDD), 191, 196, 200
70 (RCD), 190
71 (PCH), 196
90 to 99 (BD0 to BD9), 197–198
no operation, 213–214
optimization, 162, 171, 205, 210–211, 222
output, 195–196
program register, 159
punch bands, 195–196
read bands, 189–190
registers, 158–159, 222
rounding, 181–182
shifting, 180–183
standard board, 188
stop, conditional, 213–215
storage capacity, 158
storage codes, 168
subroutines, 206, 209–214
transfer, conditional, 196–200
word size, 158
Illiac, 16, 21
Incrementing techniques, 52–54, 68–73, 150, 201–203
(*See also* Loops)
Infinite series, 10, 73–76, 80, 141–142
Initialization techniques, 69–73, 150, 202–203, 221
Input techniques, 8, 33–41, 221
(*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT; IBM Type 650; SOAP)
Instructions, 22, 221
inserting, 106–107
(*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT; IBM Type 650; SOAP)

Integration techniques, 50, 132
Interest table, 80
Interpolation problems, 10, 49–50
Interpretive programs, 23–24, 221
  (*See also* Bell Telephone Laboratories interpretive code)
IT, 122

Key punch, 36–39, 126, 221

Library, 5, 98–99, 206, 210–211, 214
Link, 86–89, 151–152, 212–213
Load card, 189–195, 203
Loading techniques, 39–42, 136–137, 188–195, 209, 213, 221
Loop box, 55–63, 221
Loops, 51–80, 93–97, 115–116, 140–143, 147–150, 200–203, 221
  arrays in, 147–149
  counting, 52, 59–60
  without counting, 54–55
  nested, 76–79, 142–143
  summation, 60–61, 93–97, 147–150
  tracing, 115–116

Machine language, 22, 157, 221
  (*See also* IBM Type 650)
Magnetic core, 8, 163
Magnetic drum, 8, 158, 189–190, 195–196
Magnetic tape, 8
Mantissa, 19
Matrices, 145, 153, 217
Memory, 8, 21, 25–26, 163, 222
  (*See also* Magnetic drum)
Memory dump, 103–104, 215, 220

Newton-Raphson method, 64–66
No operation instructions, 105, 142, 213–214
Normalization techniques (*see* Floating-point arithmetic)
Number systems, 13–17
Numerical analysis, 6–8

Object program, 128, 222
Octal numbers, 13–17, 222

Optimization techniques, 162, 171, 205, 210–211, 222
Output techniques, 8, 43, 137–138, 195–196, 222
Overflow, 176–177, 180, 186, 197, 222
Overflow circuit, 177, 181, 183, 197
Overflow switch, 177
Overwriting error, 113

Packing information, 17, 189, 216
Planck function, 93, 140
Polynomial, 31, 55–59, 69–73, 103, 153
Program, 4, 222
Program register, 159
Programmed Stop Switch, 111, 114, 213
Pseudo-code, 23
  (*See also* Bell Telephone Laboratories interpretive code; FORTRAN-FORTRANSIT; SOAP)
Pseudo-operation (*see* SOAP)
Punched cards, 8, 33–39, 126–127, 168–169, 222
Punched tape, 8

Read band, 189–190
Rounding techniques, 181–182

Scalar product, 60–61, 94–97, 147–150
Scaling techniques, 164, 183–187
Sentence, 167
Sequential programming, 25, 43, 162, 222
Series expansions, 10, 73–76, 80, 141–142
Setting of link, 86–89, 151–152, 212–213
Seven-per-card loader, 195, 215
Sexadecimal numbers, 13–17, 222
Shifting techniques, 180–183
Single-address instruction, 22, 222
Single-word load card, 193, 215
Snapshots, 104–106
SOAP, 163–164
  addresses, absolute, 170, 200, 204–205, 207, 210
    blank, 172–173, 198–199, 210, 215
    regional, 170–172, 190–196, 201–203
    symbolic, 170–172, 207–209
  assembly, 168–173, 195, 203–212, 214–215

SOAP, availability table, 205–206, 214–215
block availability, 204
block reservation, 204
card format, 168–169, 188–189
codes, ALO (15), 167
AML (17), 167
AUP (10), 167
BD0 to BD9 (90 to 99), 197–198
BMI (46), 197
BOV (47), 177, 197
DIV (14), 179
DVR (64), 179
HLT (01), 213, 215
LDD (69), 191, 196, 200
MPY (19), 177
NOP (00), 213
NZE (45), 197
NZU (44), 197
PCH (71), 196
RAL (65), 165–167
RAU (60), 167
RAM (67), 167
RCD (70), 190
RSL (66), 167
RSM (68), 167
RSU (61), 167
SCT (36), 182
SDA (22), 202, 216
SIA (23), 202
SLO (16), 167
SLT (35), 180
SML (18), 167
SRD (31), 181
SRT (30), 181
STD (24), 163, 191, 196, 200
STL (20), 168
STU (21), 168
SUP (11), 167
constants, 200
error correction, 215
fixed-point arithmetic, addition, 159–160, 165–168
division, 179–180
multiplication, 177–178
subtraction, 165–168
heading, 206–207, 209
incrementing, 201–203
initialization, 202–203
SOAP, input, 188–195, 198
loading, 188–195, 209, 213, 221
loops, 200–203
no operation, 213–214
optimization, 162, 171, 205, 210–211, 222
output, 195–196
pseudo-operations, 171–172, 203–212, 216
BLA, 204, 214
BLR, 204, 214
EQU, 207, 214
HED, 206, 214
PAT, 205, 214
RBR, 211
REG, 171–172, 190, 196, 201–203, 214
REL, 210, 214
REQ, 211
SYN, 208–209, 214
rounding, 181–182
shifting, 180–183
stop, conditional, 213–215
subroutines, 206, 209–213
entry and exit, 212–213
relocatable, 210–212
transfer, conditional, 196–200
type column, 168, 206, 210
Source program, 128, 222
Special symbols, 126
Spherical coordinates, 100
Statement, 125, 222
(*See also* FORTRAN-FORTRANSIT)
Statistics, 10
Stirling's approximation, 100
Stop, 79, 139
conditional, 110, 115, 139, 213–215
Storage, 8, 21, 25–26, 163, 222
temporary, 90
(*See also* Magnetic drum)
Storage-entry switches, 138, 159, 193–194, 209, 216
Stored-program machine, 8, 22
Subroutines, 84–99, 151–152, 206, 209–214, 222
code words in, 93–97
constants in, 90
entry and exit, 86–89, 151–152, 212–213
generalized, 91–93
testing, 116–118

Summation techniques (*see* Loops)
Symbolic variables (*see* FORTRAN-FORTRANSIT; SOAP)

Tables, 10, 76–80
Tabulator, 39, 206, 215
Tag, 56–59, 168–169, 222
Test instructions (*see* Branching operations; Transfer instructions)
Testing programs, 101–118, 138, 150, 152, 213–215, 220
Three-address instruction, 22, 222
Trace instructions, 107–110, 115–116, 215, 222
  selective, 109–110
Transfer instructions, conditional, 63–69, 149–152, 196–200
  unconditional, 43, 87–89, 106, 138

Updating techniques (*see* Incrementing techniques)

Verifier, 39

Word size, 17
Word time, 162
Write-ups, 5, 98–99, 206, 210–211, 214

X-punch, 33, 137, 189, 222

Y-punch, 33, 137, 189, 222

Zone, 33–36