W. Y. STEVENS

A Simplified System

FOR THE USE OF AN

Automatic Calculator

Watson Scientific Computing Laboratory
612 West 116th Street
New York, N. Y.

A Simplified System

FOR THE USE OF AN

Automatic Calculator

By DAVID MACE and JOYCE ALSOP

Watson Scientific Computing Laboratory
Columbia University
International Business Machines Corporation
1957

PREFACE

The System described in this pamphlet was developed at the Watson Scientific Computing Laboratory to enable classes and research groups in Columbia University to use the equipment of the Laboratory for solving problems in mathematics, science, and technology.

Large numbers of students and research workers are eager to obtain first-hand experience in applying machines to their problems, but they feel that they can not afford the time for a professional course in machine methods. The instruction presented here meets the needs of these people with a minimum expenditure of time; it consists of the presentation of the System and the solution of a problem chosen from the field of specialization of the group. Three one-hour sessions of formal discussion, each preceded by one or two hours of home preparation, are considered sufficient for the student to learn the System and to be ready to undertake a problem of moderate complexity. His familiarity with the problem is a great asset to the student since a major part of the total effort in solving a problem by machines involves his thinking through the method of solution in detail and stating exactly the necessary steps for the solution.

At the conclusion of the instruction the student should have precise knowledge of all the factors that were involved in obtaining a machine solution. His concrete experience with one machine enables him to solve simple problems on it and to cooperate effectively with professional computers in the solution of more complicated problems on any machine.

Important contributions to the project were made by J. Jeenel, H. Smith, and E. Hankam.

CONTENTS

I. Introduction

1. THE SYSTEM

One of the most exciting achievements of our generation is the development of the electronic automatic digital calculator. Although any schoolboy can perform any operation done by the calculator, the speed and economy with which the calculator does them are so great that automatic calculation is revolutionizing large areas of science, engineering, business, industry, and defense. A single giant calculator can do more arithmetic than the entire population of the United States could do with pencil and paper.*

The calculator described here (the IBM 650) is one of medium speed, i.e. it will perform in two or three minutes calculations that would require a week on a desk calculator. A very important feature of the calculator is the fact that simple written instructions control its operation. This pamphlet in conjunction with several lectures aims to train the novice to write these instructions for the machine solution of moderately complex problems.

In order that the machine be used effectively on the solution of such problems by a large number of people, simple uniform procedures should be followed. This uniformity will help the novice avoid many of the common time-consuming errors made by beginners and will enable the professional computing staff to render effective assistance when needed. It will also permit standardized sets of instructions previously prepared for other purposes to be incorporated in a new problem. This pamphlet therefore describes not only the machine but a "System" for its use. The beginner is requested to follow this System until he has successfully completed several problems. During this initial period he should concentrate on understanding this System rather than dis-

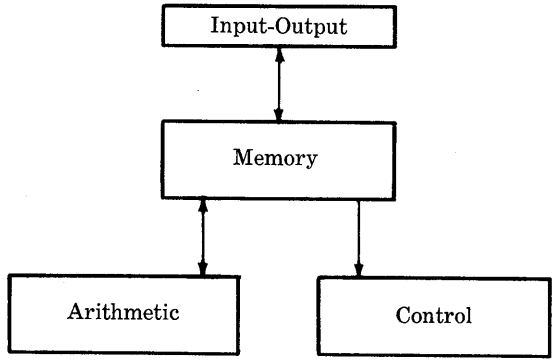
*Eckert, W. J. and Jones, R., *Faster, Faster,* McGraw-Hill, New York, 1955, Introduction.

cussing other systems (of which there are many).

The System described here is direct and simple. It will handle efficiently problems of moderate size and complexity, and it will give to the student a good introduction to more advanced methods suitable for the solution of larger and more complicated problems. After he has gained experience on small problems, he can undertake somewhat more complicated ones with the guidance of the professional computing staff. Those who wish to undertake the solution of large complicated problems, however, would be well advised to seek more formal instruction.* It should be remembered that time on this calculator is worth more than a dollar a minute and problems requiring hours of machine time should be prepared in a manner that will avoid waste of machine time due to inadequate numerical analysis, inefficient planning of machine operations, and erroneous machine instructions. Of perhaps greater importance than wasted machine time is the loss of time of the person writing instructions in an unsystematic manner. The urge to be independent is commendable, but it should be remembered that thousands of people including some of the world's greatest mathematicians have worked on these procedures for the past ten years and the beginner should profit by all of their early mistakes.

2. THE MACHINE

The calculator consists of an input-output device, a memory, an arithmetical unit, and a control unit. Numbers are



*Columbia University Bulletin of Information, Announcement of the Watson Scientific Computing Laboratory.

read from punched cards by the input-output device and stored in the memory. Numbers are taken from the memory to the arithmetical unit where they are added, subtracted, multiplied, or divided, and the results are returned to the memory. Certain other logical operations to be described later are also performed in the arithmetical unit. Numbers stored in the memory may be recorded in punched cards for later use by the machine or for producing a printed record by means of a separate printing machine.

3. THE MEMORY

The 650 memory will store as many as 2000 "words", where a "word" consists of ten digits and algebraic sign. The 2000 storage locations, each capable of holding a word, are numbered consecutively from 0000 to 1999; these location numbers are called addresses. In order to put a "word" into one of these locations or to remove a word from one, it is necessary to specify the address of the memory location or "cell". For this System, locations 0000-0499 and locations 1700-1999 have been reserved for special purposes so that only locations 0500-1699 can be used for problems.

A word in a memory cell is retained until a new word is placed in the cell; before the new word is accepted by the cell the old word is erased automatically.

4. THE ARITHMETICAL UNIT

The arithmetical unit consists essentially of a 10-digit accumulator with sign, called the upper accumulator; there is also a 10-position extension, called the lower accumulator, that is used in multiplication and division.

Unlike the storage cells, the contents of the accumulator are not erased unless the accumulator is deliberately reset to zero. To add a number from the memory into the accumulator, we instruct the machine to "reset add upper" (that is, first reset the accumulator to zero and then add the word into the upper accumulator) and we give the address of the word to be added. For example, to add the word from cell 0562 into the accumulator, we write:

reset add upper (RAU) 0562

Similarly, to subtract the same number, we write:

reset subtract upper (RSU) 0562

We can also instruct the machine to add or subtract without first resetting; in this way we accumulate a sum. For

example, if we have three numbers A, B, C, stored in locations 0901, 0902, 0903, respectively, we can form $(S = A + B - C)$ with the following sequence of commands:

reset add upper	(RAU)	0901
add upper	(AU)	0902
subtract upper	(SU)	0903

These commands and others will be described in detail in Chapter II.

5. THE CONTROL UNIT AND INSTRUCTIONS

All commands are given to the machine as numerically coded instructions, which are in the form of the standard word, 10 digits and sign. Since the instructions are entirely numerical, they can be read from cards and stored in memory as data. Each instruction has its own address. Since instructions are indistinguishable from data in the memory, the 650 is called a "stored-program calculator".

If an instruction is entered into the arithmetical unit, it will be treated as a data word. Only when it is entered into the control unit will it be treated as an instruction; for it is in the control unit that the instruction is interpreted and executed. The digits of an instruction word have the following meaning:

xx	xxxx	xxxx
Operation code	Data address	Instruction address

The operation code is the numerical equivalent of the operation we wish the machine to perform; for example, the operation code for "reset add upper" is 60. The data address locates in memory the data word on which the operation is to be performed.

With some operation codes the data address has different meanings that will be explained in the discussion of specific operation codes.

After an instruction has been executed, the control unit goes to memory for the next instruction. The instruction address of the instruction just executed tells the control unit where in memory the next instruction is stored. For example,

60 0632 0502

is interpreted as, "Reset add upper the word in 0632 and get the next instruction from 0502". The coder generally writes on his worksheet the location of the instruction and the

alphabetic abbreviation of the operation to be performed, in addition to the instruction:

0501 | RAU 60 0632 0502

The alphabetic abbreviation is just for his convenience, but the location of the instruction (0501) is a necessity since he must know where each instruction is stored. However, only the 10-digit instruction word, 60 0632 0502, appears in the machine.

We can see now that the machine needs only the location of the first instruction; it will then execute automatically a whole series of instructions in the proper order. Such a series of instructions is called a program.

6. THE SOLUTION OF A PROBLEM

The solution of a problem on an automatic calculator involves the following steps:

Stating the problem
Establishing the numerical procedures
Planning the machine solution
Programming
Writing detailed machine instructions
Testing the instructions
Running the problem on the machine.

The problem is generally stated in the form of mathematical equations; in the second step these equations are replaced by a set of numerical procedures including the necessary problem data.

The numerical procedures are then analyzed to determine how they may be arranged to the best advantage for machine solution. This analysis and the general outline of the machine solution is called "problem planning". We refer to the development from this general plan to a detailed plan for all of the operations and the order in which they are to be performed as "programming".

When the program is complete, we translate it into coded form by expressing each operation in terms of specific machine codes; we refer to this process as "coding". Generally, the clerical part of large-scale coding is done automatically by machines.

The order of the steps given above is that in which a problem is solved; it is not the best order for exposition. In this pamphlet we discuss coding first, then programming, scaling and, finally, testing; machine operation is not discussed.

II. Coding

1. ARITHMETICAL OPERATIONS

There are six arithmetical and two associated operations, the first two of which have already been discussed:

RAU	60	reset add upper
RSU	61	reset subtract upper

The entire (upper and lower) accumulator is first reset to zero, and then the contents of the cell specified by the data address are added or subtracted into the upper accumulator.

To store in memory a word that is in the upper accumulator, we have the operation code 21:

STU	21	store upper

The word is stored in the cell indicated by the data address of the instruction, and the accumulator remains unchanged.

A store upper (STU) instruction can be combined with a reset add (RAU) instruction to perform a simple word transfer (moving a word from one memory location to another). To transfer a word from 0800 to 0900, we write:

0600	RAU	60	0800	0601
0601	STU	21	0900	0602

Note that we have arbitrarily put these instructions in locations 0600 and 0601.

The next two operation codes enable us to add and subtract into the upper accumulator without first resetting the accumulator to zero:

AU	10	add upper
SU	11	subtract upper

For example, if we have four numbers A, B, C, D, stored in, say, cells 0801 to 0804, respectively, and we wish to form

$$E = A + B - C + D$$

and to store E in cell 0805, we write the following instructions:

0500	RAU	60	0801	0501
0501	AU	10	0802	0502
0502	SU	11	0803	0503
0503	AU	10	0804	0504
0504	STU	21	0805	0505

Again the instructions are arbitrarily stored in locations 0500 to 0504.

Two 10-digit numbers can be multiplied to give the 20-digit product (with sign):

MPY 19 multiply

Two steps are involved: the first step clears the upper and lower accumulator and places the multiplier in the upper accumulator by the RAU operation just described; the next step, an instruction containing the operation code 19, initiates the multiplication. The data address locates the multiplicand. For example, if we wish to multiply the number in cell 0850 by the one in cell 0851, we write:

0500	RAU	60	0850	0501
0501	MPY	19	0851	0502

The product appears in the accumulator with the 10 digits in the highest-order positions in the upper accumulator and the 10 lowest-order digits in the lower accumulator:

Number in 0850	1100000012	+
Number in 0851	1300000014	−
Product in accumulator	0143000003 | 1000000168	−

If the lower accumulator is not clear when a multiply operation is called for, the result will be completely incorrect. For this reason, in successive multiplications the intermediate products must be stored between successive multiplication steps. For example, the following instructions are necessary to compute x^3 when x is stored in 0600:

0500	RAU	60	0600	0501
0501	MPY	19	0600	0502
0502	STU	21	0601	0503
0503	RAU	60	0601	0504
0504	MPY	19	0600	0505

Note that location 0601 is used for temporary storage of the intermediate product, x^2. The final product is now in the accumulator and can be stored in any desired location.

Two 10-digit numbers can be divided to give a 10-digit quotient:

```
DIVR   64   divide
```

The dividend is placed in the upper accumulator with a RAU instruction, and the data address of the divide instruction locates the divisor. The 10-digit quotient appears in the *lower* accumulator, and the upper accumulator resets to zero.

To store in the memory a word that is in the lower accumulator, we have the operation code 20:

```
STL    20   store lower
```

Since the quotient appears in the lower accumulator, we need an operation analogous to "store upper" in order to put the quotient in memory; operation code 20 instructs the machine to store the contents of the lower accumulator.

There is one restriction on the size of the numbers in division. The absolute value of the dividend must be less than the absolute value of the divisor. If this restriction is not observed, the machine will want to develop more than 10 quotient digits and stop. The subject of scaling is discussed further in Chapter IV.

Note: All arithmetical operations are algebraic.

2. SHIFT OPERATIONS

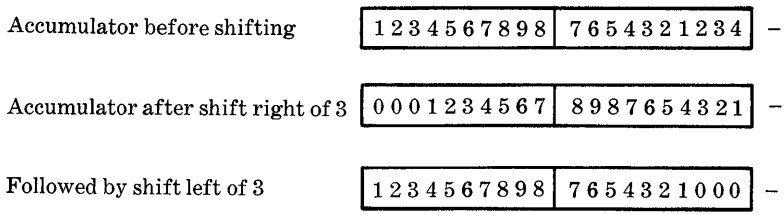
The contents of the *entire* accumulator may be shifted to the right or left any number of positions from 0 to 9:

```
SHRT   30   shift right
SHLT   35   shift left
```

The number of positions to be shifted is indicated by the data address of the shift instruction. Shift right and shift left 3 positions are written:

```
SHRT   30   0003   xxxx   (shift right)
SHLT   35   0003   xxxx   (shift left)
```

The digits shifted out at either end of the accumulator are lost, as illustrated below:



3. BRANCH OPERATIONS

A very important property of a calculator is its ability to choose one of two alternative programs depending on the contents of the accumulator. We have two of these conditional operations; in the first, the machine branches on non-zero in the upper accumulator:

BRNZU 44 branch on non-zero in upper

In this operation the choice depends on a zero or non-zero condition in the upper accumulator. If the upper accumulator is zero, the next instruction is taken from the instruction address as usual. If the upper accumulator is non-zero, the machine branches and takes the next instruction from the data address:

0500	BRNZU	44	0501 go here if acc. $\neq 0$	0502 go here if acc. $= 0$

In the second branch operation, the choice of program depends on the sign of the accumulator:

BR– 46 branch on minus

If the sign of the accumulator is positive, the next instruction is taken from the instruction address; if the sign is negative, the machine branches and takes the next instruction from the data address. The accumulator is unchanged by a branch operation.

Operation code 01 is provided to stop the machine:

STOP 01 stop

For various purposes it is desirable to have the machine stop

when it reaches a designated point in the calculations or under other specified conditions.

4. SPECIAL OPERATIONS

There are standard programs for performing sequences of operations that are used frequently, such as data transfers, and computation of functions, such as sin, cos, exponential, and square root. To eliminate the necessity of writing these programs over and over again, we have written them permanently in such a way that they can be incorporated easily in a larger program. We sometimes refer to such standard programs as "library" programs, since you can take them off of the shelf and use them when you need them. A library program is also known as one type of subroutine; other types of subroutines will be discussed later.

Since the same subroutine may be used in several places in the program, it is necessary in each case to tell the machine where to take up the main program again when the subroutine is completed. A subroutine is called for by the operation code:

SPOP 69 special operation

The data address of code 69 is the address of the instruction that follows the subroutine; the instruction address is the code number of the desired special operation. For example,

SPOP 69 1103 0061

means, "Perform subroutine 0061 and go to location 1103 for the next instruction".

If the subroutine is the computation of a function, we must specify the argument of the function to be computed by entering the argument into the accumulator just before giving the SPOP command. For convenience of notation we use α as the argument and $L(\alpha)$ as the location of α in the memory.

The instructions to compute sine α are, therefore:

0701	RAU	60	$L(\alpha)$	0702
0702	SPOP	69	0703	0071

where 0703 is the location of the next instruction. At the completion of a subroutine, the function being computed always appears in the upper accumulator, and the lower accumulator resets to zero so that a multiplication can be performed directly.

The following list of special functions is a preliminary one; other functions can be added:

0070	$\sqrt{\alpha}$	0074	$\log_e \alpha$
0071	$\sin \alpha$	0075	$\arctan \alpha$
0072	$\cos \alpha$	0076	$\arcsin \alpha$
0073	e^{α}		

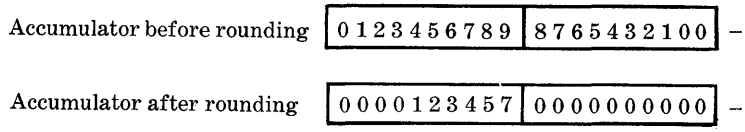
These functions are described more fully in Appendix I.

The special operations that are used most frequently concern the rounding of numbers in the accumulator and the transfer of blocks of numbers from cards to the memory, from one part of the memory to another, and from the memory into cards.

Rounding is accomplished by a shift of the number in the accumulator and the addition of 5 into the highest-order position of the lower accumulator. This combined operation is controlled by the special operation codes 0050 to 0059, where the units digit of the code number designates the number of positions to be shifted. For example,

69 0801 0053

means, "Shift right 3 places and round; take next instruction from 0801".



The lower accumulator contains zeros at the end of the rounding special operation.

Subroutines for the block transfer of data are:

Memory to memory
0060 from one set of memory locations to another
Memory to cards
0061 from a set of memory locations to cards
Cards to memory
0062 from cards to a set of memory locations

The subroutines for block transfers of data are initiated in much the same way as the subroutines for evaluating functions. Again we must tell the machine where to take up the main program after the subroutine is completed, and we must also specify how many words to transfer, the memory locations involved, etc. This information concerning the

blocks of data to be transferred is contained in a code word that we construct and enter into the upper accumulator just before the subroutine is called for. This code word is called α, just as the argument of the function is called α.

In the memory to memory subroutine 0060, α is constructed as follows:

$$\alpha = \underset{N}{\text{xx}} \quad \underset{a}{\text{xxxx}} \quad \underset{b}{\text{xxxx}}$$

where

N = the total number of words to be transferred
a = the first address of original set of locations
b = the first address of new set of locations

For example, "α = 20 0800 0950" indicates that we wish to transfer 20 words from locations 0800-0819 to locations 0950-0969, respectively.

Subroutine 0061 will punch a block of numbers into a group of cards. α is constructed as follows:

$$\alpha = \underset{N}{\text{xx}} \quad \underset{a}{\text{xxxx}} \quad \underset{n}{\text{x}} \; \underset{b}{\text{xxx}}$$

where

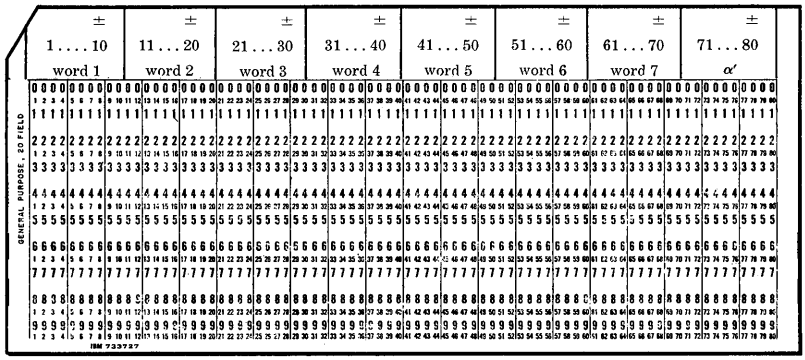
N = the total number of words to be punched
a = the location of the first word in the block
n = the number of words to be punched per card
b = the block number (for identification)

For example,

$$\alpha = 30 \quad 1501 \quad 5 \quad 182$$

indicates that we wish to punch the 30 words in locations 1501-1530 into (6) 5-word cards, each card containing the block number 182. Each card will contain, therefore, the 5 words and a code word that differs from α in that the location of the first word in the block is replaced by consecutive numbers for identification of each card within a group. Let us call this punched-out code word α'.

A card has 80 columns numbered from left to right. It will hold 8 words of 10 digits each, since the sign of each word is contained in the same column as the units digit. We shall always use the 8th word for identification, leaving a maximum of 7 words per card. In subroutine 0061 just described, α' is the identification word which appears as "word 8".



Subroutine 0062 will read a block of data from a group of cards into consecutive memory locations. α is constructed as follows:

$$\alpha = \underset{N}{\text{xx}} \quad \underset{a}{\text{xxxx}} \quad \underset{n}{\text{x}} \; \underset{b}{\text{xxx}}$$

where

N = the number of words in the block
a = the location where the first word is to be stored
n = the number of words to be read from each card
b = the block number

The cards that are read in by subroutine 0062 must also contain an α' for identification.

Note: When subroutines 0061 and 0062 are used, it is important that the total number of words in the block be an integral multiple of the number of words per card; if this condition is not fulfilled, extra words can be read in or punched out.

We can now put individual instructions together to form a complete program. For an example of a program, let us write the following instructions that will: read four numbers A, B, C, D, from a card into locations 1200, 1201, 1202, 1203, respectively; compute $\frac{A}{B} + C \sin D = E$; store E in 1204; and punch out A, B, C, D, E. Let us store our instructions in consecutive locations starting at 0500. Since it will be necessary to use certain constants, which we call program constants, for the SPOP operations, we shall store them in locations starting at 0600. For a temporary storage location, let us use 0700.

0500	RAU	60	0600	0501	Read in
0501	SPOP	69	0502	0062	A, B, C, D
0502	RAU	60	1200	0503	Compute A/B and
0503	DIVR	64	1201	0504	store temporarily in 0700
0504	STL	20	0700	0505	
0505	RAU	60	1203	0506	Compute sine D
0506	SPOP	69	0507	0071	
0507	MPY	19	1202	0508	Compute $\left(\frac{A}{B} + C \sin D\right)$
0508	AU	10	0700	0509	
0509	STU	21	1204	0510	and store in 1204
0510	RAU	60	0601	0511	Punch A, B, C, D, E
0511	SPOP	69	0512	0061	and stop
0512	STOP	01	0000	0000	
0600		04	1200	4001	Constants
0601		05	1200	5002	

To read four numbers from a card and store them in locations 1200-1203 requires the code word, α = 04 1200 4001, which is stored as the first constant in location 0600 where it is available when called for by the instruction in 0500. In instruction 0507 you will notice that multiplication can be performed immediately because the lower accumulator resets to zero after the completion of subroutine 0071.

After the instructions and constants have been punched into cards, the operator enters the program deck into the machine by means of a loading procedure which is initiated by the manual controls on the console. The loading procedure places the program in memory so that the first instruction is stored in 0500, the next instruction in 0501, and so on. The starting address 0500 (the address of the first instruction to be performed) is set up on switches, and the depression of a button transfers this address into the control section of the machine. Then the calculator is started by the depression of the start button.

Exercise 1:

Write the instructions that will:

Read five numbers A, B, C, D, E from a card into locations 1401, 1402, . . . , 1405, respectively;

Compute $A\sqrt{B + C} + \frac{D}{E} = F$;

Store F in 1406;

Punch A, B, C, D, E, F into a card.

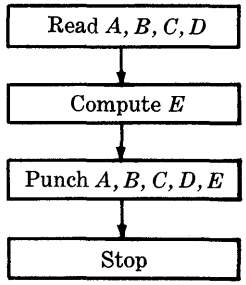
III. Programming

1. INTRODUCTION

In Chapter II we learned the language that our machine understands; now we must learn how to translate a problem from its mathematical symbols into this machine language. In translating directly from the detailed language of a specific problem to the detailed language of a specific machine, you must keep in mind all of the details of the problem and of the machine at the same time. Just as many problems have general similarities yet differ greatly in terminology, so modern computing machines conform to the same general pattern, but they differ in their detailed languages. It is advantageous, therefore, to have an intermediate language between the wide variety of problems and the numerous types of machines. With such a language available, we can begin by analyzing the problem, keeping in mind only general properties of the machine, and then forget the terminology of the problem and concentrate on the details of the machine. The intermediate language to be described in this chapter consists of simple diagrams that indicate the logical structure of the problem. The diagrams, which we call "flow charts", are graphic representations of the various phases of a problem.

2. THE PROGRAM LOOP

Let us consider again the problem for which the instruc-



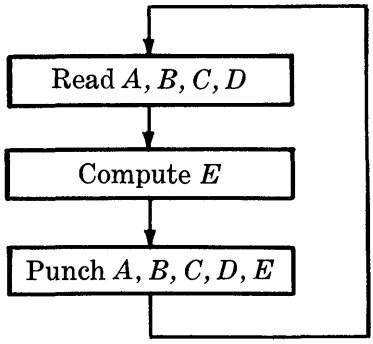
tions were written on page 14. We can represent the program coded there by the flow chart on the preceding page. For convenience in the following discussion, we shall copy here a portion of that program.

0500	RAU	60	0600	0501	Read in
0501	SPOP	69	0502	0062	*A, B, C, D*
. . .	. . .	. .	. . .	. . .	Compute *E* and
0509	STU	21	1204	0510	store in 1204
0510	RAU	60	0601	0511	Punch
0511	SPOP	69	0512	0061	*A, B, C, D, E*
0512	STOP	01	0000	0000	Stop

If we delete the last instruction and change the preceding one to read:

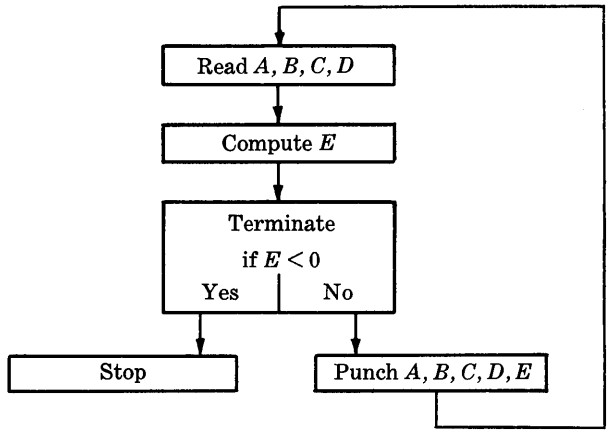
0511 | SPOP 69 0500 0061,

the machine will follow the punching of the card by reading another card. Once started on this routine, the machine will continue the process. In other words, the program will flow back on itself in a loop:



Once this program is initiated, it will continue to run until the cards run out.

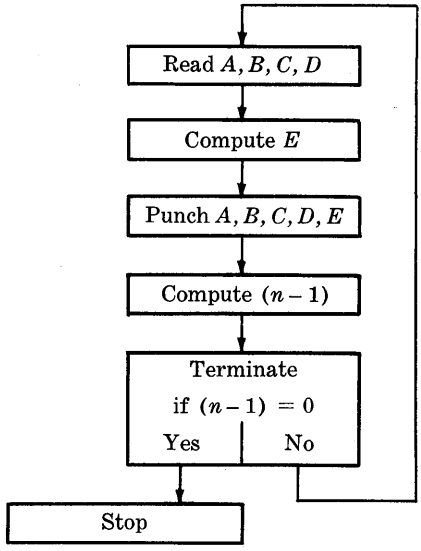
Program loops are very common and they can be terminated in many ways in addition to the one used above. Usually we use a branch operation to terminate a loop. For example, in the program used above we may instruct the machine to examine *E* each time it is computed and to terminate the process when the value of *E* becomes negative, as shown in the following flow chart:



In order to make this change in the program, we insert one instruction before the STU operation:

0509	BR–	46	0513	0510
0510	STU	21	1204	0511
0511	RAU	60	0601	0512
0512	SPOP	69	0500	0061
0513	STOP	01	0000	0000

As another example of terminating a loop, we can instruct the machine to go through the loop ten times and stop.



We assume that the instructions and the constants, 10 and 1, are in the machine and we instruct the machine to subtract 1 from 10 each time it goes around the loop. After ten times around, there will be a zero in the location that contained the digit 10. We then branch on zero to terminate the loop, as shown in the flow chart. The program on page 14 will perform this example if we change the instruction in location 0512 and add four more instructions:

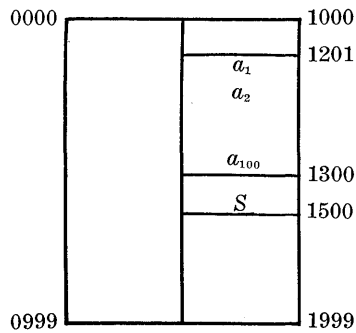
0511	SPOP	69	0512	0061	
0512	RAU	60	0602	0513	
0513	SU	11	0603	0514	
0514	STU	21	0602	0515	
0515	BRNZU	44	0500	0516	
0516	STOP	01	0000	0000	
0602		00	0010	0000	Constant
0603		00	0001	0000	Constant

3. MACHINE COMPUTATION OF INSTRUCTIONS

An important concept of a stored-program calculator is the machine's ability to compute its own instructions. This procedure as well as the principles of looping and terminating are illustrated in the following discussion of the accumulation of 100 numbers:

$$S = \Sigma_{i=1}^{100} a_i = a_1 + a_2 + a_3 + \ldots + a_{100}.$$

In the drawing of a memory layout for the problem, we show that the a_i's will occupy locations 1201-1300 and that the sum will be stored in location 1500:

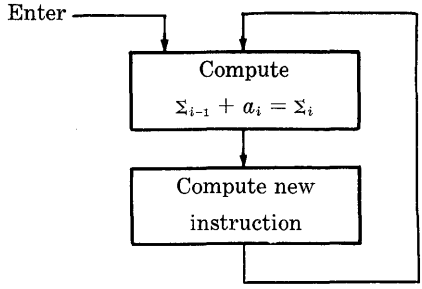


There follows the most straightforward program we can

write for this problem:

0500	RAU	60	1201	0501	Reset add a_1 into accumulator
0501	AU	10	1202	0502	Add a_2 into accumulator
0502	AU	10	1203	0503	Add a_3 into accumulator
. . .	. . .	. . .	. . .	. . .	. . . . .
0599	AU	10	1300	0600	Add a_{100} into accumulator
0600	STU	21	1500	Exit	Store Σa_i in 1500

Although this program is simple and straightforward, we find that it is costly in terms of its storage requirement of 101 locations (0500-0600). To alleviate this storage problem, we resort to the machine's ability to compute its own instructions. You will note that 99 of the instructions have the same operation code and that their data addresses form a sequence of integers from 1202 through 1300. Moreover, the first instruction at 0500 could be made to conform to this pattern if we could assume that the accumulator was reset to zero before the program started, i.e. the operation code (AU) 10 would be used instead of code (RAU) 60. Since the instructions conform to a simple pattern, it is not necessary to write them out and to store them in memory. As a part of the program, we can instruct the machine to compute detailed instructions as they are needed. The machine will execute the instruction that adds a term to the partial sum, and from this instruction it will compute the next instruction, which adds the next term to the partial sum, and so on. This process involves the program loop, and we represent the chronological execution of this loop by means of the following flow chart:



Since the accumulator is used to compute the new instruction, we must store the partial sum while the next instruction is being computed and restore it to the accumulator before adding in the next term. We shall use location 1500 for storing the partial sum. Each time around the pro-

gram loop we are adding the partial sum of $(i-1)$ terms to the i^{th} term, i.e. $\Sigma_i = \Sigma_{i-1} + a_i$; the instructions for this program loop read as follows:

0500	RAU	60	1500	0501	Compute
0501	AU	[10	1201	0502]	$\Sigma_{i-1} + a_i = \Sigma_i$
0502	STU	21	1500	0503	and store Σ_i in 1500
0503	RAU	60	0501	0504	Compute new instruction
0504	AU	10	0600	0505	
0505	STU	21	0501	0500	
0600		00	0001	0000	Program constant

We shall assume that the contents of location 1500, the summation cell, are zero initially, i.e. $\Sigma_0 = 0$, so that $\Sigma_1 = \Sigma_0 + a_1 = a_1$.

The instruction in 0500 enters the partial sum into the accumulator, and the instruction in 0501 adds a term a_i to this partial sum. Initially, the instruction in 0501 reads (10 1201 0502) so that a_1 will be added to the partial sum to form: $\Sigma_1 = \Sigma_0 + a_1$. Instruction 0502 stores Σ_1 back in 1500, the summation cell.

In order to add a_2 to the partial sum the next time through the loop, we change the instruction in 0501 by entering it into the accumulator and adding (00 0001 0000). In this manner the machine computes the new instruction:

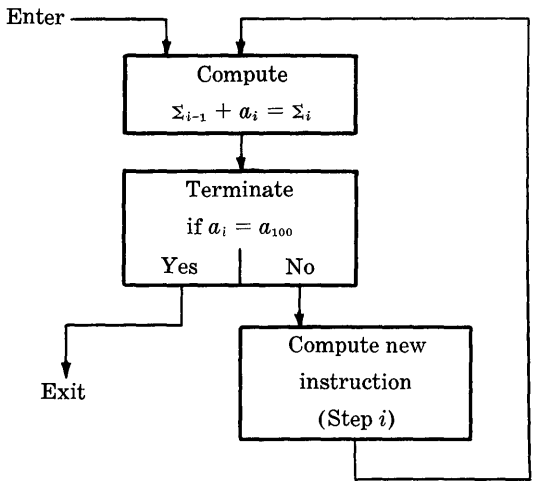
$$\begin{array}{rrrr} & 10 & 1201 & 0502 \\ + & 00 & 0001 & 0000 \\ \hline & 10 & 1202 & 0502 \end{array}$$

We refer to this method of converting one instruction into the following one as "stepping". The instruction in 0505 stores the new instruction in location 0501 and closes the program loop by going to 0500 for the next instruction.

Again we enter the partial sum, Σ_1, into the accumulator, and we perform the instruction in 0501, which now adds a_2 to the partial sum (the data address is 1202 instead of 1201). This operation gives a new partial sum, $\Sigma_2 = \Sigma_1 + a_2$, which we store back in 1500.

The brackets enclosing the instruction in 0501 indicate that the instruction varies each time around the loop.

When 100 numbers have been accumulated, i.e. after a_{100} in cell 1300 has been added to the partial sum, we must terminate the loop. With the terminating box added to our flow chart, we have:



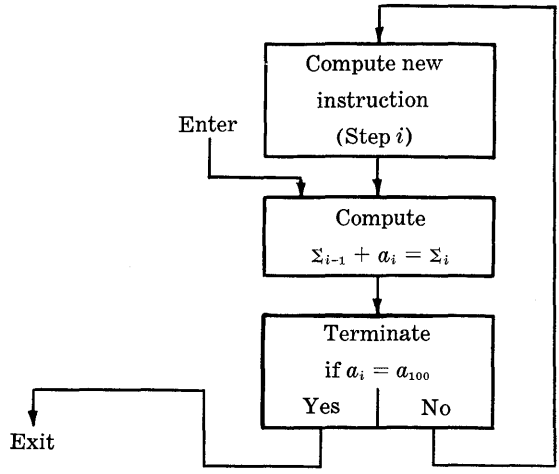
We could terminate the loop by setting up a counter as we did on page 18; however, the following alternative method is generally preferable. We note that the instruction in cell 0501 is changed each time a term is added, and when the 100th term has been added the instruction will read (10 1300 0502). This instruction itself is used as a criterion for termination rather than the setting-up and stepping of a special counter for the purpose. Each time around the loop we subtract from the instruction its terminal value (10 1300 0502) and test for zero. The instructions now include the termination procedure:

0500	RAU	60	1500	0501	
0501	AU	[10	1201	0502]	Compute $\Sigma_{i-1} + a_i = \Sigma_i$
0502	STU	21	1500	0503	
0503	RAU	60	0501	0504	
0504	SU	11	0601	0505	Terminate if $a_i = a_{100}$
0505	BRNZU	44	0506	Exit	
0506	RAU	60	0501	0507	Compute
0507	AU	10	0600	0508	new instruction
0508	STU	21	0501	0500	(step i)
0600		00	0001	0000	Stepping constant
0601		10	1300	0502	Terminating constant

The instruction in 0503 enters the variable instruction in 0501 into the accumulator, and we subtract the terminal value of the variable instruction which is stored in 0601. If the result is zero, we know that the last term has just been

added to the sum and we can terminate the loop. On the other hand, if the result is not zero, we must continue to add terms to the sum. We use a "branch on non-zero in the upper" instruction to test for a zero condition. Finally, when the variable instruction becomes (10 1300 0502) the BRNZU test will cause the next instruction to be taken from the location that we have designated "exit", which terminates the loop.

Generally we draw our flow chart in the following form:

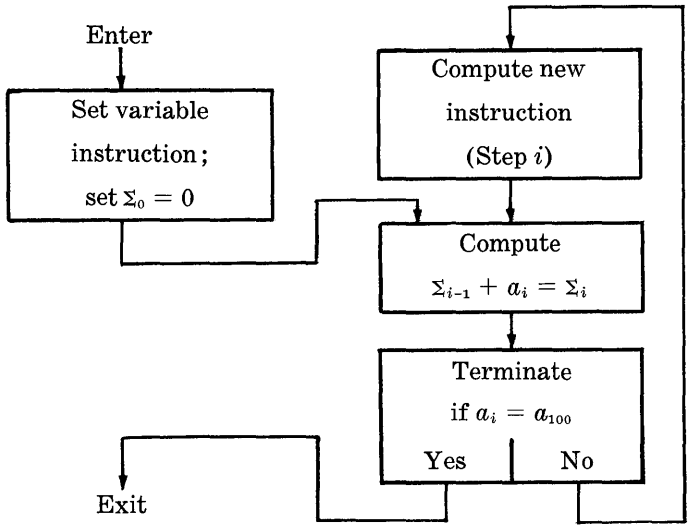


The coding and the order of execution are exactly the same as before; only the relative positions of the boxes in the flow chart differ from those in the preceding flow chart.

The conventions about flow charts that we establish here serve several purposes. One of the aims is to attach as much significance as possible to the relative position of the boxes on the two-dimensional chart and to reduce the significance of lines and arrow heads to a minimum.

4. "SETTING" THE INITIAL CONDITIONS FOR THE PROGRAM LOOP

We assumed that the summation cell, 1500, contained zero and that the variable instruction in 0501 was (10 1201 0502) at the beginning of the program. To make sure that these conditions are satisfied, we must add an "initializing" or "setting" box to our flow chart and write instructions that will enter zeros into 1500 and enter (10 1201 0502) into 0501. Now we have the following flow chart:

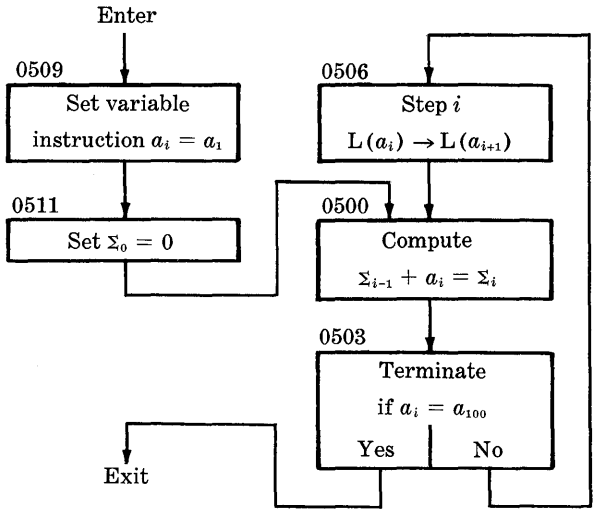


We must add the following instructions:

0509	RAU	60	0602	0510	Set variable
0510	STU	21	0501	0511	instruction
0511	RAU	60	0603	0512	
0512	STU	21	1500	0500	Set $\Sigma_0 = 0$
0602		10	1201	0502	
0603		00	0000	0000	Constants

At the beginning of the program we transfer the initial value of the variable instruction, which we have stored in 0602 as a program constant, into 0501 with RAU and STU instructions (0509 and 0510). Next we transfer zeros, which have been stored in 0603, into 1500, the summation cell. Now that we are ready to begin the accumulation, we go to the instruction in 0500. The final flow chart and a complete listing of the instructions for the accumulation problem are given below. In the flow chart we speak of the level on the left as the "open" level because it is not a loop, or the "problem" level because it is executed once per problem. Note that the number above the left-hand corner of a box in the flow chart indicates the location of the first of the group of instructions corresponding to that box.

Accumulation example: $S = \Sigma_{i=1}^{100} a_i$
a_1 in 1201, a_2 in 1202, . . . , a_{100} in 1300; S in 1500



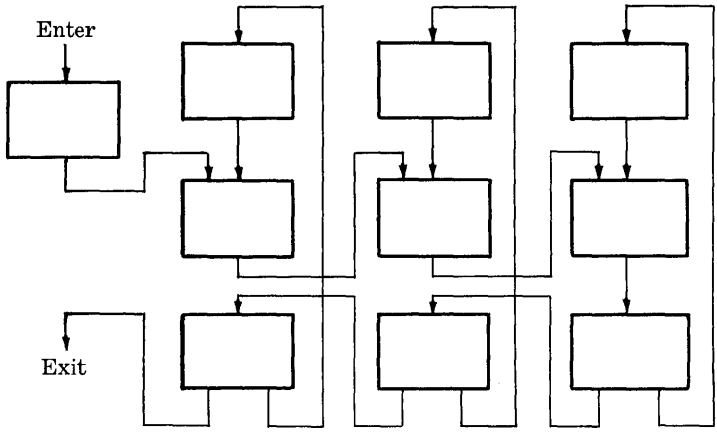
0500	RAU	60	1500	0501	Compute $\Sigma_{i-1} + a_i = \Sigma_i$
0501	AU	[10	1201	0502]	
0502	STU	21	1500	0503	
0503	RAU	60	0501	0504	Terminate if $a_i = a_{100}$
0504	SU	11	0600	0505	
0505	BRNZU	44	0506	0513	
0506	RAU	60	0501	0507	Step a_i to a_{i+1} (compute new instruction)
0507	AU	10	0601	0508	
0508	STU	21	0501	0500	
0509	RAU	60	0602	0510	Set variable instruction $a_i = a_1$
0510	STU	21	0501	0511	
0511	RAU	60	0603	0512	Set $\Sigma_0 = 0$
0512	STU	21	1500	0500	
0600		10	1300	0502	Terminating constant
0601		00	0001	0000	Stepping constant
0602		10	1201	0502	Setting constant
0603		00	0000	0000	Zero setting constant

5. THE BASIC FORM

The preceding flow chart is drawn as a "basic form". It has two vertical levels. The right-hand level is called the "term level" or the "i level". We go through this level once for each term, a_i, that we wish to accumulate. The left-hand level is called the problem level, and the instructions on this

level are executed once for each solution of the problem. The "frequency of execution" is 100 for the i level and one for the problem level.

Generally a basic form consists of a number of levels arranged from right to left by decreasing frequency:



The execution of a level proceeds from top to bottom. The loop on a level is closed by a line running upward on the right of the level. Boxes are entered at the top; lines emerging from a box leave at the bottom of the box.

We can see that in the accumulation program the setting box of the level on the left "controls" the level to the right in the following sense. The setting of the variable instruction in 0501 to (10 1201 0502) causes the "term level" to accumulate 100 terms. If, on the other hand, the variable instruction had been set to (10 1220 0502), only 81 terms, $a_{20} - a_{100}$, would be included in the sum. Similarly, if a number N had been transferred to the summation cell instead of zero prior to the accumulation, the term level would produce the sum $(S' = N + \Sigma a_i)$. The term level, on the other hand, is not capable of affecting the level to its left in a similar manner.

The coding of a basic form is begun in the middle of the rightmost level (box 0500 in the accumulation problem), which represents the logical core of the problem. The symbolic expression for accumulation $(\Sigma_{i-1} + a_i = \Sigma_i)$ is the fundamental concept in the statement of the problem. The rightmost level is completed by the coding of the terminating and stepping boxes (0503 and 0506). Finally, in the level on the left one codes the setting boxes (0509 and 0511) for the level on the right.

This procedure covers the two-level basic form of the accumulation program. For a larger basic form one repeats this procedure for additional levels, one level at a time, proceeding from right to left.

6. EXAMPLE OF A THREE-LEVEL BASIC FORM

Let us program and code the following problem as an example of a three-level basic form.

Problem: Compute values of the function $(f(x, y) = x \cdot y)$ for all combinations of the variables, $x = x_1, x_2, \ldots, x_{10}$ and $y = y_1, y_2, \ldots, y_{10}$.

Let us assign the following memory locations for the variables:

x_1 in 1201	y_1 in 1301	$x_i y_1$ in 1401
x_2 in 1202	y_2 in 1302	$x_i y_2$ in 1402
. . . . .	. . . . .	. . . . .
x_{10} in 1210	y_{10} in 1310	$x_i y_{10}$ in 1410

There will be 100 values of the product and we shall compute them in the following order:

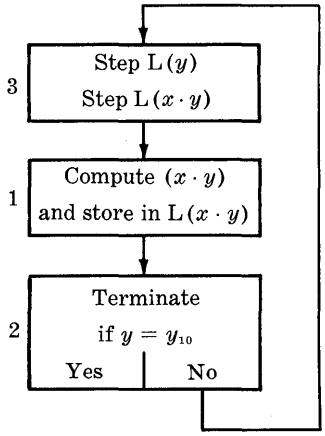
$$x_1\ (y_1, y_2, \ldots, y_{10})$$
$$x_2\ (y_1, y_2, \ldots, y_{10})$$
$$\ldots\ldots\ldots\ldots$$
$$x_{10}\ (y_1, y_2, \ldots, y_{10}).$$

As the products in the first line $(x_1y_1, x_1y_2, \ldots, x_1y_{10})$ are computed, they will be stored in the memory and they will be punched out when the line is completed; similarly, the second line will be computed, stored, and punched out, and so on for all 10 lines.

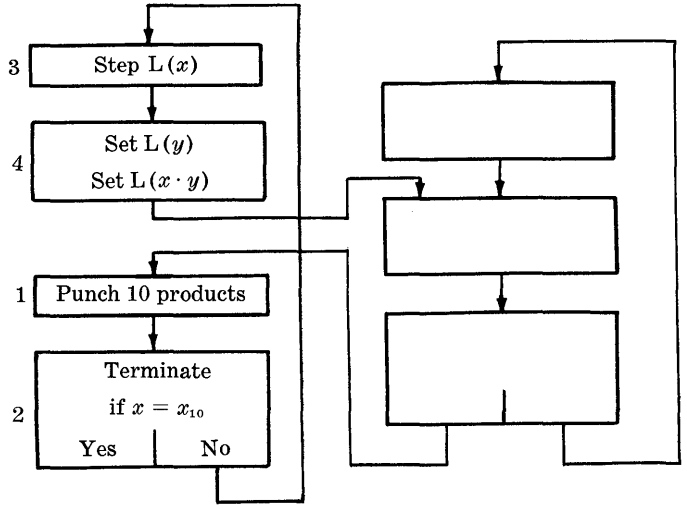
The most frequently repeated operations will be the forming and storing of a product; this operation will be done 10 times for each line. As each product is computed and stored, it is necessary to go to a new storage location for y and to a new one for storing $(x \cdot y)$, i.e. $\mathrm{L}(y)$ and $\mathrm{L}(x \cdot y)$ must be stepped. It is also necessary to look at the subscript of y each time a product is computed in order to determine whether or not the end of the line (y_{10}) has been reached. The rightmost level of the flow chart consists of the following three boxes:

1. Compute $(x \cdot y)$ and store in $\mathrm{L}(x \cdot y)$
2. Terminate if $y = y_{10}$
3. Step $\mathrm{L}(y)$ and step $\mathrm{L}(x \cdot y)$

The three boxes are drawn as follows:

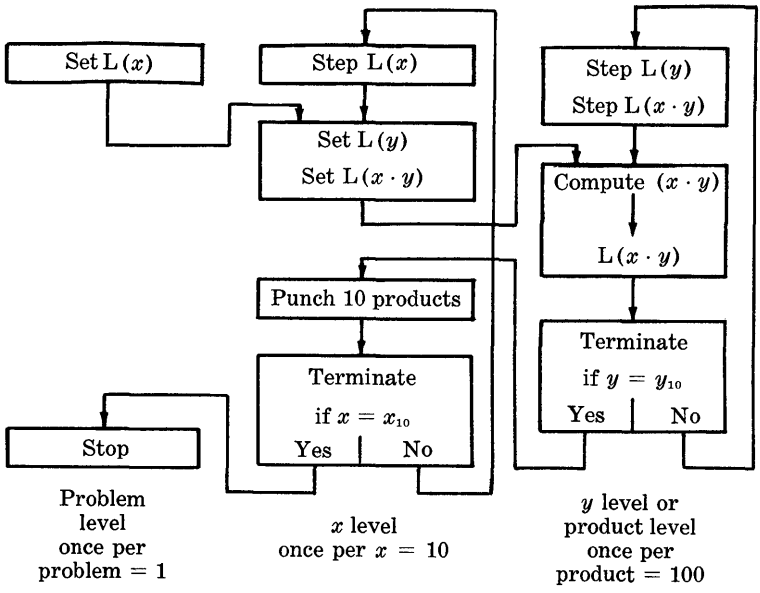


When 10 products have been computed, we want to punch them out and proceed to the next line of the problem, i.e. take the next value of x and compute 10 more products as before. The starting of the next line involves stepping $L(x)$ and setting y back to its initial value, y_1. Here, again, before we step $L(x)$, we must look at the subscript of x to see if we have finished the last line of the computation. We can now add the four boxes of the second level from the right of the flow chart in the following order:



1. Punch 10 products
2. Terminate if $x = x_{10}$
3. Step L(x)
4. Set L(y), set L($x \cdot y$)

The third level consists of starting the problem by setting x at x_1, and of stopping when the last or x_{10} line of the computation has been completed.



We can now see how the program is executed chronologically. The computation starts on the problem level with the setting of $x = x_1$; it proceeds to the x level where y is set to y_1 and L($x \cdot y$) is set to L($x_1 \cdot y_1$); it then goes to the y level where x_1y_1 is computed and stored. When 10 products have been produced, the procedure goes to the left for punching and for testing to see if the problem is finished; if it is not finished, L(x) is stepped and L(y) and L($x \cdot y$) are set for the next 10 products. When 100 products have been computed and punched, the program returns to the problem level and stops.

The coding proceeds as follows:

0500	RAU	[60	1201	0501]	Compute ($x \cdot y$)
0501	MPY	[19	1301	0502]	
0502	STU	[21	1401	0503]	

0503	RAU	60	0501	0504	
0504	SU	11	0600	0505	Terminate
0505	BRNZU	44	0506	0512	if $y = y_{10}$
0506	RAU	60	0501	0507	
0507	AU	10	0601	0508	Step L(y)
0508	STU	21	0501	0509	
0509	RAU	60	0502	0510	
0510	AU	10	0601	0511	Step L($x \cdot y$)
0511	STU	21	0502	0500	
0512	RAU	60	0602	0513	Punch
0513	SPOP	69	0514	0061	10 products
0514	RAU	60	0500	0515	
0515	SU	11	0603	0516	Terminate
0516	BRNZU	44	0517	0531	if $x = x_{10}$
0517	RAU	60	0500	0518	
0518	AU	10	0601	0519	Step L(x)
0519	STU	21	0500	0520	
0520	RAU	60	0602	0521	
0521	AU	10	0604	0522	Step
0522	STU	21	0602	0523	x identification
0523	RAU	60	0605	0524	
0524	STU	21	0501	0525	Set L(y)
0525	RAU	60	0606	0526	
0526	STU	21	0502	0500	Set L($x \cdot y$)
0527	RAU	60	0607	0528	
0528	STU	21	0500	0529	Set x
0529	RAU	60	0608	0530	Set
0530	STU	21	0602	0523	x identification
0531		01	0000	0000	Stop
0600		19	1310	0502	
0601		00	0001	0000	
0602		[10	1401	5001]	
0603		60	1210	0501	
0604		00	0000	0001	
0605		19	1301	0502	
0606		21	1401	0503	
0607		60	1201	0501	
0608		10	1401	5001	

The order of the coding is the same as that in which it is written. The compute box is represented by instructions 0500-0502, which enter x into the accumulator, multiply x by y, and store the product. Since these three instructions are all variable, they are enclosed in brackets; their initial values are written within the brackets on the coding sheet.

The instruction in 0501, which selects y for each product, will be involved in three other boxes in the flow chart: "terminate when $y = y_{10}$", "step L(y)", and "set L(y)". Instructions 0503-0504 for the terminate box compare the current value of the variable instruction 0501 with its final value (19 1310 0502) which is stored in 0600; if $y \neq y_{10}$, the BRNZU instruction directs the program to the "step L(y)" box which begins at 0506. In the "step L(y)" box the current value of the instruction in 0501 is stepped by the addition of (00 0001 0000) from location 0601. Similarly, L($x \cdot y$) is stepped by the addition of (00 0001 0000) to the current value of the STU instruction in 0502.

The punching of the 10 products starts with the instruction in 0512, the address to which the branch instruction in 0505 sends us when $y = y_{10}$. The first part of α for the punch-out instruction is (10 1401 5) since there are 10 products in locations starting at 1401 to be punched five to a card. The last three digits of α are used for identification; for this identification we shall use the subscript of x that is involved in the products being punched. The initial value of α, therefore, is (10 1401 5001) which is stored in 0602. Since the value of α varies with x, it will be stepped when x is stepped and set when x is set.

The termination on x and the stepping on x are coded in an analogous manner to those on y. The initial instruction to be changed is in 0500 and the final one for termination is in 0603. The stepping constant is the same as for y (0601).

The stepping of the x identification involves α from 0602 and its stepping constant in 0604.

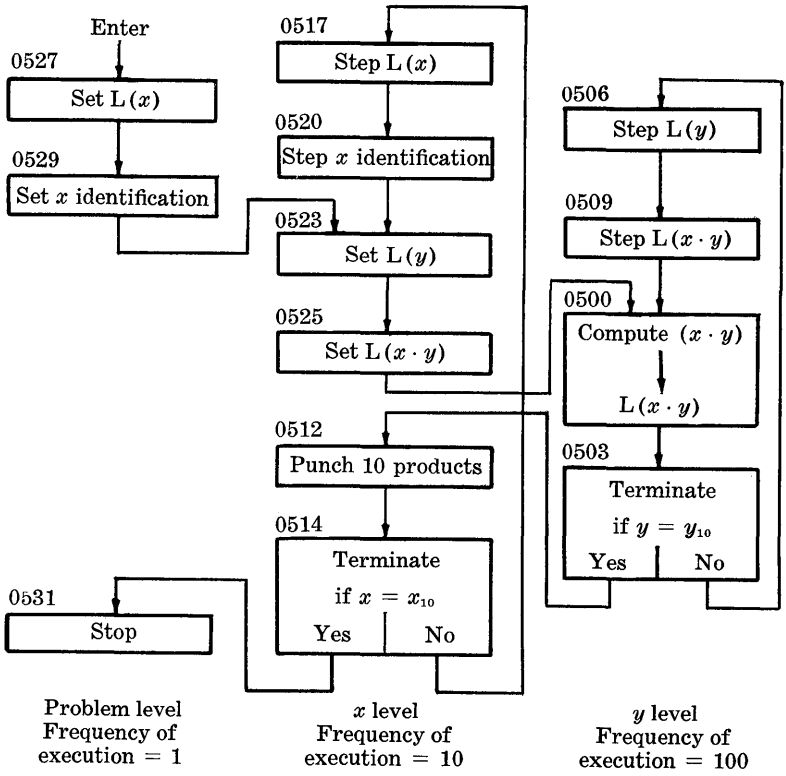
To set L(y) we transfer the initial value of the y-instruction from 0605 to 0501; to set L($x \cdot y$) we transfer the initial store instruction from 0606 to 0502; to set L(x) we transfer the initial location of x from 0607 to 0500; and to set x identification we transfer the initial value of α from 0608 to 0602.

To stop the program when the last line of the computations is finished, we fill in the address of the stop instruction (0531) as the instruction address of 0516.

As we code the instructions, we also draw a final detailed

flow chart. This final flow chart, which is drawn below, is very useful in testing and in locating errors in the program.

x_1 in 1201	y_1 in 1301	$x_i y_1$ in 1401
x_2 in 1202	y_2 in 1302	$x_i y_2$ in 1402
.	.	.
.	.	.
.	.	.
x_{10} in 1210	y_{10} in 1310	$x_i y_{10}$ in 1410



7. SQUARE-ROOT PROBLEM

The determination of the square root of a number illustrates a useful method of terminating a loop. In this example we shall terminate on the accuracy of a number. We have already discussed other methods of terminating in this chapter.

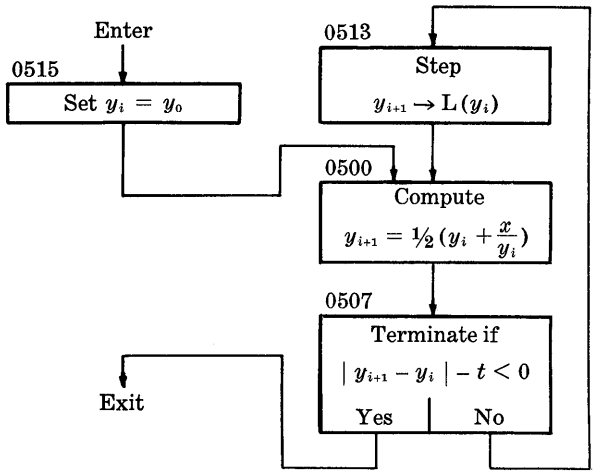
If we are given a number, x, and we want to determine y where $y = \sqrt{x}$, we use Newton's iteration formula:

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

The desired accuracy is obtained when the difference between two successive iterations is less than a predetermined tolerance, t, i.e. terminate if

$$|\, y_{i+1} - y_i \,| - t < 0.$$

The basic form for the solution of this problem is the following:



The symbols, $y_{i+1} \rightarrow L(y_i)$, which indicate the stepping procedure, mean, "Transfer the new approximation, y_{i+1}, into the location of the old value of y_i".

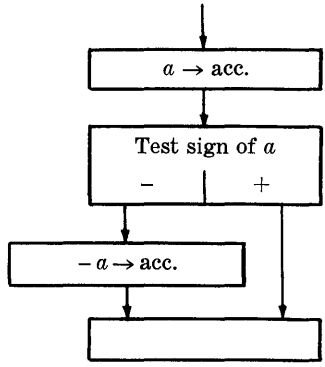
The program for this problem is the following:

0500	RAU	60	0600	0501	
0501	DIVR	64	0601	0502	
0502	STL	20	0602	0503	Compute
0503	RAU	60	0602	0504	$y_{i+1} = \frac{1}{2}\left(y_i + \frac{x}{y_i}\right)$
0504	AU	10	0601	0505	
0505	MPY	19	0603	0506	
0506	STU	21	0604	0507	
0507	SU	11	0601	0508	
0508	BR–	46	0509	0511	
0509	STU	21	0602	0510	Terminate on
0510	RSU	61	0602	0511	$\lvert y_{i+1} - y_i \rvert < t$
0511	SU	11	0605	0512	
0512	BR–	46	Exit	0513	

0513	RAU	60	0604	0514	Store
0514	STU	21	0601	0500	y_{i+1} in L(y_i)
0515	RAU	60	0606	0516	Set
0516	STU	21	0601	0500	$y_i = y_0$
0600		[		]	x
0601		[		]	y_i
0602		[		]	Temporary storage cell
0603		50	0000	0000	Constant
0604		[		]	y_{i+1}
0605		[		]	t
0606		[		]	y_0

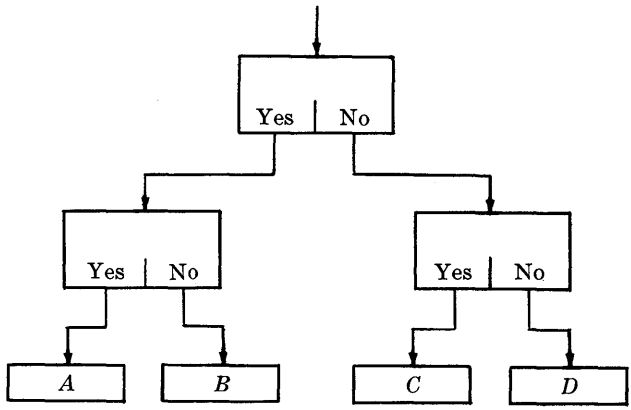
At the outset of the problem the radicand (x), the tolerance (t), and the first approximation (y_0) are located in 0600, 0605, and 0606, respectively. The program constant (50 0000 0000) is located in 0603. The locations 0601, 0602, and 0604 are used as work cells in the course of the problem. A word transfer of the first approximation (y_0) to L(y_i) takes place when y_0 is used in the first evaluation of the formula (0515). On succeeding iterations y_{i+1}, the result of the formula evaluation is used as the approximation in the next formula evaluation. Finally, the loop is terminated when the difference between y_{i+1} and y_i is less than the tolerance, t. The word transfer, y_{i+1} to L(y_i), occurs at the top of the loop (0513).

Note that a branch operation was required in order to obtain an absolute value. The procedure to be followed to find the absolute value of a number, a, can be illustrated in the following way:



The instructions that represent this illustration are found in locations 0508-0510 of the square-root program.

The problem of finding the absolute value of a number is a simple example of the use of a branch operation for some process other than terminating. It often happens that a single branch operation or a network of branch operations is an integral part of the solution of a problem. For example, it might be necessary to choose one of four different computational procedures, *A*, *B*, *C*, *D*, depending on certain conditions; in this case a network of branch operations similar to the following one, might be used:



8. PROGRAMMING AND CODING

The preceding sections of this chapter have been devoted to problems that essentially consist of one phase. That is, we had data in memory; we performed certain simple arithmetical operations on these data; and we produced new data. The basic form was used to represent this transformation of data.

Most computational problems will consist of more than one phase. One of the aspects of programming in this System as outlined in the Introduction involves connecting the different phases of a problem. We then have a whole picture of the computational problem. In order to fix our ideas, let us consider a particular problem:

Solve the set of equations:

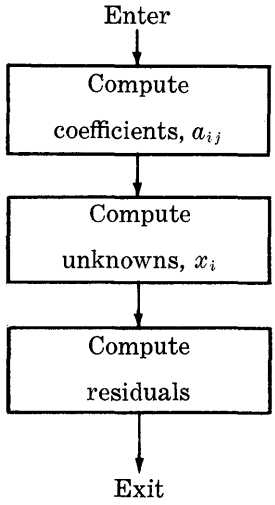
$$\begin{aligned} a_{11}\,x_1 + a_{12}\,x_2 + \ldots + a_{1n}\,x_n &= y_1 \\ a_{21}\,x_1 + a_{22}\,x_2 + \ldots + a_{2n}\,x_n &= y_2 \\ \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot & \\ a_{n1}\,x_1 + a_{n2}\,x_2 + \ldots + a_{nn}\,x_n &= y_n \end{aligned}$$

where the coefficients a_{ij} are functions of known quantities u_i, v_j, i.e.

$$a_{ij} = u_i \cdot v_j.$$

We are required to substitute the unknowns back in the equations to determine the residuals.

If we assume that the computational method is established and memory locations are assigned, we can proceed with the programming of the problem. There are three distinct phases in the solution of this problem, and we can represent the phases by boxes in a flow chart:



We investigate the first box and quickly determine that it can be represented by a basic form of three levels (see Section 6). A rough flow chart of the first phase is now drawn. The predetermined memory locations for the output of the first phase will contain the input for the second phase.

The second is represented by one or two distinct basic forms depending on the computational method selected. The third phase, in which the residuals are computed, is also represented by a basic form.

It should be noted that a phase of a problem may be represented by a network of branch operations.

The coding begins with the set of rough flow charts. We take one flow chart at a time and write the corresponding instructions. In conjunction with writing the instructions, we draw the final detailed flow charts.

9. TIMING

In planning problems it is important to estimate with reasonable accuracy the running time that the machine will require for a particular computation. Such estimates are helpful in the choice of possible alternative solutions of a problem, in the decision of whether or not a problem is worth the necessary machine time, and in effective scheduling of machine use.

The time required for the execution of an instruction can be divided into two parts: the time required to actually perform the operation specified by the operation code, and the time to secure the data and the next instruction from memory.

The time required to perform an operation depends on the particular operation. Each of the operations RAU, RSU, AU, SU, STU, STL, BRNZU, BR–, and STOP requires approximately 0.4 millisecond (0.0004 second) ; on the average, multiplications require 10 ms, divisions require 15 ms, and shift operations require 2.5 ms. The execution times for special operations are listed in Appendix I.

To secure a data word or an instruction word can take from 0 ms to 5.0 ms, i.e. the average time is 2.5 ms. This time can be reduced to a minimum by a method called "optimum coding" (see Chapter VI).

For programs that do not use optimum coding and that do not contain an unusually large number of multiplications and divisions, we can obtain a fairly accurate (within 10 per cent) time estimate by allowing 5 milliseconds for the execution of each instruction. This estimate will be good enough for programs that are expected to run less than an hour; however, a longer program will require a more careful investigation since 10 per cent of the running time will amount to a considerable length of time.

Exercise 2.

Draw a flow chart, and write instructions that will compute:

$$x_i + y_i = z_i, \text{ where } i = 1, 2, \ldots, 50$$

Read the data into the following memory cells:

x_1 in 1201	y_1 in 1301
x_2 in 1202	y_2 in 1302
. . . . .	. . . . .
x_{50} in 1250	y_{50} in 1350

Assume that the x_i's are punched in 10 cards, 5 words per card, and that the y_i's are punched in 10 cards, 5 words per card.

Store the z_i's in the following locations:

z_1 in 1401
z_2 in 1402
. . . . .
z_{50} in 1450

After all of the z_i's have been computed, punch them into 10 cards, 5 words per card.

IV. Precision and Scaling

1. ACCUMULATION OF ERRORS

Most calculating problems are first stated as a set of mathematical equations and the method of solution is indicated in the same form. An "equivalent" numerical procedure is then evolved in which certain of the variables are represented by numerical values and the relations between these values are represented by arithmetical operations. Because of the nature of the initial numbers in the calculation and of the arithmetical processes, the mathematical equations and the numerical procedure cannot be assumed to be exactly equivalent. In planning a problem, it is necessary to justify not only the validity of the mathematical equations but the extent to which the numerical process is equivalent to the mathematical equations.

To illustrate, let us consider a simple physical measurement. The length and width of a rectangle are each measured to the nearest millimeter, and the following results are recorded:

$$l = 292 \text{ mm.} \qquad w = 103 \text{ mm.}$$

At the time of measurement the observer had difficulty in deciding whether to record 103 or 104; he recorded 103. From these recorded measurements, it is required to compute the area of the rectangle. The mathematical expression is

$$A = l \times w.$$

The numerical equivalent is

$$A = 292 \times 103 = 30076 \text{ sq. mm.}$$

Here the mathematical expression is exact and the arithmetic is without error, but we should not conclude that the area of the rectangle is 30076 square millimeters. Had the observer written 104 for the width, the computed area would

have been 30368, which differs from the above result by 292 square millimeters. The last two digits of the computed area really tell us nothing about the area of the rectangle except the position of the decimal point. In the theory of measurement and in numerical analysis, a distinction is made between digits that locate the decimal point and those that have additional significance; we use the term "significant" digits for the latter. A number, A, is written in the form

$$A = a \times 10^{\alpha}$$

where a contains the significant digits and the integer α specifies the location of the decimal. The number of digits in a is the precision of A. The exponent α specifies the magnitude of A. In the above example we would write for the area

$$A = 301 \times 10^{2} \text{ sq. mm.}$$

The true value may differ by a unit or two from this value, but the "1" has significance. The value of α, of course, depends upon the units that are employed. For example, we can express the above measurements in centimeters or meters:

$$l = 292 \times 10^{0} \text{ mm.} = 292 \times 10^{-1} \text{ cm.} = 292 \times 10^{-3} \text{ m.}$$
$$A = 301 \times 10^{2} \text{ sq. mm.} = 301 \times 10^{0} \text{ sq. cm.}$$
$$= 301 \times 10^{-4} \text{ sq. m.}$$

In numerical analysis it is convenient to associate with each number, x, an error, $\mathrm{E}(x)$, where $\mathrm{E}(x)$ is the amount by which the number differs from the true value of the quantity that it represents. For example, when the quantity, π, is rounded to a given number of places,

$$\pi = 3.14 + E(\pi),$$

$\mathrm{E}(\pi) = .00159\cdots$. In this case we know the value of $\mathrm{E}(\pi)$, but in many cases we can only give limits to E. When a number is rounded, we know that the error is less than half a unit in the last place.

In planning a calculation, one should consider where each initial number came from and assign to it an E. When an arithmetical operation is performed on two such numbers, it is possible to evaluate the E of the result from the following relations:

$$a + b = c \qquad \mathrm{E}(c) = \mathrm{E}(a) + \mathrm{E}(b)$$
$$a - b = c \qquad \mathrm{E}(c) = \mathrm{E}(a) - \mathrm{E}(b)$$
$$a \times b = c \qquad \mathrm{E}(c) = a \cdot \mathrm{E}(b) + b \cdot \mathrm{E}(a)$$
$$a \div b = c \qquad \mathrm{E}(c) = [b \cdot \mathrm{E}(a) - a \cdot \mathrm{E}(b)]/b^{2}$$

In the above example,

$$l = 292, w = 103 \qquad E(l) = E(w) = 1$$
$$A = l \times w = 30076 \qquad E(A) = l \cdot E(w) + w \cdot E(l) = 395$$

i.e. if the error in each measurement can be as great as 1 mm., the error in the computed area can be as large as 395 sq. mm. In deriving the error equations, we assumed that $E(a)$ and $E(b)$ are small compared with a and b, respectively.

Starting with the assigned errors of the initial numbers and using the error equations, we could trace the precision in each operation to the final results and give an estimate of their reliability. This process is not generally carried out in practice, but by keeping the method in mind we can form some estimate of what is happening and avoid some of the worst pitfalls in planning a problem. We note, for example, that in addition and subtraction the error, $E(c)$, is never greater than the sum of errors $[E(a) + E(b)]$. However, when two nearly equal numbers are subtracted, several digits on the left will disappear and the number of significant figures will be reduced. The problem of small divisors is also serious since the error varies inversely as the square of the divisor. A single-digit divisor can give a large quotient whose first digit at most will be significant, and, of course, division by zero is excluded. Thus it is possible to prescribe a sequence of arithmetical operations that will lose all of the precision of the initial data and give results that are completely fictitious, because the problem itself is not capable of solution, e.g. trying to solve a set of linear equations with a zero (or very small) determinant. Again, the problem may be soluble, but care must be exercised in the general layout of the calculation to prevent undue loss of precision. It is not the purpose of this memorandum to examine these questions, except to point out their existence and to emphasize that, by keeping track of the expected errors of each stage of the calculation, the computer can recognize unsatisfactory conditions in the problem.

To guard against unnecessary loss of precision in the individual arithmetical operations, the manner in which the numbers are entered into the accumulator must be considered. It should be remembered that:

1) Numbers to be added into the accumulator must have the decimal points aligned.
2) When numbers are added into the left-hand position of the accumulator, the "carry" may "spill" over.

3) The divisor must be greater than the dividend; if not, the machine will stop in division.
4) When the multiplicand and multiplier are too far to the right, we lose precision because all of the significant figures of the product will not appear in the upper accumulator. For example:

Multiplicand	0 0 0 1 1 1 1 1 1 1
Multiplier	0 0 0 2 2 2 2 2 2 2
Product	0 0 0 0 0 0 0 2 4 6 | 9 1 3 . . . . .

The general rule should therefore be to keep all numbers as far to the left as possible without causing an overflow.

The process of shifting input data and intermediate results to yield final results with the desired precision is called "scaling". This shifting may be planned by the coder and written into the machine instructions ("fixed-point calculation") or it may be done automatically by the machine ("floating-point calculation").

2. FIXED-POINT CALCULATIONS

Whenever we perform arithmetical equations on numbers, we keep the numbers as far to the left as possible to preserve maximum precision. The relative position of numbers in memory cells is under the control of the program, and, when there is uniformity in the range of numbers, we can write a program that is consistent with all of the numbers. For example, if the numbers in a problem are the following angles, θ, ranging from zero to 2π, we would say that their range is uniform:

0.2504. . . . .
0.5269. . . . .
0.7324. . . . .
0.9887. . . . .
1.2784. . . . .
.
.
.
6.2837. . . . .

A program that is written to operate on these numbers would generally consist of arithmetical and shifting operations in the computing box of a loop. Each time the machine progresses through the loop, an angle, θ, is chosen and $f(\theta)$ is computed. The arithmetical and shift operations must be consistent with all of the angles, the smallest as well as the largest. That is, the program must be written so that, for each angle, there is no overflow in addition, a dividend and divisor must satisfy the conditions necessary for division, etc.

In general, when the machine performs the arithmetical and shifting operations, there will be a loss of precision which is considered as an error and must be kept within tolerable limits. In fixed-point calculations we can compute the error since we know all of the operations in the program and the range of the numbers. Our convention for keeping track of the necessary shifts on the coding sheets is explained in Section 5 of this chapter.

3. FLOATING-POINT CALCULATIONS

Problems do occur, however, in which the range of data is not uniform. For example, let us suppose that we have the following data:

48916.278
1678.3214
207.43098
37.003290
1.4365026
.61986543
.094217899
.
.
.

The data range from (5×10^4) to (9×10^{-2}), but the relative precision or significance is the same in each case, i.e. each number contains eight significant digits. It would be impossible to perform repeated fixed-point calculations on this block of data, since a 14-digit memory cell (five digits to the left of the decimal point and nine to the right) would be required to store each number with the decimal point fixed in the same relative position. In order to accommodate these data in memory, they must be stored in the following form:

48916.278
1678.3214
207.43098
37.003290
1.4365026
.61986543
.094217899

Now the position of the decimal point is different for each number in the block. The arithmetic used on these shifted data is called "floating point". The necessary shifts are different for each number in the block and they are performed automatically.

Each number in a floating-point operation must also have a tag called an exponent that will determine what shifts are to be performed. The numbers are written in the form $(A = a \times 10^{\alpha})$, with the convention that $(0.1 \leq a < 1.0)$. For example,

$$76.345039 = .76345039 \times 10^{2}$$
$$.00076345039 = .76345039 \times 10^{-3}$$

The number, a, and the exponent, α, are stored in memory as a word. To avoid negative exponents when the word is in memory, we arbitrarily add 50 to each exponent. We also write the adjusted exponent to the right of the number:

76345039 52
76345039 47

The machine looks at the exponent, performs shifts depending on the arithmetical operation called for, performs the arithmetic, and gives the result in the same form.

In some calculators floating-point operations are built into the machine and in others they are performed by means of subroutines. Such subroutines are included in the System, and directions for their use are contained in Appendix II.

Floating-point operation preserves maximum precision in each arithmetical operation without any attention on the part of the coder. The method has, however, the following disadvantages:

1. Since two digits of each number are used for decimal indication, there remain only eight significant figures instead of ten.
2. Running time on the machine is increased by a *factor of three.*
3. Since the coder is unaware of the shifts made by the

machine, there could be serious loss of precision without his being aware of it (see p. 40).

If the coder does not know for *a priori* reasons that the arithmetical process is legitimate, a detailed examination of the process is necessary.

4. DOUBLE-PRECISION ARITHMETIC

When the range of data is uniform and the precision exceeds the word size of a memory cell, more than one cell can be used for each number. If the precision of the data is not more than 20 digits, two memory cells can store each number and 20-digit, or double-precision, arithmetic can be used. If there is a considerable loss in precision (e.g. from subtracting two nearly equal numbers), the use of double-precision arithmetic sometimes will overcome the difficulty. Double-precision operations can be handled by subroutines in which each 20-digit number is contained in two memory cells. Such subroutines are included in the System and they are described in Appendix II. Double-precision fixed-decimal operation obviously permits the storage of only half as many numbers as single-precision work, and it increases the running time by a factor of about 2.5. The procedure can be extended, of course, to triple or higher precision.

When both high precision and automatic scaling are necessary, it is possible to perform multiple-precision floating-point arithmetic; this type has not been included in the System.

It is not essential that an entire calculation be executed in fixed-point, floating-point, or in double-precision arithmetic if proper provisions are made for the junctions of the systems. For example, if both floating point and fixed point are used in the same problem, it will be necessary to use a conversion subroutine to change floating-point to fixed-point numbers. The necessary conversion subroutines are described in Appendix II. It should be emphasized that one would seldom use floating-point arithmetic for red-tape instructions, i.e. stepping, setting, terminating.

5. NOTATION FOR FIXED-POINT SCALING

As mentioned in a previous paragraph, the general rule of scaling is to keep the numbers as far to the left as possible in the accumulator and still avoid overflow. Moreover, we wish to use the same scaling for an entire block of numbers, especially in coding a loop.

In fixed-decimal computation we think of each number in the form ($A = a \times 10^{\alpha}$); only a is actually stored in the memory. To keep track of the shifting, we write on our coding sheet the scale factor, n, where

$$A \times 10^{n} = a, \quad \text{i.e.} \quad n = -\alpha$$

and the left-hand side of the equation is abbreviated as $A\,(n)$. For consistency and ease of notation, we think of each memory cell and the accumulator as having a decimal point on the left. The decimal point remains fixed.

Suppose now that we want to perform the following computation:

$$A^2 + AB$$

where

$$A = 44.9876$$
$$B = 5.6321$$

or, according to the convention we have established,

$$A\,(2) = .4498760000$$
$$B\,(1) = .5632100000$$

Our problem is to code the computation and include the proper scaling. On our coding sheet we would write the following information and instructions. The information on the left indicates what the instruction on the right has accomplished.

$A\,(2)$	0700	RAU	60	L(A)	0701	.4498760000
$A\,(2) \cdot A\,(2) = (A^2)\,(4)$	0701	MPY	19	L(A)	0702	.2023884153
$(A^2)\,(4) \rightarrow$ L(A^2)	0702	STU	21	L(A^2)	0703	
$A\,(2)$	0703	RAU	60	L(A)	0704	.4498760000
$A\,(2) \cdot B\,(1) = (AB)\,(3)$	0704	MPY	19	L(B)	0705	.2533746619
$(AB)\,(4)$	0705	SHRT	30	0001	0706	.0253374661
$(A^2 + AB)\,(4)$	0706	AU	10	L(A^2)	0707	.2277258814

The sum standing in the accumulator, .2277258814, should read 2277.258814, according to the scale factor (4).

To locate the decimal point after a multiplication or a division, we follow the usual laws of exponents, i.e. the exponents are added in multiplication and subtracted in division. When we add or subtract two numbers, we must be sure that their exponents are the same, i.e. that their decimal points are aligned. If they are not the same, we must shift accordingly and adjust the exponents.

Since we could easily foresee in the above example the

size of the numbers generated (A^2 and AB), we knew that no overflow would result when they were added together. In general the formula ($A^2 + AB$) would be evaluated for many values of A and B, and the data must be examined to determine whether or not the above coding would be correct for all cases. If the coding is not correct for all cases, it must be changed to fit all values of A and B.

An analysis of the size of the numbers generated during a computation must be made prior to coding. The coder then knows the "worst" possible case and can take care of it in his coding. Although this analysis is an added chore for the coder, it is this analysis that tells him the reliability of his results.

Exercise 3.

Write instructions to compute:

$$\frac{k^2x}{r^3} \text{ where } r^2 = x^2 + y^2$$

for

x in the range $5 \leq x \leq 10$
y in the range $5 \leq y \leq 10$
$k = .9$

Assume x, y, k are stored in the following locations:

x (-2) in 1501
y (-2) in 1502
k (0) in 1503

Use the notation for scaling explained on pages 44-46.

V. Testing

1. INTRODUCTION

Testing a program for errors in coding is an essential phase in the solution of a problem. The results that we want are not necessarily what we instruct the machine to give us. We must make sure that the instructions as written will produce intended results insofar as coding errors, such as forming $(x \cdot y)$ instead of $(x \cdot z)$, are concerned. The objective of this chapter is to present methods for detecting such errors in a program.

The detection of coding errors can sometimes be laborious and costly. In order to eliminate as many mistakes as possible before testing a program, one should write carefully all flow charts and programs and check them meticulously with particular emphasis on scaling mistakes.

Errors, such as rounding, truncation, etc., that are inherent in a computational method will also affect results, but we assume that a complete error analysis has been made to determine these errors.

A distinction is made between testing and checking. Checking refers to the proper functioning of the machine, which is the responsibility of the operator.

2. TRACING

Tracing a program for mistakes is the simple and straightforward process of writing the result of the execution of each instruction adjacent to the instruction itself; by studying all of the details of the trace, we can detect any mistakes that appear in the program. Rather than calculate the results by hand, we instruct the machine to perform the tracing automatically.

Automatic tracing techniques require that one card be punched for every instruction executed. Indiscriminate use of automatic tracing results in the punching of many cards which is costly in terms of the time required for punching as

well as for studying a great deal of information. In order to reduce the number of instructions that we trace, we shall follow two rules for automatic tracing:

1. Begin the tracing at the logical core of the program, which is the right-hand level, and test it completely before testing the next level to the left.

2. Shrink the program, i.e. do not trace through a loop many times when a few times will suffice. For example, in the problem in Chapter III that involved the computation of 100 products, we would trace the formation of only nine products for first, last, and middle values of x and y. If the program operates for three values, we can assume that it will operate for ten values. We shrink the program by changing the terminating constants on page 29 as follows:

For 100 products				For 9 products			
0600	19	1310	0502	0600	19	1303	0502
0603	60	1210	0501	0603	60	1203	0501

You must make sure that there is not a mistake in either of the two terminating constants.

3. AUTOMATIC TRACERS

One tracer in the System is an auxiliary subroutine that causes the machine to execute instructions of a program, one at a time, and to record the contents of the entire accumulator. The tracing can begin at any instruction in a program and, after each instruction is executed, a card will be punched out with the following information:

xxxx	xxxx	xx xxxx xxxx	xx xxxx xxxx	xx xxxx xxxx	xx xxxx xxxx
No.	L (Inst.)	Instruction	Upper	Lower	Distributor

Column 1 = Consecutive number of instruction
Column 2 = Location of instruction
Columns 4-6 = Contents of upper accumulator, lower accumulator, and distributor before execution of instruction

The distributor is a register intermediate between the accumulator and memory. All information passing from memory to the accumulator and from the accumulator to memory passes through the distributor. The information in the cards that are punched out by the tracing subroutine can be printed on paper and examined away from the machine.

As an example of the use of the tracer, let us consider the program for the problem on page 13. The operator feeds

into the machine the program on page 14, the data card, and the tracer. If the numbers in the data card are the following:

A (-1) = .01 2345 6789 $\quad$ C (0) = .36 9121 5182
B (0) = .13 5790 2468 $\quad$ D (-1) = .07 0710 6781,

the sample calculations should be:

(A/B) (-1) = .09 0917 2727 $\quad$ $(C \sin D)$ (-1) = .02 3979 4972
$(\sin D)$ (-1) = .06 4963 6937 $\quad$ E (-1) = .11 4896 7699

and the information in the cards that are punched out by the tracer is printed as follows:

No.	L (Inst.)	Instruction	Upper	Lower	Distributor
1	0500	60 0600 0501	xx xxxx xxxx	xx xxxx xxxx	xx xxxx xxxx
2	0501	69 0502 0062	04 1200 4001	00 0000 0000	04 1200 4001
65	0204	60 1200 0503	04 1200 4001	00 0000 0000	04 1200 4001
66	0503	64 1201 0504	01 2345 6789	00 0000 0000	01 2345 6789
67	0504	20 0700 0505	00 0000 0000	09 0917 2727	13 5790 2468
68	0505	60 1203 0506	00 0000 0000	09 0917 2727	09 0917 2727
69	0506	69 0507 0071	07 0710 6781	00 0000 0000	07 0710 6781
113	0301	19 1202 0508	06 4963 6937	00 0000 0000	06 4963 6937
114	0508	10 0700 0509	02 3979 4972	46 4237 7534	36 9121 5182
115	0509	21 1204 0510	11 4896 7699	46 4237 7534	09 0917 2727
116	0510	60 0601 0511	11 4896 7699	46 4237 7534	11 4896 7699
117	0511	69 0512 0061	05 1200 5002	00 0000 0000	05 1200 5002
1 2345	6789	13 5790 2468	36 9121 5182	07 0710 6781	11 4896 7699
184	1759	01 0000 0000	05 1200 5002	00 0000 0000	05 1200 5002

Each line listed above contains the information that is punched in one tracer-output card. The "L (Inst.)" and "Instruction" columns are identical with the corresponding columns on page 14 with the following exceptions: The locations that immediately follow the SPOP instructions are fictitious; and the next to the last line contains the data that were punched out by the memory-to-card subroutine, i.e. the values of A, B, C, D, and E, properly scaled.

Examination of the consecutive numbers in column 1 shows that the first SPOP (read in) instruction involved 63 instructions, the second one involved 44 instructions, and the last one involved 67 instructions.

The last three columns of the tracer show the contents of the accumulator and distributor. The data associated with an operation are printed on the line following the instruction. The x's on the first line indicate that this information does not, in general, relate to the problem since these data were present before the first instruction was executed.

A comparison of the numbers in the subsequent lines with the values of A, B, C, etc. in our example facilitates the identification of each quantity. In the following table we have listed the quantities as they appear in the tracer and opposite them we have listed from page 14 the operation being performed. This comparison shows not only that each operation uses the correct factor, but also that the scaling is correct.

Read in A, B, C, D (α = 04 1200 4001)	α	0	α
	α	0	α
Compute A/B; store in 0700	A	0	A
	0	A/B	B
	0	A/B	A/B
Compute sin D	D	0	D
	sin D	0	sin D
Compute $A/B + C \sin D$; store in 1204 (Note: $C \sin D$ extends into lower)	$C \sin D$	–	C
	E	–	A/B
	E	–	E
Punch A, B, C, D (α = 05 1200 5002)	α	0	α
	C	D	E
	α	0	α

By studying this printed record of the tracer, we can investigate all of the arithmetical details of the program away from the machine. We call this type of tracing subroutine an arithmetical tracer since it is used primarily to trace arithmetical operations.

When we suspect that mistakes are caused by faulty terminating procedures, we use another type of tracing subroutine, called the logical tracer, which traces only branch instructions. In order to use the logical tracer effectively, we must examine the standard terminating procedure. In the example of a three-level basic form (Section 6 of Chapter III), there are three instructions corresponding to the terminate box in the right-hand level with termination on y_{10}:

0503	RAU	60	0501	0504	
0504	SU	11	0600	0505	
0505	BRNZU	44	0506	0512	
0501	MPY	[19	1301	0502]	Variable instruction
0600		19	1310	0502	Constant

The machine determines whether or not the "variable in-

struction minus the constant" is non-zero. If the branch instruction is traced, the following information will be punched out:

```
1  0505  44 0506 0512  00 0009 0000–  00 0000 0000  19 1310 0502
```

The contents of the upper accumulator (00 0009 0000–) indicates that one product has been formed, and the constant (19 1310 0502) appears in the distributor.

Since it is the data address in the variable instruction that we frequently need to trace, the program for the terminate box is changed to compute "minus constant plus variable instruction" followed by the BRNZU instruction:

```
0503 | RSU     61  0600  0504
0504 | AU      10  0501  0505
0505 | BRNZU   44  0506  0512
0501 | MPY    [19  1301  0502]   Variable instruction
0600 |         19  1310  0502    Constant
```

The results of these instructions for the terminate box will be exactly the same as before, but the logical tracer will display the variable instruction instead of the constant.

```
1  0505  44 0506 0512  00 0009 0000–  00 0000 0000  19 1301 0502
```

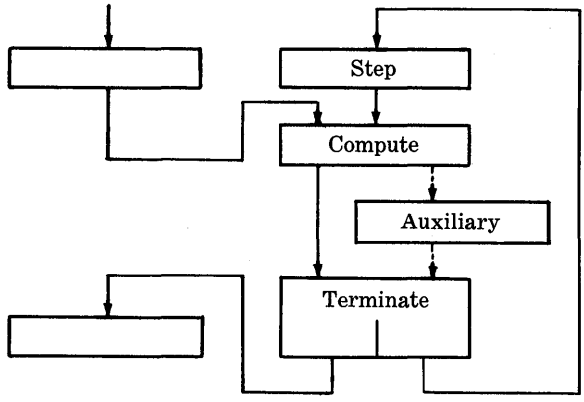
Let us combine two of the testing methods discussed above in order to shorten the testing phase of the products problem. First, we shrink the program from 100 to 9 products by changing the terminating constants as indicated in Section 2 and, secondly, using the new terminating procedure, we form the following logical trace of the entire products problem:

```
 1  0505  44 0506 0512  00 0002 0000–  00 0000 0000  19 1301 0502
 2  0505  44 0506 0512  00 0001 0000–  00 0000 0000  19 1302 0502
 3  0505  44 0506 0512  00 0000 0000   00 0000 0000  19 1303 0502

 4  0516  44 0517 0531  00 0002 0000–  00 0000 0000  60 1201 0501

 5  0505  44 0506 0512  00 0002 0000–  00 0000 0000  19 1301 0502
 6  0505  44 0506 0512  00 0001 0000–  00 0000 0000  19 1302 0502
 7  0505  44 0506 0512  00 0000 0000   00 0000 0000  19 1303 0502

 8  0516  44 0517 0531  00 0001 0000–  00 0000 0000  60 1202 0501

 9  0505  44 0506 0512  00 0002 0000–  00 0000 0000  19 1301 0502
10  0505  44 0506 0512  00 0001 0000–  00 0000 0000  19 1302 0502
11  0505  44 0506 0512  00 0000 0000   00 0000 0000  19 1303 0502

12  0516  44 0517 0531  00 0000 0000   00 0000 0000  60 1203 0501
```

4. AUXILIARY PUNCH-OUT ROUTINE

During the testing phase in the solution of a problem, we try to anticipate the information that we might wish to examine if mistakes should occur. We can say, in general, that it is always useful to be able to examine intermediate results and the contents of the various work cells that are used. Other useful information will depend on the nature of the particular program. In the technique described in this section we obtain a picture of memory cells at a particular point in the program by means of the memory-to-cards special operation (SPOP code 0061).

After deciding which locations are to be examined, the coder writes the instructions that call in the special operation 0061. An operator can insert these instructions into the main program by using the manual controls on the console. If the program were running on the machine, it must be stopped and the instruction address of one of its instructions must be changed so that the main program will include the auxiliary one, e.g.



In general, a coder can use the memory-to-cards special operation any number of times during the testing phase of a program since there can be any number of parts in a program that will yield significant information.

5. CONSOLE ERROR DETECTION

The facilities of the machine include the automatic stopping of the machine under certain circumstances accompanied by the appearance of corresponding indicator lights on the console. The reasons for the automatic stopping of

the machine can be divided into two categories: machine errors and coding errors.

It is possible that the machine will make an occasional random error, which it detects by means of a "validity" check; if the check does not hold, the machine will stop and an indicator light will appear on the operator's console. The validity check is based on the manner in which the individual digits are indicated in the machine. Each digit is broken into two parts, the first of which is 0 or 5 and the second is 0, 1, 2, 3, or 4. Thus,

0 = 0 and 0	5 = 5 and 0
1 = 0 " 1	6 = 5 " 1
2 = 0 " 2	7 = 5 " 2
3 = 0 " 3	8 = 5 " 3
4 = 0 " 4	9 = 5 " 4

In this so-called bi-quinary system of indication, the 0- or 5-indicator is the binary part and the 0-, 1-, 2-, 3-, or 4-indicator is the quinary part. Whenever a number is transferred into the control unit, the accumulators, or the distributor, the machine checks each digit to verify that it has one and only one binary indication and one and only one quinary indication. If an indication is lost or gained, the machine will stop and lights will indicate the register in which the error occurred. The chance that two errors, losing one indication and gaining another, will compensate each other and thereby remain undetected is extremely small. As far as we know, the machine at the Watson Laboratory has never made an undetected error.

Various types of coding and punching errors will cause the machine to stop automatically:

1. Control unit: an invalid address, such as one greater than 1999, in an instruction, or an operation code that is not meaningful to the machine.
2. Overflow: numbers that are not scaled properly. More digits will be developed in the accumulator than it can hold and the "overflow" indicator will light up.
3. Card punching errors: a missing digit (blank column). A column with more than one digit punched in it or a word without a sign will be picked up as a validity check error.

These errors will be detected by the machine. However, there are many types of coding errors that the machine will not recognize as errors, e.g. an incorrect but valid address,

a scaling error that does not result in an overflow but does result in an incorrect answer, an incorrect terminating constant, etc. Careful checking of the program and coding can minimize these errors, but, for those that are not uncovered by checking, a testing procedure must be used. For example, if there should be a machine stop that is not caused by a machine error, it is often advisable to perform an auxiliary memory punch-out at that point and to restart the problem, tracing the segment of the program that precedes the stop. The combined information from the punch-out routine and the trace should give a clear picture of the nature of the error.

A program should be organized in such a way that it will be possible to "back up" if, for example, there is a machine error. The basic form lends itself to a restarting or "backing up" procedure; usually it is possible to back up to the set box in one of the levels to the left of the level in which the error occurred. It is advisable to back up far enough to obtain some overlapping results that will check the restarting procedure, especially when it is necessary to interrupt a program and to restart it the next day.

The auxiliary punch-out routine, which is used for the initial testing of a program, can also be used to punch out intermediate results during the running of a long problem that is not producing expected partial results. If, during programming, the routine is written to punch out intermediate results, it can be inserted to give you the extra information needed to determine whether or not the questionable partial results are correct.

6. MEMORANDA

Testing a program is one of the most difficult phases in the solution of a problem and requires that you thoroughly understand your program. To assist you in the preparations for testing your program, we have listed the information that should be readily available to you:

1. A neat set of final detailed flow charts and instructions.
2. A set of precomputed results, both partial and final.
3. A clear picture of memory assignments.
4. A list of stops that have been included in the program.
5. Points in the program where it can be restarted.
6. Points in the program where a trace or an auxiliary memory punch-out would be useful.

7. Auxiliary punch-out routines that have been written to punch out intermediate results, variable instructions, or contents of work cells, and where to insert them in the program. These routines are written when the main program is written, punched into cards, and entered into the machine when the main program is entered into the machine.

VI. Conclusion

This pamphlet is intended to give to the reader a basic understanding of the operations involved in the solution of computational problems on an automatic calculator; it should also enable him to solve problems of moderate size and complexity on the 650 and to cooperate with professional computing groups in solving large problems on any calculator.

In order to solve large and intricate problems with the necessary efficiency, the reader will require additional knowledge about the machine, coding, programming, and probably about numerical analysis. At this stage he will have no difficulty in obtaining more information about the 650 from the operator's "Manual of Operation". Three features of the machine will contribute to the efficiency of its operation:

First, we have mentioned only 13 basic machine instructions whereas there are actually 44 instructions on the standard machine. The use of these additional instructions will facilitate the machine work for many problems.

A second feature of the machine is a pluggable control panel that controls the reading and punching of the cards. In the System we have used a single, general purpose panel and a standardized arrangement of the data in a card. However, the use of a panel that is wired especially for a problem with considerable input and output may add greatly to the machine's efficiency.

Finally, the memory of the 650 is on the surface of a rotating magnetic drum, and each cell of the memory is available for access once each revolution of the drum. In general, there is some waiting for a particular cell to become available to the machine, but this waiting time can be reduced if the words are placed on the drum according to the sequence of instructions to be performed. The process of placing words on the drum where they will be ready for access when called for is known as "optimizing" the program. To produce the "optimum" program for a given problem is intricate and laborious, but a good approximation can be obtained with little effort. A program can be optimized by the coder with

the aid of the timing charts in the "Manual" or it may be done automatically by the machine as described in a following paragraph. As would be expected, the special operations used in the System have been optimized carefully.

In optimizing a program by hand, the coder would devote most of his attention to the coding of the loops in the rightmost levels of the flow chart since these are the ones that are repeated most frequently.

We have seen that, once the flow charts have been constructed, the coding, though tedious, is fairly straightforward. Machine methods for coding have been devised, and the most comprehensive one for the 650 is known as "SOAP"*. This system will convert rather general instructions into an optimized program for the machine; it is widely used and includes a large assortment of library programs.

Some models of the 650 are equipped with magnetic tapes, large auxiliary storage devices, and printers. This additional equipment offers the necessary capacity for large problems.

A comprehensive treatment of programming as used in the System will be contained in a forthcoming book by J. Jeenel. As an example of increased efficiency through improved programming we might mention the relation between operating time and storage space. It frequently happens that in a particular part of the program we have the choice of saving storage space at the expense of increased operating time or vice versa. In Section 3 of Chapter III we saw that in the summation problem we could save space by using the technique of looping; however, the resulting program involved many more instructions than the one written out in full. The flow chart, which shows the relative frequency of execution of the various levels, serves as a guide in balancing time and space considerations. The general rule is to save time in the levels to the right of the diagram and to save space in the levels on the left.

In conclusion one cannot refrain from commenting on the widespread activity in machine computation. Soon there will be in operation more than a thousand 650's in addition to other machines of the same general scope, and machines of greater speed and capacity than the 650 will soon number in the hundreds. Most universities now give formal instruction in numerical and machine methods. The professional people who are engaged in machine computation are numbered in the thousands.

---

*650 Programming Bulletin No. 1, International Business Machines Corporation, New York, 1956.

Appendix I. Summary of Operations

BASIC OPERATIONS AND THEIR AVERAGE EXECUTION TIMES IN MILLISECONDS

			ms
RAU	60	reset add upper	0.4
RSU	61	reset subtract upper	0.4
AU	10	add upper	0.4
SU	11	subtract upper	0.4
STU	21	store upper	0.4
MPY	19	multiply	10.
DIVR	64	divide	15.
STL	20	store lower	0.4
SHRT	30	shift right	2.5
SHLT	35	shift left	2.5
BRNZU	44	branch on non-zero in upper	0.4
BR–	46	branch on minus	0.4
STOP	01	stop	0.4

SPECIAL OPERATIONS, THEIR MEMORY ASSIGNMENTS, AND THEIR AVERAGE EXECUTION TIMES IN MILLISECONDS

1. ROUNDING
   005x shift right and round
   (0050-0059, 0266-0269) 20 ms
2. BLOCK TRANSFERS
   0060 memory to memory
   (0060, 0398-0440) (40 + 10n*) ms
   0061 memory to cards
   (0061, 0100-0159) 600 ms/card
   0062 cards to memory
   (0062, 0200-0265) 300 ms/card
   *n = number of words to be transferred
3. FUNCTIONS

0070	$\sqrt{\alpha}$	(0070, 0160-0199)	125 ms
0071	$\sin \alpha$	(0071, 0298-0397)	124 ms
0072	$\cos \alpha$	(0072, 0298-0397)	124 ms
0073	e^{α}	(0073, 0441-0499)	280 ms
0074	$\log_e \alpha$	(0074, 1744-1811)	190 ms
0075	$\arctan \alpha$	(0075, 1700-1742)	130 ms
0076	$\arcsin \alpha$	(0076, 1812-1880)	150 ms

We recall that each number in the machine is in the form, $\alpha\,(n) = .xxxxxxxxxx$, where n is the scale factor on the coding sheet. Before using an argument, α, in a subroutine for the computation of a function, we must adjust n to the standard value listed in column 4 of the following function table.

SPOP Code	Function	Range of α	Standard n for α	Maximum error in $f(\alpha)$
0070	$\sqrt{\alpha}$		even*	4×10^{-10}
0071	$\sin \alpha$	$-2\pi \leq \alpha \leq 2\pi$	–1	3×10^{-9}
0072	$\cos \alpha$	$-3.2\pi \leq \alpha \leq 2.5\pi$	–1	3×10^{-9}
0073	e^{α}	$-1 < \alpha < 1$	0	8×10^{-9}
0074	$\log_e \alpha$	$1 \leq \alpha < 10$	–1	4×10^{-9}
0075	$\arctan \alpha$	$-1 \leq \alpha \leq 1$	–1	4×10^{-8}
0076	$\arcsin \alpha$	$0 \leq \alpha \leq 1$	–1	2×10^{-8}

* If the scale factor, n, is odd, shift the argument one place to the right, or multiply the square root of α by the square root of 10.

The computed functions will all have a scale factor of –1, $(n = -1)$, except for the square root of α.

The arctangent of α for α in the range, $0 \leq \alpha \leq \infty$, can be computed from the following expression:

$$\arctan \alpha = \pi/4 + \arctan \left(\frac{\alpha - 1}{\alpha + 1}\right)$$

The subroutines for the seven functions were written by G. R. Trimble, Jr., and they are contained in the IBM Technical Newsletter No. 9.

Appendix II. Floating Point and Double Precision

FLOATING-POINT OPERATIONS

The following operation codes can be used in floating-point as well as in fixed-point operations:

RAU	60	reset add upper
RSU	61	reset subtract upper
AU	10	add upper
SU	11	subtract upper
STU	21	store upper
STL	20	store lower
MPL	19	multiply
DIVR	64	divide
BRNZU	44	branch on non-zero in the upper
BR–	46	branch on minus

All of the coding rules that apply to these operations in fixed point are applicable in floating point.

Since the operation codes are the same for both fixed- and floating-point operations, we use SPOP instructions to tell the machine to start floating-point and to return to fixed-point operation. The special operation code 0081 means, "Start floating-point operation"; special operation code 0080 means, "Return to fixed-point operation". For example, if the instructions in 0500-0525 had been executed in fixed point and the next calculation in the program required the floating-point mode of operation, we would write the following SPOP instruction to cause the series of instructions beginning in 0527 to be executed as floating-point instructions:

```
0526 | SPOP   69   0527   0081
```

In order to return to the fixed-point mode for the instructions beginning in 0641, we would write another SPOP instruction:

```
0640 | SPOP   69   0641   0080
```

The first instruction after entering or leaving the floating-point mode must be a reset instruction, i.e. RAU 60 or RSU 61.

All input, output, and red-tape (setting, stepping, and,

in general, terminating) instructions must be executed in fixed-point operation. Since rounding is automatic in floating-point operations, the special operation for rounding is not used in the floating-point mode. Block transfers can be executed only in the fixed-point mode. Provisions have been made for including in the floating-point mode the subroutines that compute functions.

Floating-point operations can be traced; their trace sheet contains the same information as the trace sheet for fixed-point operation.

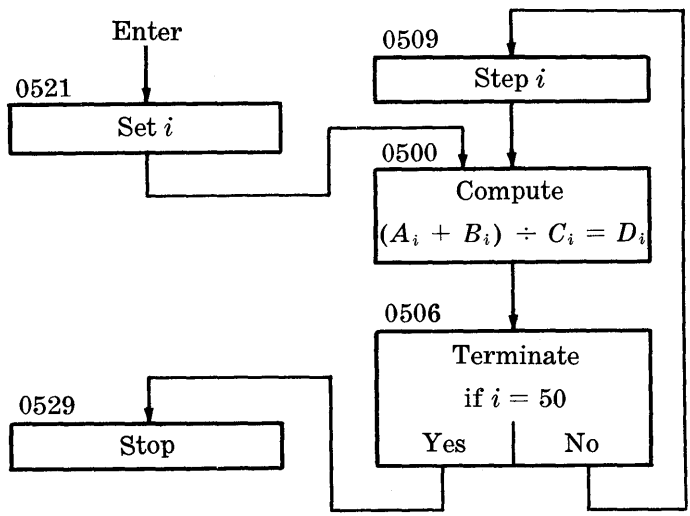
The average execution time for a floating-point operation is 3½ times the duration of the corresponding optimized fixed-point operation.

Since the adjusted exponent of a floating-point number (see p. 43) occupies only two digit positions in memory, an exponent greater than 99 will cause the machine to stop. A number with an adjusted exponent less than 01 will be represented automatically as: 00 0000 0000.

As an example of a floating-point calculation, let us compute in floating point: $(A_i + B_i) \div C_i = D_i$ for 50 values of A, B, and C, with

A_1 in 1301	B_1 in 1401	C_1 in 1501	D_1 in 1601
A_2 in 1302	B_2 in 1402	C_2 in 1502	D_2 in 1602
. . .	. . .	. . .	. . .
A_{50} in 1350	B_{50} in 1450	C_{50} in 1550	D_{50} in 1650

The flow chart follows:



The program is coded as follows:

0500	SPOP	69	0501	0081	Start fl. pt.
0501	RAU	[60	1301	0502]	
0502	AU	[10	1401	0503]	Compute
0503	DIVR	[64	1501	0504]	$(A + B) \div C = D$
0504	STL	[20	1601	0505]	
0505	SPOP	69	0506	0080	Return to fixed pt.
0506	RAU	60	0501	0507	
0507	SU	11	0600	0508	Terminate
0508	BRNZU	44	0509	0529	
0509	RAU	60	0501	0510	
0510	AU	10	0601	0511	
0511	STU	21	0501	0512	
0512	RAU	60	0502	0513	
0513	AU	10	0601	0514	
0514	STU	21	0502	0515	
0515	RAU	60	0503	0516	Step i
0516	AU	10	0601	0517	
0517	STU	21	0503	0518	
0518	RAU	60	0504	0519	
0519	AU	10	0601	0520	
0520	STU	21	0504	0500	
0521	RAU	60	0602	0522	
0522	STU	21	0501	0523	
0523	RAU	60	0603	0524	
0524	STU	21	0502	0525	Set i
0525	RAU	60	0604	0526	
0526	STU	21	0503	0527	
0527	RAU	60	0605	0528	
0528	STU	21	0504	0500	
0529	STOP	01	0000	0000	
0600		60	1350	0502	
0601		00	0001	0000	
0602		60	1301	0502	
0603		10	1401	0503	
0604		64	1501	0504	
0605		20	1601	0505	

CONVERSION SUBROUTINES

Special operation 0063 converts a block of fixed-point data to floating-point data; special operation 0064 converts a block of floating-point data to fixed-point data. The code word, α, is constructed as follows:

$$\alpha = \underset{N}{\text{xx}}\ \underset{a}{\text{xxxx}}\ \underset{b}{\text{xxxx}}$$

where N = total number of consecutive words to be converted

a = location of first word in the block to be converted

b (with 0063) = 50 minus the scale factor for the block

b (with 0064) = the largest adjusted exponent in the block; the machine will stop if an adjusted exponent is greater than b.

As an example of the use of a conversion subroutine, let us convert from fixed-point to floating-point form the following block of 36 words, x_i, stored in locations 0801-0836,

.4891627831
.0167832146
.0020743098
.0003700329
.0000143650
. . . .

with the scale factor, x ③ , on the coding sheet. The code word, α, is constructed as follows:

$$\alpha = 36\ 0801\ 0053$$

To make the conversion, we write the following instructions:

0600	RAU	60	L(α)	0601
0601	SPOP	69	0602	0063

When the subroutine is completed, the block of 36 words beginning in location 0801 will appear in memory as follows:

.4891627853
.1678321452
.2074309851
.3700329050
.1436500049
. . . .

If we wished to convert this floating-point block of data

to fixed-point form, we would use the same α as the one above, and we would write the same instructions as those above except that the SPOP code would be 0064 instead of 0063. The converted data would duplicate the original fixed-point array, except that, at most, only eight digits of each word would be retained.

DOUBLE-PRECISION SUBROUTINES

All of the basic operations that are used in floating-point can also be used in the double-precision mode. The special operation code 0082 means, "Start double-precision operation", and special operation code 0080 again means, "Return to fixed-point operation".

Since 20-digit numbers are used in double-precision arithmetic, two cells are required for each number. If we write the following instruction in double-precision operation,

```
0534| RAU  60  0843  0535
```

the high-order part of the number is in location 0843, and the low-order part is in 0844.

Appendix III. Additional Programming Techniques

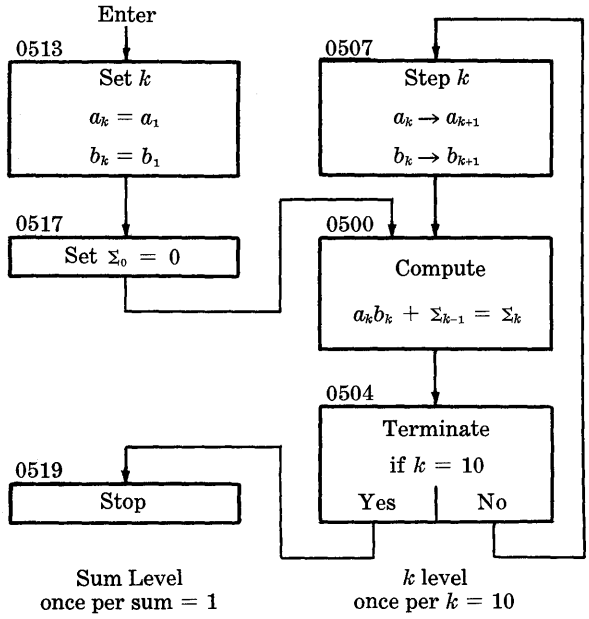
SUMS OF PRODUCTS

Let us consider the following program for computing sums of products, i.e. $\Sigma_{k=1}^{n} a_k b_k$.

Example 1. $S = \Sigma_{k=1}^{10} a_k b_k$

a_1 in 1200, a_2 in 1201, $\cdots$, a_{10} in 1209;

b_1 in 1300, b_2 in 1301, $\cdots$, b_{10} in 1309; S in 1400.



0500	RAU	[60	1200	0501]	Compute $(a_k b_k + \Sigma_{k-1}) = \Sigma_k$
0501	MPY	[19	1300	0502]	
0502	AU	10	1400	0503	
0503	STU	21	1400	0504	
0504	RAU	60	0500	0505	Terminate if $k = 10$
0505	SU	11	0600	0506	
0506	BRNZU	44	0507	0519	

0507	RAU	60	0500	0508	
0508	AU	10	0601	0509	Step a_k
0509	STU	21	0500	0510	$a_k \rightarrow b_{k+1}$
0510	RAU	60	0501	0511	
0511	AU	10	0601	0512	Step b_k
0512	STU	21	0501	0500	$b_k \rightarrow b_{k+1}$
0513	RAU	60	0602	0514	Set
0514	STU	21	0500	0515	$a_k = a_1$
0515	RAU	60	0603	0516	Set
0516	STU	21	0501	0517	$b_k = b_1$
0517	RAU	60	0604	0518	Set
0518	STU	21	1400	0500	$\Sigma_0 = 0$
0519	STOP	01	0000	0000	
0600		60	1209	0501	
0601		00	0001	0000	
0602		60	1200	0501	
0603		19	1300	0502	
0604		00	0000	0000	

This problem is similar in form to the simple accumulation problem in Chapter III. Both programs are two-level basic forms, and the "red-tape" boxes (terminating, stepping, and setting boxes) are the same except that in this problem there are two variable instructions to be set and stepped instead of one. In the compute box there are now a multiplication and an addition.

This example can also be thought of as a "vector times a vector" multiplication. The rightmost or k level is the term level, i.e. we go through it once for each term of the resulting element. The sum level is the element level, which we go through once for a "vector by vector" multiplication.

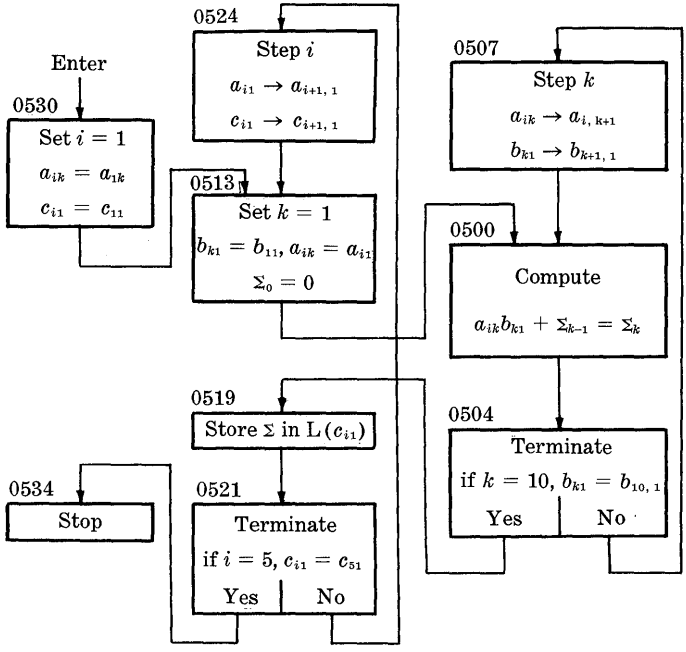
"MATRIX TIMES VECTOR" MULTIPLICATION

The above program can be expanded to perform a "matrix times a vector" multiplication by the addition of another level to the left. The added level represents another dimension that has been added to the "vector by vector" multiplication problem. The program and coding for a "matrix times vector" multiplication with the matrix stored by columns are given below.

Example 2. "Matrix Times Vector" multiplication with 5×10 matrix stored by columns and a column vector of 10 elements.

$$\begin{bmatrix} a_{11}\ a_{12} \cdots a_{1,\,10} \\ a_{21}\ a_{22} \cdots a_{2,\,10} \\ \cdot \quad \cdot \quad \cdot \\ a_{51}\ a_{52} \cdots a_{5,\,10} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \cdot \\ \cdot \\ \cdot \\ b_{10,\,1} \end{bmatrix} = \begin{bmatrix} c_{11} \\ c_{21} \\ \cdot \\ \cdot \\ \cdot \\ c_{5,\,10} \end{bmatrix}$$

a_{11} in 1200 b_{11} in 1300 c_{11} in 1400 S in 1299
a_{21} in 1201 b_{21} in 1301 c_{21} in 1401
. . . . . .
a_{12} in 1205 $b_{10,\,1}$ in 1309 c_{51} in 1404
. . .
$a_{5,\,10}$ in 1249



Vector level or j level | Element level or i level | Term level or k level

0500	RAU	[60	1200	0501]	Compute $(a_{ik}b_{k1} + \Sigma_{k-1}) = \Sigma_k$
0501	MPY	[19	1300	0502]	
0502	AU	10	1299	0503	
0503	STU	21	1299	0504	
0504	RAU	60	0501	0505	Terminate if $b_{k1} = b_{10,\,1}$
0505	SU	11	0600	0506	
0506	BRNZU	44	0507	0519	

0507	RAU	60	0500	0508	Step a_{ik}
0508	AU	10	0609	0509	$a_{ik} = a_{i,\,k+1}$
0509	STU	21	0500	0510	
0510	RAU	60	0501	0511	Step b_{k1}
0511	AU	10	0601	0512	$b_{k1} = b_{k+1,\,1}$
0512	STU	21	0501	0500	
0513	RAU	60	0602	0514	Set
0514	STU	21	0501	0515	$b_{k1} = b_{11}$
0515	RAU	60	0603	0516	Set
0516	STU	21	0500	0517	$a_{ik} = a_{i1}$
0517	RAU	60	0604	0518	Set
0518	STU	21	1299	0500	$\Sigma_0 = 0$
0519	RAU	60	1299	0520	Store c
0520	STU	[21	1400	0521]	
0521	RAU	60	0520	0522	Terminate
0522	SU	11	0605	0523	if $c_{i1} = c_{51}$
0523	BRNZU	44	0524	0534	
0524	RAU	60	0603	0525	Step a_{i1}
0525	AU	10	0601	0526	$a_{i1} \rightarrow a_{i+1,\,1}$
0526	STU	21	0603	0527	
0527	RAU	60	0520	0528	Step c_{i1}
0528	AU	10	0601	0529	$c_{i1} \rightarrow c_{i+1,\,1}$
0529	STU	21	0520	0513	
0530	RAU	60	0607	0531	Set
0531	STU	21	0603	0532	$a_{ik} = a_{1k}$
0532	RAU	60	0608	0533	Set
0533	STU	21	0520	0513	$c_{i1} = c_{11}$
0534	STOP	01	0000	0000	

0600	19	1309	0502	Terminating constant, k level
0601	00	0001	0000	Stepping constant
0602	19	1300	0502	Setting constant, b_{k1}
0603	[60	1200	0501]	Variable setting constant, a_{ik}
0604	00	0000	0000	Zero setting constant
0605	21	1404	0521	Terminating constant, i level
0607	60	1200	0501	Setting constant, a_{ik}
0608	21	1400	0521	Setting constant, c_{i1}
0609	00	0005	0000	Stepping constant

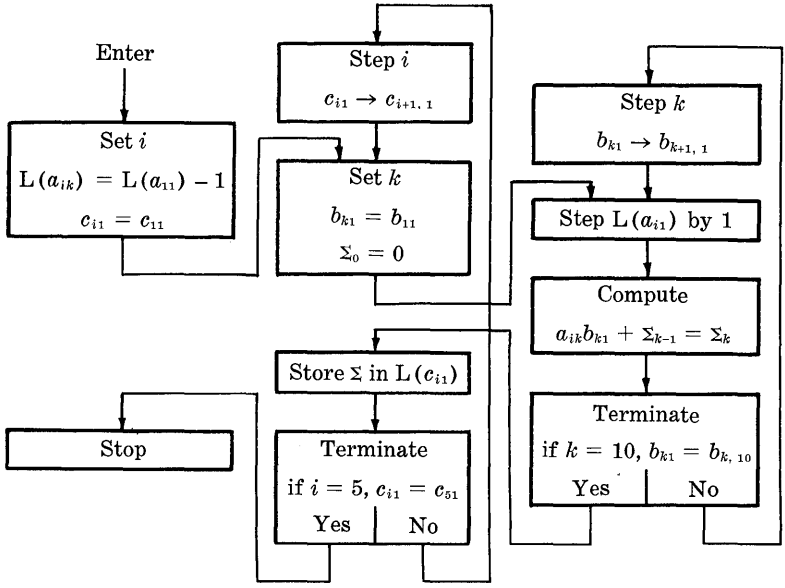
The rightmost or term level can also be called the k level; on

this level the loop is terminated on k and k is stepped. To the left of the k level is the element or i level where we set or initialize those instructions that vary with k; on this level we also terminate on and step i.

You will note that the stepping constant in 0603 is itself variable. The set box on the i level always starts a column with $k = 1$, but i must also be stepped for each row. Instructions 0524-0526 step the setting constant from the first element of one row to the first element of the next row. Here, the stepping constant must be 1 because of the order in which the elements are stored. The frequencies of the levels are as follows:

Term level	Once per term or once per k	= 50
Element level	Once per result element or once per i =	5
Vector level	Once per result vector or once per j =	1

The order in which the variables are stored often affects the program. For example, if the elements of the matrix were stored by rows instead of columns, the flow chart would appear as follows:



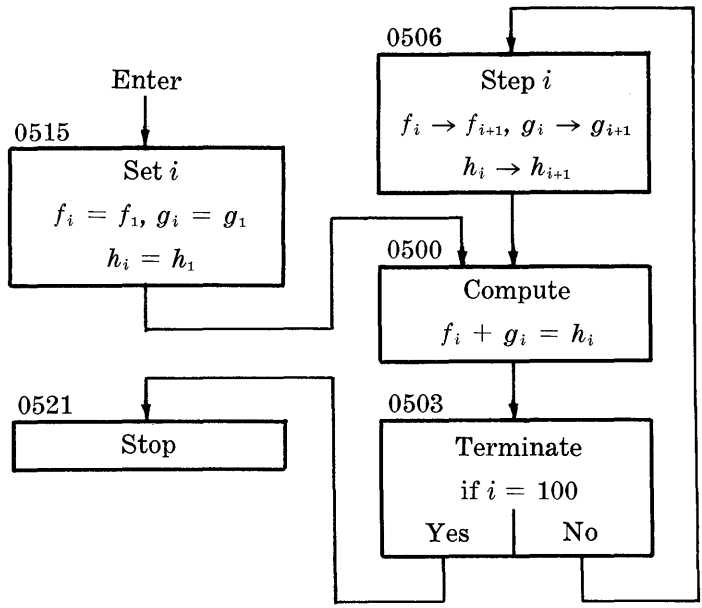
When the matrix elements are stored by rows, it is not necessary to set or step a_{ik} on the i level. However, since a_{ik} is always stepped on the k level, we must allow for this stepping by undersetting a_{ik} in the leftmost level.

MATRIX ADDITION

There are cases where the order in which the data are stored is of importance. For example, let us consider the following problem: $f_{ij} + g_{ij} = h_{ij}$, where i and $j = 1, 2, \cdots, 10$. Both f and g are two-dimensional arrays of numbers, and we must add an element of f to the corresponding element of g. If both sets of numbers are stored in the same order, the problem reduces to: $f_i + g_i = h_i$ where $i = 1, 2, 3, \cdots, 100$. This is a one-dimensional problem and can be programmed and coded in the following way.

Example 3. Sum of Planes
$f_{ij} + g_{ij} = h_{ij}$ with i and $j = 1, 2, \cdots, 10$, where f and g are stored in the same order.

f_{11} in 1200	g_{11} in 1300	h_{11} in 1400
f_{12} in 1201	g_{12} in 1301	h_{12} in 1401
. . .	. . .	. . .
$f_{10,10}$ in 1299	$g_{10,10}$ in 1399	$h_{10,10}$ in 1499



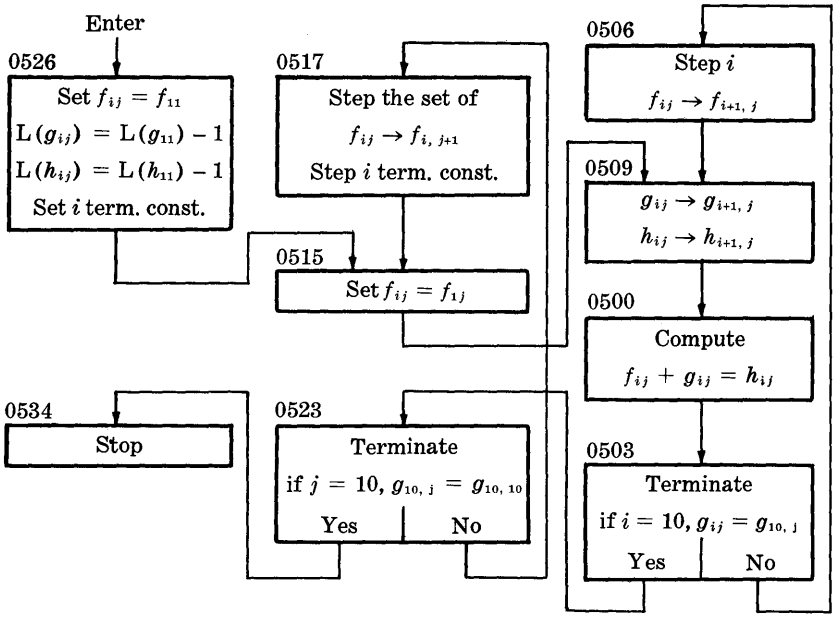
0500	RAU	[60	1200	0501]	Compute
0501	AU	[10	1300	0502]	$f_i + g_i = h_i$
0502	STU	[21	1400	0503]	

0503	RAU	60	0500	0504	Terminate
0504	SU	11	0600	0505	if $i = 100$
0505	BRNZU	44	0506	0521	
0506	RAU	60	0500	0507	
0507	AU	10	0601	0508	Step f_i
0508	STU	21	0500	0509	$f_i \rightarrow f_{i+1}$
0509	RAU	60	0501	0510	
0510	AU	10	0601	0511	Step g_i
0511	STU	21	0501	0512	$g_i \rightarrow g_{i+1}$
0512	RAU	60	0502	0513	
0513	AU	10	0601	0514	Step h_i
0514	STU	21	0502	0500	$h_i \rightarrow h_{i+1}$
0515	RAU	60	0602	0516	Set
0516	STU	21	0500	0517	$f_i = f_1$
0517	RAU	60	0603	0518	Set
0518	STU	21	0501	0519	$g_i = g_1$
0519	RAU	60	0604	0520	Set
0520	STU	21	0502	0500	$h_i = h_1$
0521	STOP	01	0000	0000	
0600		60	1299	0501	
0601		00	0001	0000	
0602		60	1200	0501	
0603		10	1300	0502	
0604		21	1400	0503	

The problem becomes more complex if the two sets of numbers are stored differently. If f is stored by rows and g is stored by columns, we have a two-dimensional problem and another level must be added to the flow chart.

Example 4. Sum of Planes
$f_{ij} + g_{ij} = h_{ij}$, where f and g are stored in different orders.

f_{11} in 1200	g_{11} in 1300	h_{11} in 1400
f_{12} in 1201	g_{21} in 1301	h_{21} in 1401
. . .	. . .	. . .
$f_{1,10}$ in 1209	$g_{10,1}$ in 1309	$h_{10,1}$ in 1409
f_{21} in 1210	g_{12} in 1310	h_{12} in 1410
. . .	. . .	. . .



0500	RAU	[60	1200	0501]	Compute
0501	AU	[10	1300	0502]	$f_{ij} + g_{ij} = h_{ij}$
0502	STU	[21	1400	0503]	
0503	RAU	60	0501	0504	Terminate if
0504	AU	11	0600	0505	$i = 10$
0505	BRNZU	44	0506	0523	$g_{ij} = g_{10,\, j}$
0506	RAU	60	0500	0507	
0507	AU	10	0601	0508	Step f_{ij}
0508	STU	21	0500	0509	$f_{ij} \rightarrow f_{i+1,\, j}$
0509	RAU	60	0501	0510	
0510	AU	10	0602	0511	Step g_{ij}
0511	STU	21	0501	0512	$g_{ij} \rightarrow g_{i+1,\, j}$
0512	RAU	60	0502	0513	
0513	AU	10	0602	0514	Step h_{ij}
0514	STU	21	0502	0500	$h_{ij} \rightarrow h_{i+1,\, j}$
0515	RAU	60	0603	0516	Set
0516	STU	21	0500	0509	$f_{ij} = f_{1j}$
0517	RAU	60	0603	0518	
0518	AU	10	0602	0519	Step (set)
0519	STU	21	0603	0520	$f_{1j} \rightarrow f_{1,\, j+1}$

0520	RAU	60	0600	0521	Step
0521	AU	10	0601	0522	term. const.
0522	STU	21	0600	0515	on i level
0523	RAU	60	0501	0524	
0524	SU	11	0604	0525	Terminate
0525	BRNZU	44	0517	0534	if $g_{10,j} = g_{10,10}$
0526	RAU	60	0605	0527	Set
0527	STU	21	0501	0528	$L(g_{ij}) = L(g_{11}) - 1$
0528	RAU	60	0606	0529	Set
0529	STU	21	0502	0530	$L(h_{ij}) = L(h_{11}) - 1$
0530	RAU	60	0607	0531	Set
0531	STU	21	0603	0532	$f_{1j} = f_{11}$
0532	RAU	60	0608	0533	Set i-level
0533	STU	21	0600	0515	term. const.
0534	STOP	01	0000	0000	

0600	[10	1309	0502]	i-level term. const.
0601	00	0010	0000	Step. const. for f_{ij}
0602	00	0001	0000	Step. const. for g_{ij} and h_{ij}
0603	[60	1200	0501]	Set. const. for $f_{ij} = f_{1j}$
0604	10	1399	0502	j-level term. const.
0605	10	1299	0502	Set. const. for g_{ij}
0606	21	1399	0503	Set. const. for h_{ij}
0607	60	1200	0501	Set. const. for $f_{ij} = f_{11}$
0608	10	1309	0502	Set for i-level term. const.

The locations of the g and h elements are continuously stepped by one, while the location of the f elements must be stepped by 10 on the i level and reset to the first element of a column on the j level. Compare this example with the "matrix times vector" multiplication where the matrix is stored by rows.

GROUP STEPPING

The variable instructions in the previous examples have been individually set and stepped. In all of the examples in this chapter the difference between any two of the variable instructions is always a constant. For instance, in Example 3 the difference between the instruction in 0501 and the instruction in 0500 is always (49 9899 9999 –), and the difference between the instruction in 0502 and the instruction in 0501 is always (11 0100 0001). This constant difference permits the second two variable instructions to be "gener-

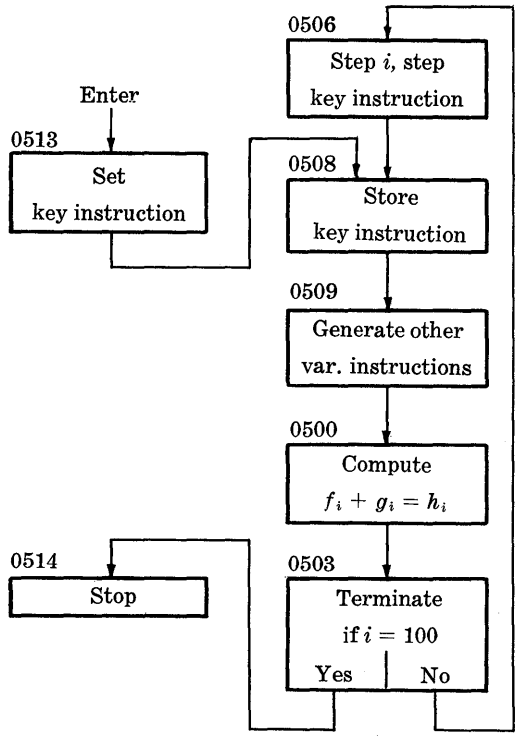
ated" or "constructed" from the first. Generating an instruction, which is equivalent to stepping, is called "group stepping", and the instruction from which the other variable instructions are generated is called the "key instruction". The problem in Example 3 has been reprogrammed to use the group stepping in the following example.

Example 5. Sum of Planes

$f_{ij} + g_{ij} = h_{ij}$ with i and $j = 1, 2, \ldots, 10$

f and g are stored in the same order, and group stepping is used.

f_{11} in 1200	g_{11} in 1300	h_{11} in 1400
f_{12} in 1201	g_{12} in 1301	h_{12} in 1401
. . .	. . .	. . .
$f_{10,\,10}$ in 1299	$g_{10,\,10}$ in 1399	$h_{10,\,10}$ in 1499




```
0500 | RAU  [60  1200  0501]  > Δ1  Compute
0501 | AU   [10  1300  0502]  > Δ2  f_i + g_i = h_i
0502 | STU  [21  1400  0503]
```

0503	RAU	60	0500	0504	Terminate
0504	SU	11	0600	0505	if $i = 100$
0505	BRNZU	44	0506	0514	
0506	RAU	60	0500	0507	Step $L(f_i)$
0507	AU	10	0601	0508	$L(f_i) \rightarrow L(f_{i+1})$
0508	STU	21	0500	0509	Store $L(f_i)$
0509	AU	10	0602	0510	Generate
0510	STU	21	0501	0511	$L(g_i)$
0511	AU	10	0603	0512	Generate
0512	STU	21	0502	0500	$L(h_i)$
0513	RAU	60	0604	0508	Set $f_i = f_1$
0514	STOP	01	0000	0000	
0600		60	1299	0501	
0601		00	0001	0000	
0602		49	9899	9999–	Δ_1
0603		11	0100	0001	Δ_2
0604		60	1200	0501	

Note that by saving two instructions on the i level, group stepping has saved two storage locations; with many variable instructions and a high frequency of execution of the level, the saving in time can be worthwhile.

By using the same "store" instruction (0508) for both setting and stepping the key instruction, we can save another storage location, but no time is saved. Obviously, this trick is not essential in the "group stepping" procedure.

TRIANGULAR ARRAYS

Another interesting problem is one that involves a triangular array of numbers, which can be represented by a three-level basic form. Let us assume that we have a triangular array of numbers in the following locations:

1200	1201	1202	1203	1204	1205
	1206	1207	1208	1209	1210
		1211	1212	1213	1214
			1215	1216	1217
				1218	1219
					1220

Let us operate on these numbers, a row at a time, in the compute box on the rightmost level. For instance, we may

use the instruction, 60 1200 xxxx. If this level is to be terminated at the end of every row, the terminating constant must first read, "60 1205 xxxx", then "60 1210 xxxx", etc. At the end of the first row the terminating constant must be stepped by 5, at the end of the second row by 4, and so on. In order to step the terminating constant, the stepping constant itself must be stepped and set. The following flow chart represents this procedure.