OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP
45 BANBURY ROAD
OXFORD

T 646
28 SEP 1973

IBM

IBM United Kingdom
Laboratories Limited

A First Course on the Schools Computer

D. M. Taub
(IBM United Kingdom Laboratories Ltd.)
J. D. Tinsley
(St. Edward's School, Oxford)

NOVEMBER 1969

215-8017-0

A First Course on the Schools Computer

D. M. Taub
(IBM United Kingdom Laboratories Ltd.)
J. D. Tinsley
(St. Edward's School, Oxford)

November 1969

IBM United Kingdom Laboratories Limited
Hursley Park
Winchester Hampshire

CONTENTS

Preface iii

1. How a computer differs from a desk calculator 1
2. The essential parts of a computer 2
   2.1 Representation of numbers 2
3. The store 4
4. Display 5
5. Loading the store 6
6. Arithmetic, INPUT and OUTPUT instructions 10
7. Arrangement of instructions in a program 12
8. Carrying out instructions 14
9. A program to compute the surface area and volume of a sphere 15
10. Running the program 19
11. Solution of two linear simultaneous equations 22
12. Compound interest 24
13. BRANCH instructions 26
    13.1 CONDITIONAL BRANCH instructions 26
    13.2 UNCONDITIONAL BRANCH instructions 27
14. Applications of CONDITIONAL BRANCH instructions . . . 28
    14.1 Compound interest 28
    14.2 Electricity account 29
    14.3 Area under a curve 32
    14.4 Square-root by the Newton-Raphson method 34
    14.5 Sin x by summation of a series 36
15. A program for constructing a bar chart 38
16. Indirect addressing 40
    16.1 Bar chart program using indirect addressing 42
17. Subroutines 46
    17.1 Solution of a quadratic equation 46
    17.2 Attenuation in a telephone line – OPEN and CLOSED subroutines 49
18. Problem-oriented languages 53
19. Conclusion 53

Appendices – Supplementary Examples and Exercises 55

A. Programs without branches 56
   A.1 Convert a temperature given in ^{o}F to ^{o}C 56
   A.2 Daily shop sales 57
B. Programs with branches 62
   B.1 Motor mileage allowance 62
   B.2 Highest common factor 64
C. Programs using indirect addressing 66
   C.1 Processing an array 66
   C.2 Sorting into descending order – 'Shunt sort' 72
   C.3 Daily shop sales – using indirect addressing 74
D. Subroutines 78
   D.1 Finding the maximum number in a given array 78
   D.2 Finding how many numbers in a given array fall within a given range 79

Summary of 3-address instructions used in this course 83

FIGURES

Fig. No.

1. Essential parts of a computer 2
2. Representation of store 4
3. Store and address register 4
4. Data format 6
5. Loading the store 7
6. Keyboard 8
7. Appearance of display while entering a number into store . 9
8. Instruction format 10
9. Illustration of ADD instruction 7023 2122 11
10. Data-flow diagram showing additional paths used for loading and updating the IAR and for fetching instructions . . . 13
11. Data-flow diagram showing additional paths used in arithmetic, INPUT and OUTPUT instructions 14
12. Appearance of display after loading the IAR 19
13. Example of display at stages A, B and C of an INPUT instruction 20
14. Example of output display 21
15. Data-flow diagram showing additional paths used in BRANCH instructions 27
16. Area under a curve 33
17. Final data-flow diagram 41
18. Display at stage A of an instruction using indirect addressing 42

PREFACE

The Schools Computer is designed for children at Secondary School level. It has three important purposes:

a) To show how a computer is organised and how it carries out its work, combining many simple operations to perform a complex calculation.

b) To be used in the mathematics course for demonstrating important concepts and techniques, e.g. iterative processes, calculation of functions, numerical integration etc.

c) To be used for elementary training in programming; covering mathematical, commercial, statistical and other data-processing applications.

Full details of the machine are given in a separate publication* but it may be helpful to restate its main features here. The machine contains a store, a group of general-purpose registers and an add-subtract unit. In the store, numbers are held in floating-point form, but in the registers they are always in integer form. The basic instructions set consists of:

i) register instructions: for carrying out arithmetic, shifting and testing operations on the contents of registers, and

ii) single-address instructions: for moving data into and out of the store, and for branching.

For most applications it would be much too tedious to write programs in the basic instruction set, so the machine also has a set of 3-address floating-point instructions. These are implemented in the same way as the 'extracode' instructions used in Atlas, i.e. by automatically calling sub-routines of basic instructions held in a protected part of the store.

Many important aspects of computer organisation and programming can be taught without burdening the student with the basic instruction set; he can work entirely with the 3-address instructions. This is the approach used in the present course.

The course starts by explaining how a computer differs from a calculating machine. It introduces the idea of a stored program and the representation of instructions by numbers. The essential parts of a computer are then considered: the store, the means of input and output, the arithmetic unit and the control unit.

The store is then discussed in more detail, and the role of the address register pointed out. The relevant parts of the keyboard and display are covered, and the student shown how to enter numbers into any storage location.

At this stage the basic arithmetic instructions and the input and output instructions are introduced. The student is taught how to combine them into a program and how the machine carries them out. This is followed by an example of a simple formula-evaluation program, computing the surface area and volume of a sphere. The student is shown how to load the program, run it, and how to follow it on the display screen. A second example given at that stage is the solution of two linear simultaneous equations.

* Taub, Owen and Day: An Experimental Computer for Schools.
IBM United Kingdom Laboratories, 1969.

The next example is a compound interest calculation. The problem is very simple but clearly demonstrates the importance of branching. The branch instructions are defined and an explanation given of how the machine carries them out. Several programs are given showing how branch instructions are used: computation of compound interest, area under a curve, extraction of a square root by the Newton-Raphson method and summation of the series for $\sin x$. Besides these mathematical examples there is a simple data-processing example in which the emphasis is on the logical structure of the problem rather than on mathematical techniques.

Following on, a program for constructing a bar chart is considered. Again, the problem is simple, but it calls for more sophisticated methods of addressing. The Schools Computer is capable of both relative and indirect addressing, but this course covers only the latter. The reason for leaving out relative addressing is that it requires the use of register instructions, and these are not covered until a later course.

Subroutines are the next item to be covered. The difference between open and closed subroutines is explained, and the Branch and Link instruction introduced. Two examples of subroutines are given. In one, the square-root program from an earlier example is used as an open subroutine in the solution of a quadratic equation; in the other, it is used as a closed subroutine in computing the performance of a telephone line.

The course itself finishes with a brief mention of problem-oriented languages. It is followed by a set of supplementary examples and exercises covering mathematical and commercial applications.

D.M.Taub

J.D.Tinsley

First issued June 1967 under the title: **An outline course on the Schools Computer.** Revised March 1969, August 1969.

Note: The publication of this document does not constitute a grant of any licence under any patent.

1. HOW A COMPUTER DIFFERS FROM A DESK CALCULATOR

Computers and desk calculators have one basic feature in common: both carry out the fundamental operations of arithmetic: addition, subtraction, multiplication and division. However, there are some important differences between them which make the computer a far more powerful and versatile instrument.

A desk calculator needs the attention of its operator throughout a calculation, pressing keys to enter numbers and to make it add, subtract, multiply or divide. We can say that the operator gives the machine its instructions *as the calculation proceeds.*

A computer, on the other hand, works automatically. The operator provides it in advance with a list of instructions covering the whole calculation. The list of instructions is called a PROGRAM.

The speed at which a calculation can be done on a desk calculator depends mainly on how fast the operator can press the keys. Increasing the speed of the internal mechanism is of no help beyond a certain point; the machine would simply spend a larger proportion of its time waiting for the next key to be pressed.

In a computer, this limitation does not apply. Once the program and any initial data have been entered and the machine started, it needs no further attention. It is therefore designed to work as fast as is economically possible. For instance, a modern fast machine can add two multi-digit numbers in about one millionth of a second.

Each instruction in a computer is represented by a number according to some prearranged code. This idea may seem strange at first, but it is not new. For at least fifty years, automatic telephone systems have used numbers to represent 'routing' instructions; for example, when a person in Winchester dials 93 he is giving the switches at the exchange an instruction to find him a line to Southampton.

The set of numbers used to represent the various instructions a computer can perform is called its INSTRUCTION CODE. Details of the instruction code of the Schools Computer are given later in this course.

2. THE ESSENTIAL PARTS OF A COMPUTER

The essential parts of a computer are shown in Figure 1. They are:–

a) A STORE to hold the numbers representing data and instructions.
b) Means for entering these numbers into the store – an INPUT device.
In the Schools Computer it is a keyboard; larger machines use punched card readers, punched paper tape readers etc.
c) Means for reading results from the store – an OUTPUT device.
The Schools Computer uses a TV set to display results; larger machines use printers, card punches, etc.
d) An ARITHMETIC UNIT to carry out the basic operations of +, –, x and ÷.
e) A CONTROL UNIT to control the movement of instructions and data within the machine, i.e. to "tell the machine what to do next".

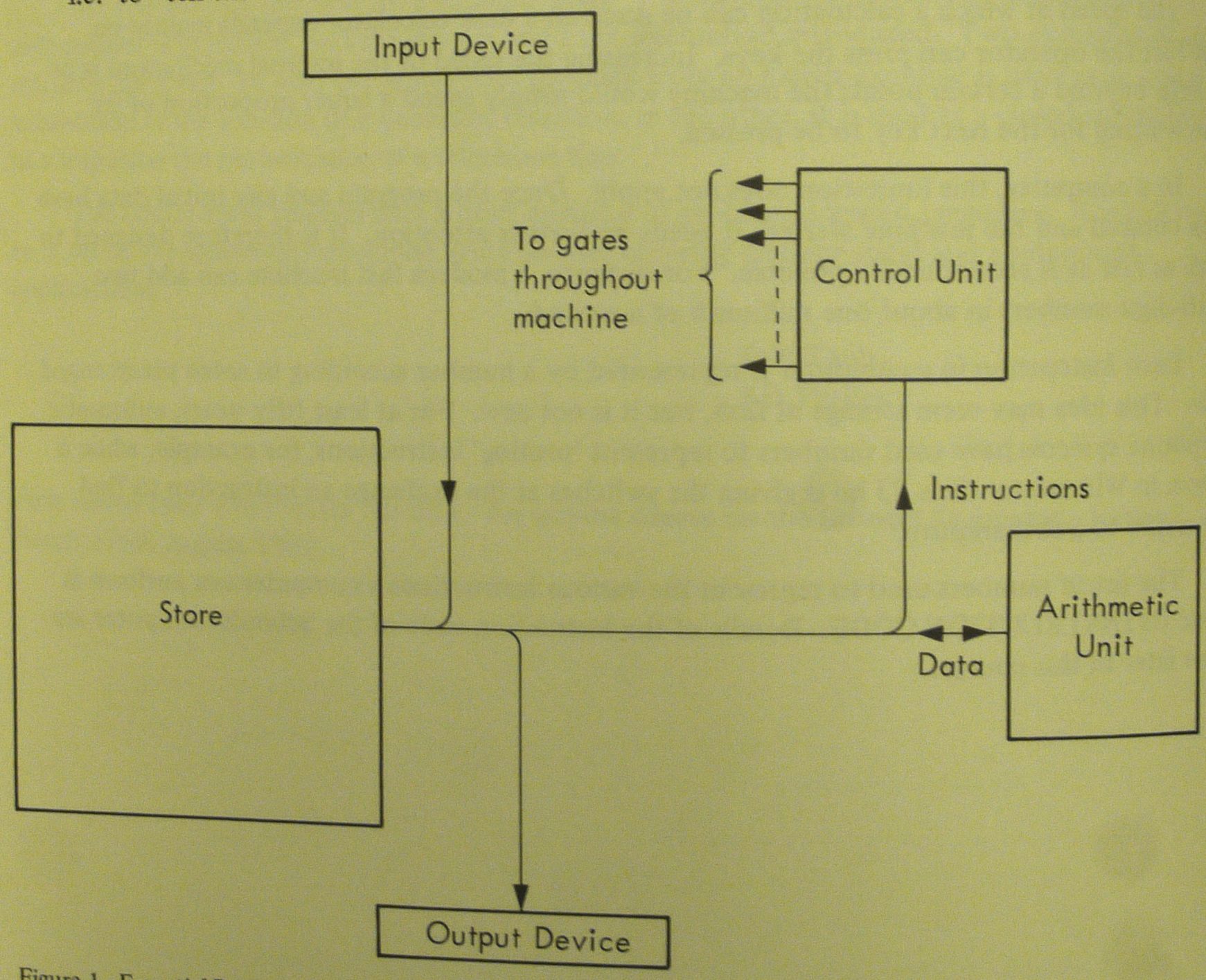


Figure 1. Essential Parts of a Computer

2.1 Representation of Numbers

In electronic digital computers, numbers are always represented in some form of binary code. This is because the basic storage devices and circuits can distinguish between only two conditions. For instance, the various circuits used in the Arithmetic and Control Units can distinguish between only two different voltage levels. One of these levels is arranged to represent the binary digit 0, and the other 1. The Store consists of a large number of tiny magnetic rings, each of which can be magnetised either clockwise or anticlockwise. One direction of magnetisation is arranged to represent 0, and the other 1.

Many computers use what is known as 'pure binary' notation, in which successive digits represent successive powers of 2. Thus, 26 would be represented by 11010. This code is economical and efficient, but it has certain disadvantages. In the Schools Computer, a different code is used. Instead of representing a whole multi-digit number in binary code as above, each decimal digit is individually represented by its binary equivalent. As there are 10 decimal digits, each one requires 4 binary digits to represent it. Thus, 26 is represented by 0010 0110, the first four digits, 0010, being the binary representation of 2, and the second four, 0110, the binary representation of 6.

The above method of coding numbers is called BINARY-CODED DECIMAL. Its advantage is that it avoids having to convert the numbers from decimal to pure binary code when they are loaded into the machine, and back again to decimal on output. Only the individual decimal digits have to be converted, which is a much easier task.

3. THE STORE

The store of a computer is divided into a number of compartments or LOCATIONS, each of which can store a number several digits long. The number can represent a piece of data or an instruction. The Schools Computer has 100 locations, but larger machines often have many thousands. In order to identify the separate locations, they are numbered in sequence, usually starting at 0. Thus the locations in our computer are numbered 0 to 99. The number that identifies a particular location is called its ADDRESS, and the contents of a location, whether it represents data or an instruction, is called a WORD. We can thus represent the store as shown in Figure 2.

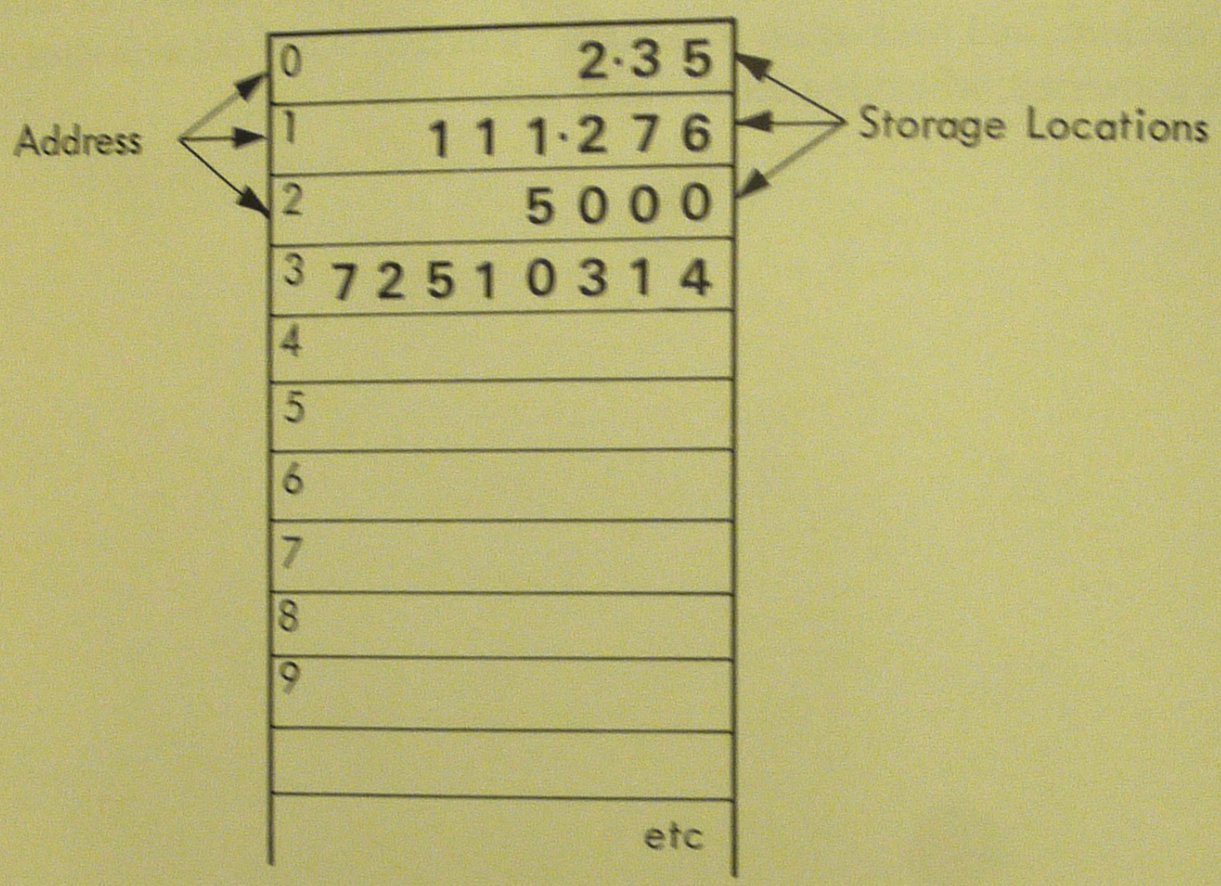


Figure 2. Representation of Store

To read a word out of the store, the store must be given the address of that word. Similarly, when a word is to be written into the store, the store must be given the address at which it is to be written. This address-information is held in an ADDRESS REGISTER connected to the store as shown in Figure 3.

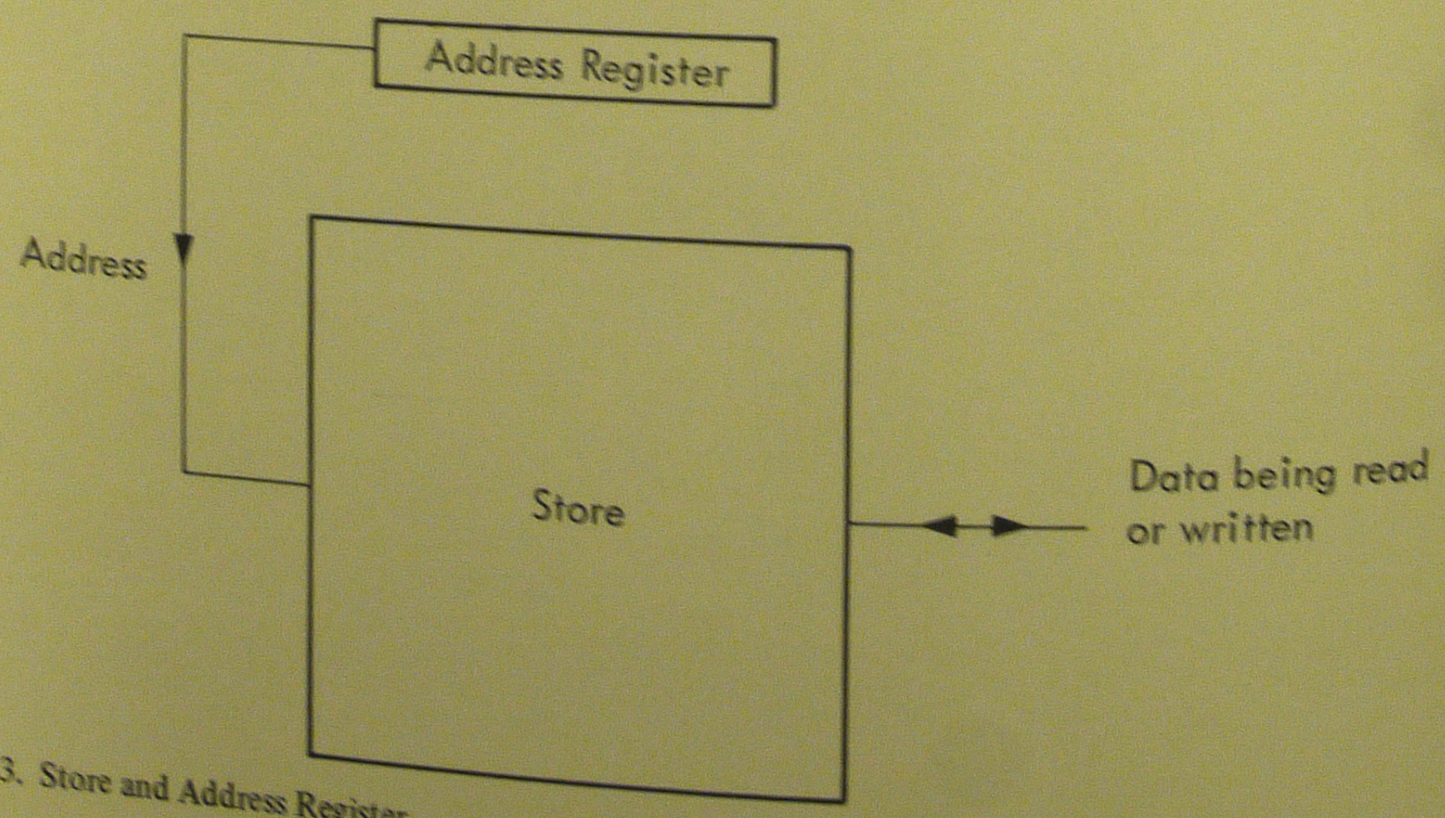


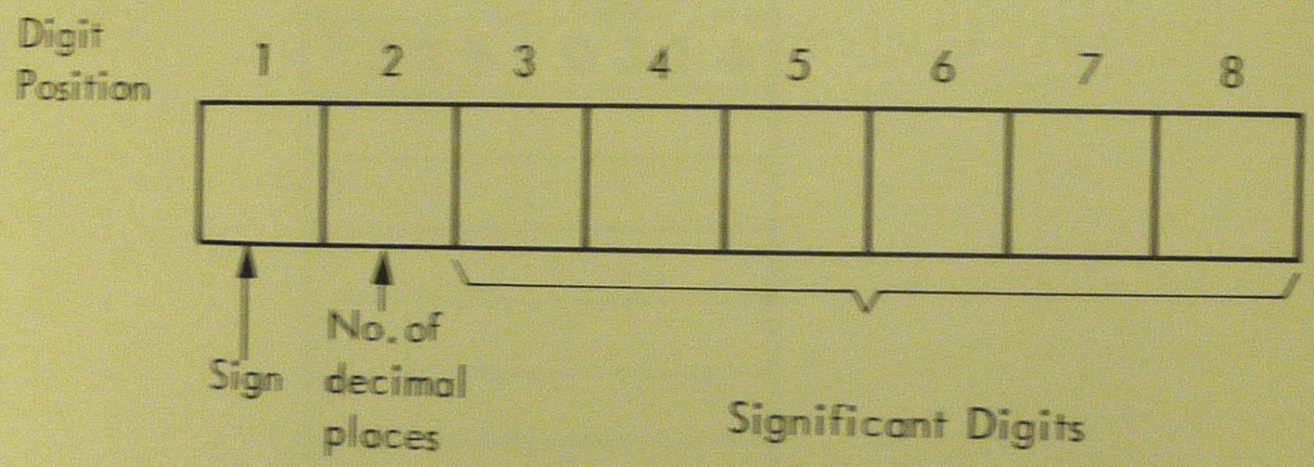
Figure 3. Store and Address Register

4. DISPLAY

The TV receiver is not used only as an output device. It can be made to display the contents of any storage location in the machine and also various registers, which will be discussed later. This is done to make it easy for the user to follow the action of the machine in detail. The layout of the display is explained in later sections.

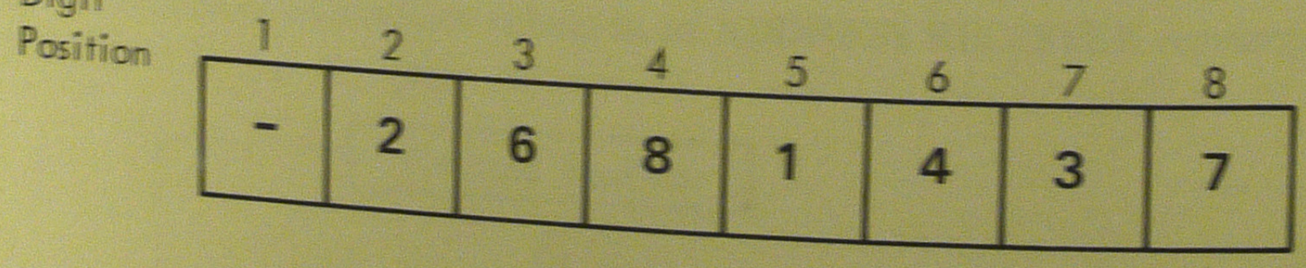
5. LOADING THE STORE

Before the machine can do any useful work, the store must be loaded with a program of instructions and any necessary data. Each instruction word consists of 8 decimal digits, the significance of which will be explained in later sections. Data words may have up to 6 decimal digits and a decimal point, and they may be positive or negative. Thus, the smallest finite number that can be represented is ± 0.000001, and the largest ± 999999. The reason why data words are restricted to 6 digits while instructions can have 8 is that the sign occupies the same storage space as a decimal digit and so does the method of representing the position of the decimal point. Thus all words, whether they represent instructions or data, occupy eight decimal digit positions in the store. The way in which the eight digit positions in a word are used when storing data, and the way in which their contents are displayed, are shown in Figure 4.



(a) Assignment of digit positions in word

In Store



As Displayed

6 8 | 4·3 7 −

(b) Example of data storage and display

Figure 4. Data Format

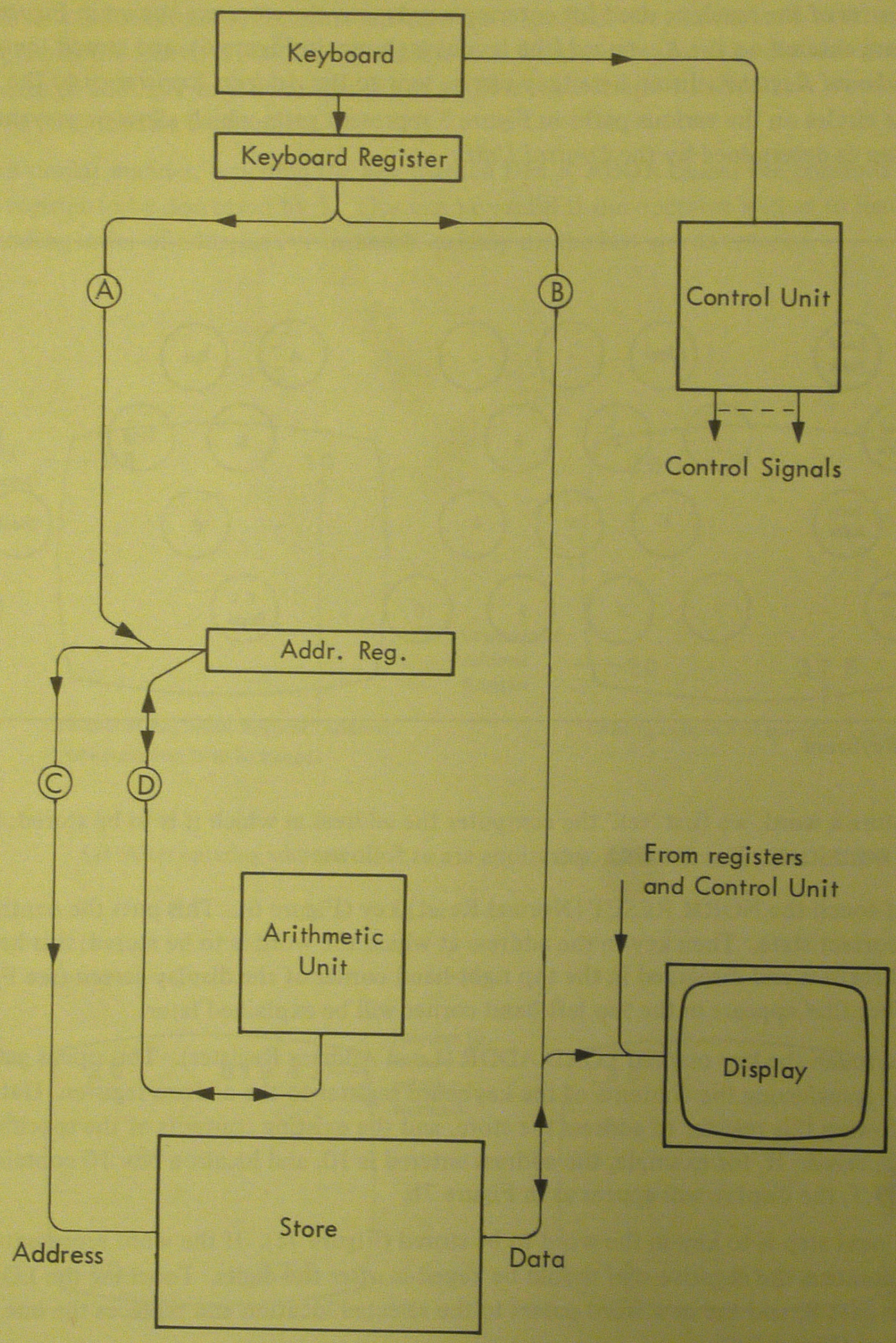


Figure 5. Loading the Store

The parts of the machine used for entering words into the store are shown in Figure 5. Numbers are entered on the *Keyboard* (the layout is shown in Figure 6), and stored temporarily in the *Keyboard Register.* From here they can be sent to the *Address Register* or to the store itself. The circles on the various paths in Figure 5 represent gates which allow or prevent the flow of data as determined by the *Control Unit.*

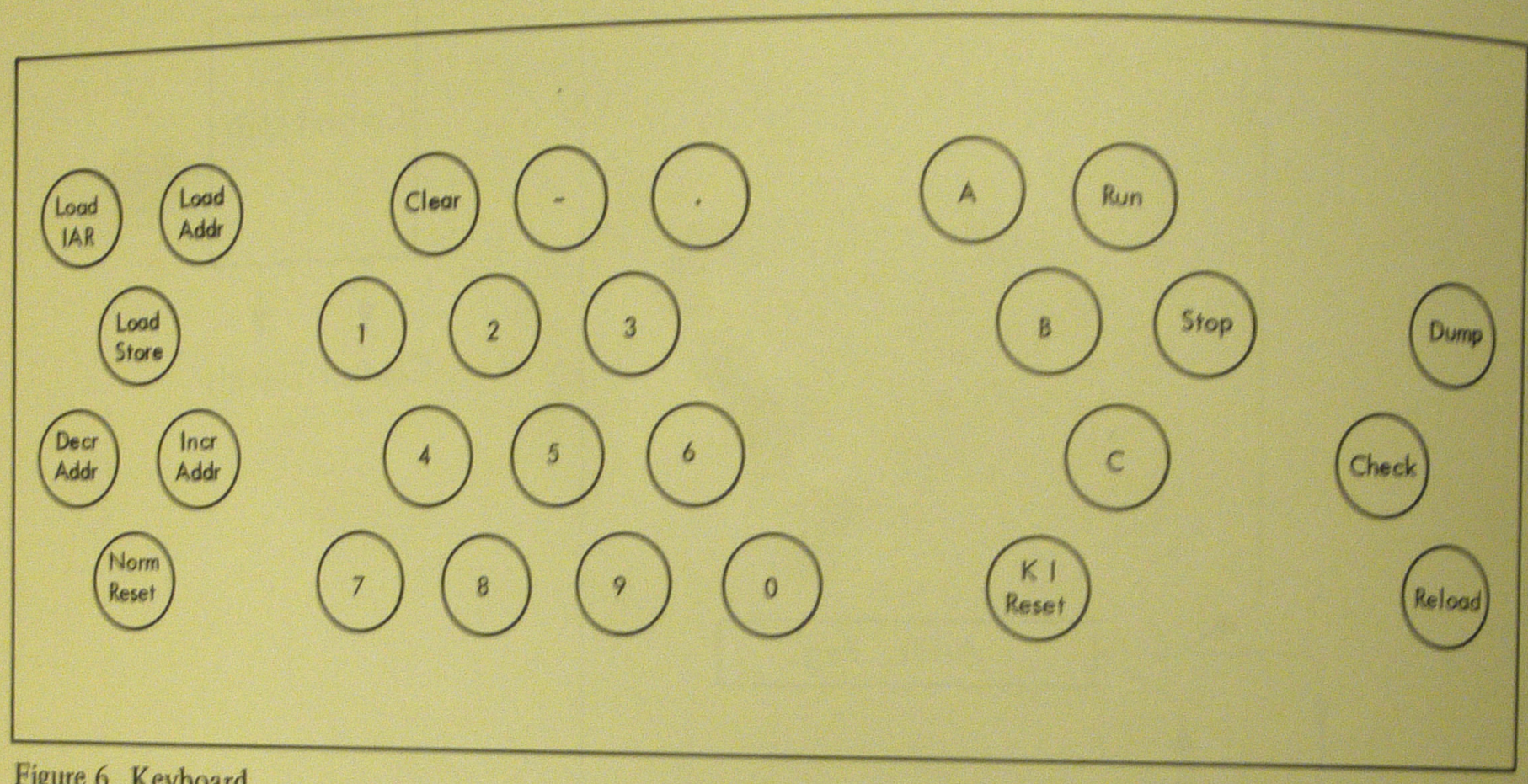


Figure 6. Keyboard

To store a word, we first 'tell' the computer the address at which it is to be stored; then we enter the word itself. The detailed operations are as follows:-

First touch the NORM RESET (Normal Reset) key (Figure 6). This puts the control unit into the correct state. Then key-in the address at which the word is to be stored; it is held in the keyboard register and displayed in the top right-hand corner of the display screen (see Figure 7a). * The number that appears in the top left-hand corner will be explained later.

Now touch the key marked LOAD ADDR (Load Address Register). This opens gate A (Figure 5) transferring the contents of the keyboard register to the address register. Gate C now opens, allowing this register to address the store, and the existing contents of the specified location will be displayed. If, for example, the address entered is 10, and location No. 10 contains the number 37.5, the display will appear as in Figure 7b.

The next step is to key-in the word to be stored (Figure 7c). If the word represents a negative number, the negative sign should be keyed-in *after* the digits. Touching the LOAD STORE key opens gate B, and the new word passes to the selected location and replaces the one that was there before (Figure 7d).

To load another word, key-in a new address and proceed as before. However, if you wish to load several store locations in sequence, for example when loading a program, there is a simpler way of doing so. Instead of keying-in the next address, touch the key marked INCR ADDR. This causes 1 to be added to the contents of the address register, as follows (see Figure 5):

i) Gate D opens, sending the contents of the address register to the *Arithmetic Unit.*
ii) The control unit generates a 1 and sends it to the arithmetic unit.
iii) The arithmetic unit adds these two quantities together and sends the sum back to the address register.

* If there is already a number in the keyboard register clear it by touching the CLEAR key.

The new address and its contents will now be displayed. Key-in the new word to be entered, touch the LOAD STORE key, and the new word will replace the old one as before. The process of adding 1 to the number in the address register and entering new words is continued until loading is complete.

In a similar manner, touching the key marked DECR ADDR causes the contents of the address register to be decreased by 1. This can be useful if the operator wishes to load the store in descending order of addresses or to check quickly on the last words entered.

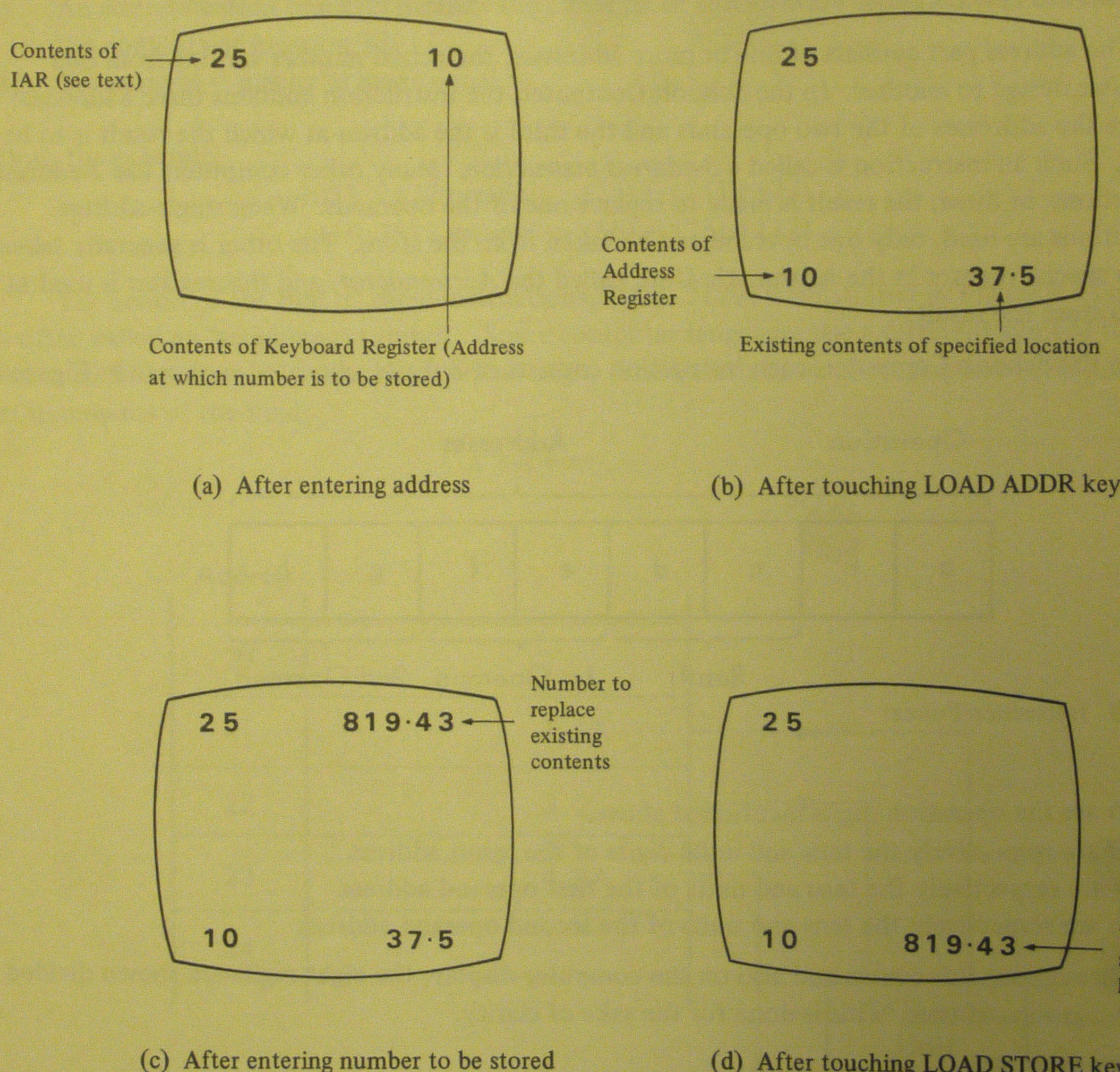


Figure 7. Appearance of display while entering a number into store.

6. ARITHMETIC, INPUT AND OUTPUT INSTRUCTIONS

So far we have regarded the instructions simply as groups of digits. We must now consider them in more detail.

An instruction is made up of two parts: an OPERATION part and an ADDRESS part. The operation part indicates which operation is to be carried out, e.g. in the Schools Computer:

70 means ADD
71 means SUBTRACT
72 means MULTIPLY
73 means DIVIDE
78 means INPUT
79 means OUTPUT

The address part consists of one or more addresses, the actual number varying from one computer design to another. In the Schools Computer, the instruction contains three addresses: two are the addresses of the two operands and the third is the address at which the result is to be stored. Such an instruction is called a *3-address instruction.* Many other computers use 2-address instructions; in these, the result is made to replace one of the operands. When single-address instructions are used, only one operand can be taken from the store. The other is generally taken from a special register in the Arithmetic Unit, called the *Accumulator,* and this register is used in turn to store the result.

In the Schools Computer, each instruction consists of 8 digits, allocated as shown in Figure 8.

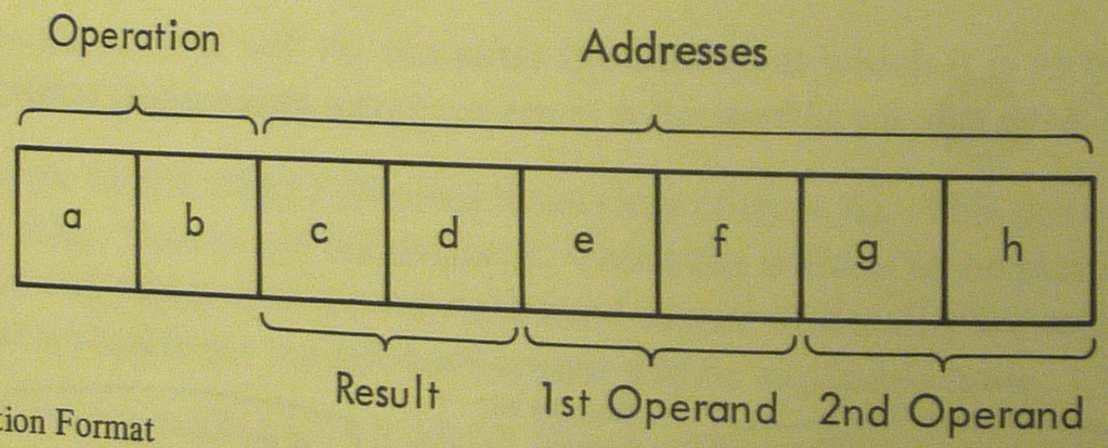


Figure 8. Instruction Format

a and *b* are the operation digits mentioned above.
c and *d* are respectively the tens and units digits of the result address.
e and *f* are respectively the tens and units of the first operand address.
g and *h* are respectively the tens and units of the second operand address.

Throughout this course and also on the computer display, the eight digits are shown divided into two groups of four. This is done for the sake of clarity.

Thus for example:

7023 2425 means: Add the contents of locations 24 and 25, and store the result in location 23.

7356 7812 means: Divide the contents of location 78 by the contents of location 12 and store the result in location 56.

7923 5617 means: Display the contents of locations 23, 56 and 17.

7845 0621 means: Store the number entered on the keyboard in location 45, and display the contents of locations 6 and 21.

Strictly speaking, one would expect an INPUT instruction to contain only one address, the address at which the input information is to be stored. But as the instruction has room for two more addresses it was decided to use them for display purposes as indicated. This is often useful in practice.

All the instructions shown above can handle integers or numbers containing a decimal point. In the case of arithmetic instructions, results are rounded as follows before being sent back to the store:

i) For numbers greater than 1, the result is rounded to six significant figures.
ii) For numbers less than 1, the result is rounded to six decimal places.

The arithmetic instructions can be defined more briefly by using the following conventions:

i) An address within brackets means: the *contents* of the location specified, e.g. (32) means: the contents of location 32.
ii) The symbol ← means 'is made equal to'.

Thus, for example:

7023 2122 means: (23) ← (21) + (22)
7356 7812 means: (56) ← (78) ÷ (12)

The action of the first of these instructions, 7023 2122, is illustrated in Figure 9. The arrow is often called an 'assignment symbol'. Some computer languages use a different symbol to mean the same thing, e.g. := or simply an 'equals' sign. The latter can be confusing however, particularly in statements of the kind:

$$x = x + x$$

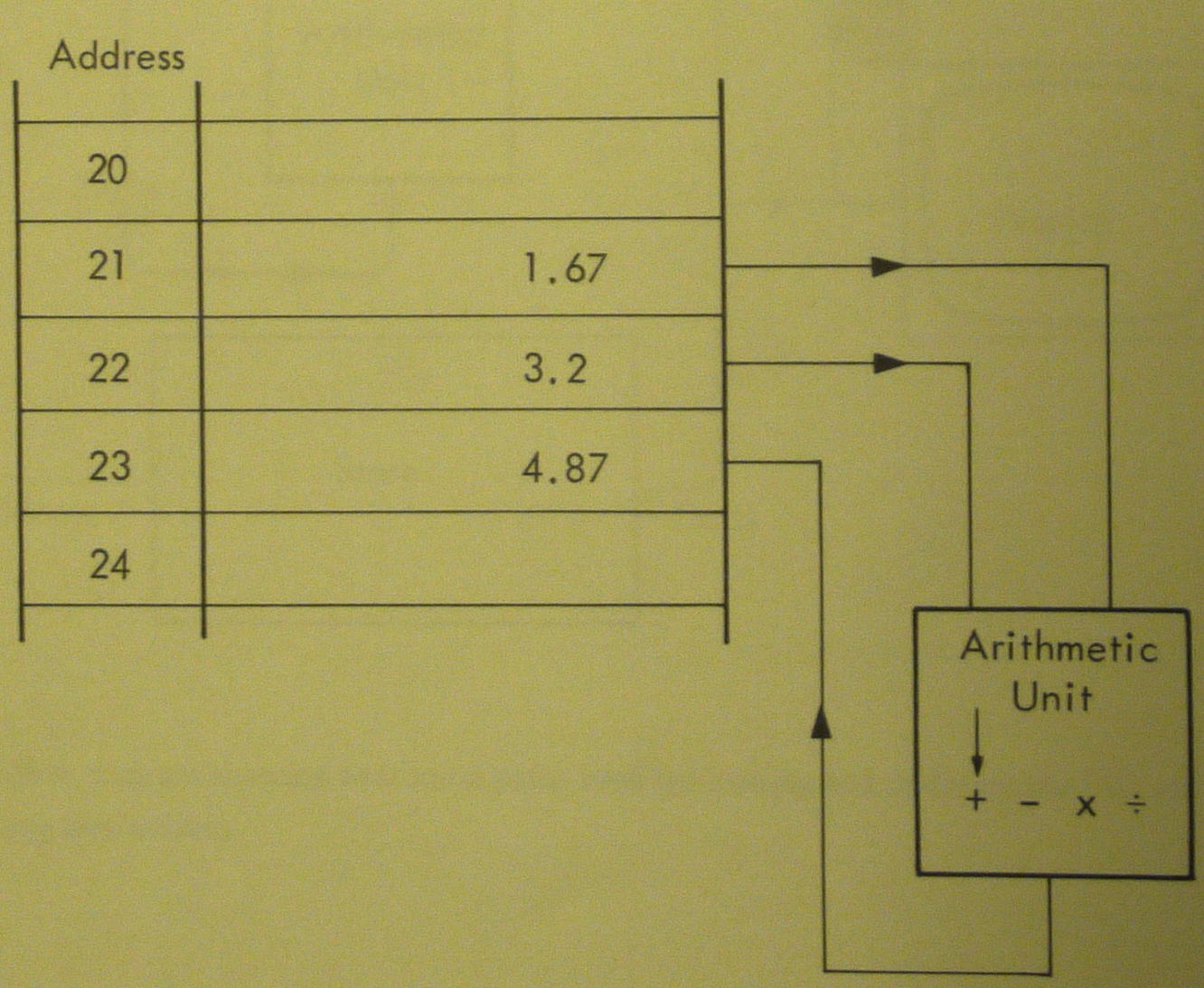


Figure 9. Illustration of 'ADD' Instruction 7023 2122

7. ARRANGEMENT OF INSTRUCTIONS IN A PROGRAM

When a computer is carrying out a program it normally takes successive instructions from successive locations in the store. For example, if it takes the first instruction in a program from location 10, it will take the second from location 11, the third from 12 and so on. This means that when the program is being loaded into the machine it is important to put the instructions into storage locations in the correct order.

Clearly, while the program is being run, there must be some way for the machine to keep track of the point that has been reached; otherwise the machine would not 'know' which instruction to do next. The task of keeping track of the program is carried out by a register called the INSTRUCTION ADDRESS REGISTER, usually abbreviated to IAR (see Figure 10).

Before a program can be run, the IAR must be loaded with the address of the first instruction; how to do this is explained in Section 10. Then, as soon as the machine is started, the IAR is made to address the store and the first instruction is read out and placed in the INSTRUCTION REGISTER. The required operation is then carried out (see Section 9), and while this is being done, the arithmetic unit adds 1 to the IAR contents. Thus, when the operation has been completed the IAR contains the address of the instruction that follows. The IAR is then made to address the store again, the second instruction is read out, and so on.

Figure 10 shows an auxiliary register just below the IAR. This register plays no direct part in the operation of the machine but it helps the user when he is following through a program. As soon as an instruction has been read from the store, the address of that instruction is copied from the IAR into the auxiliary register, and it remains there until the following instruction is read out. Thus, when an instruction has just been completed, the auxiliary register contains the address from which that instruction was read and the IAR contains the address of the following instruction. The contents of these registers are shown on the display as described in Section 10.

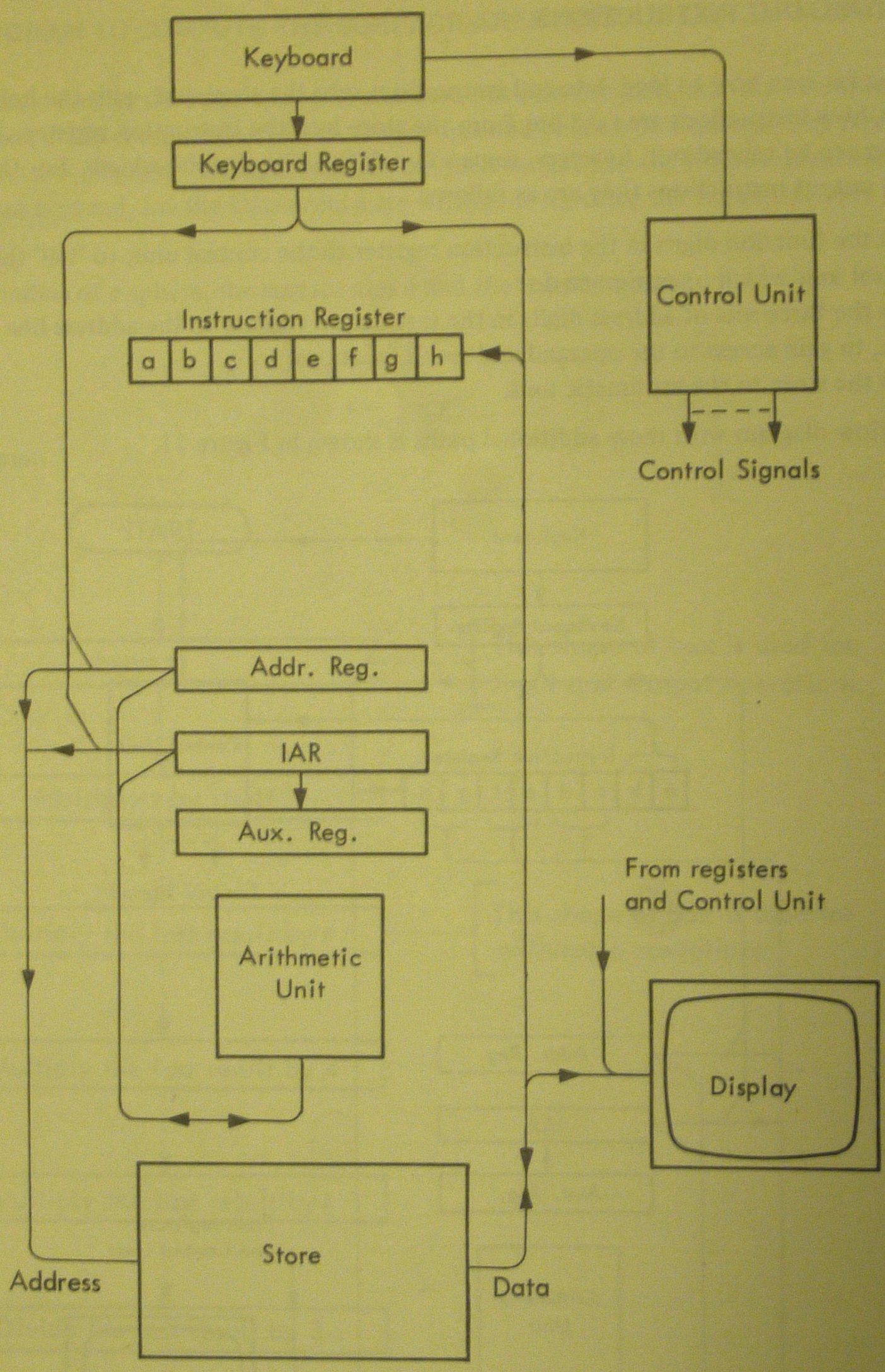


Figure 10. Data-flow diagram showing additional paths used for loading and updating the IAR and for fetching instructions

8. CARRYING OUT INSTRUCTIONS

We have so far seen how to load data and instructions into the store and, with the help of Figure 10, how instructions are read out from the store into the instruction register. Before an instruction can be carried out, however, some extra data paths must be added. For the arithmetic, input and output instructions they are as follows:

a) from the function digits in the instruction register to the control unit, to 'tell' the control unit which operation to do;
b) from the three sets of address digits in the instruction register to the address line of the store, to gain access to the operand and result locations;
c) from the store to the arithmetic unit.

The data-flow diagram with these additional paths is shown in Figure 11.

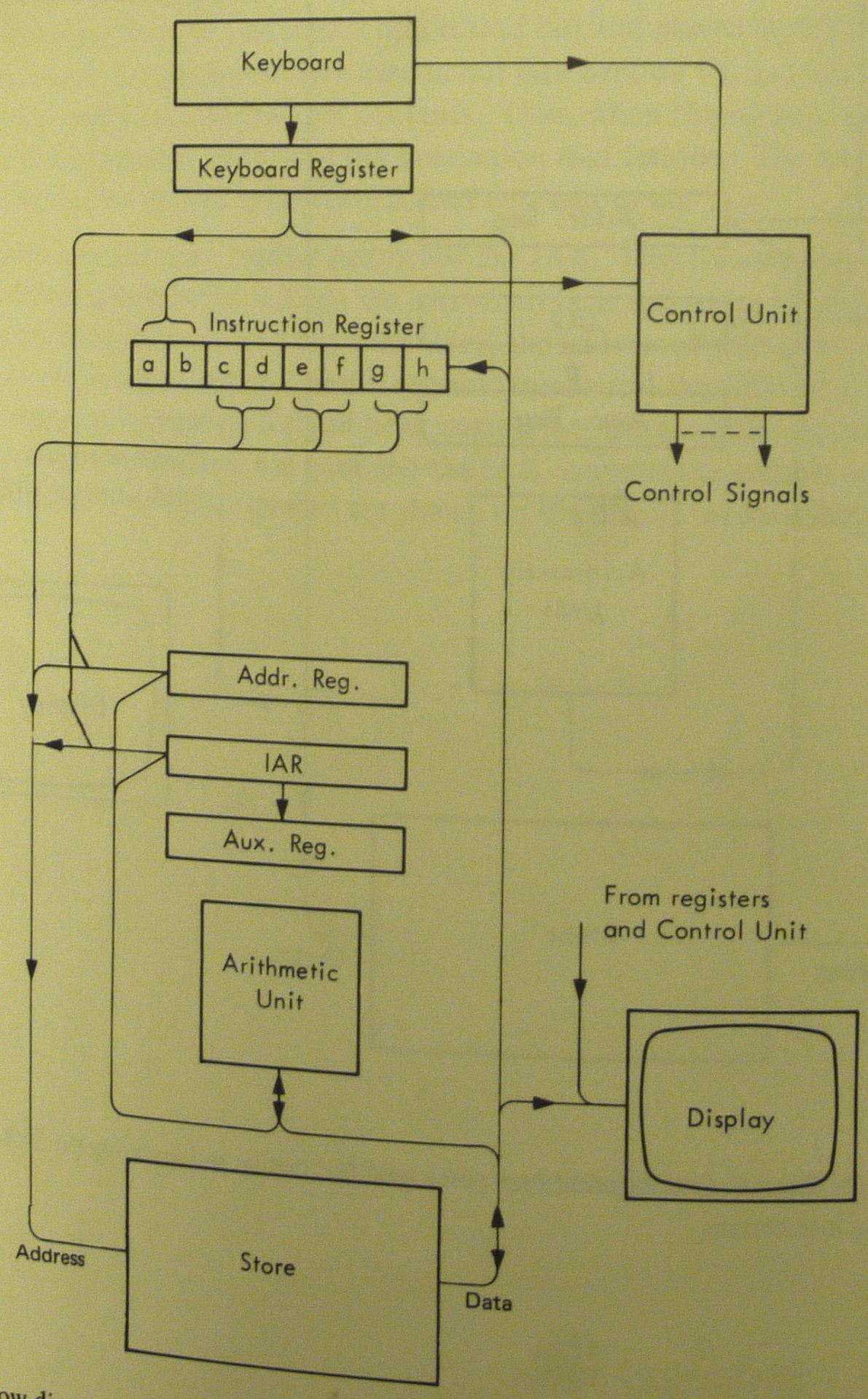


Figure 11. Data-flow diagram showing additional paths used in arithmetic, Input and Output instructions

9. A PROGRAM TO COMPUTE THE SURFACE AREA AND VOLUME OF A SPHERE

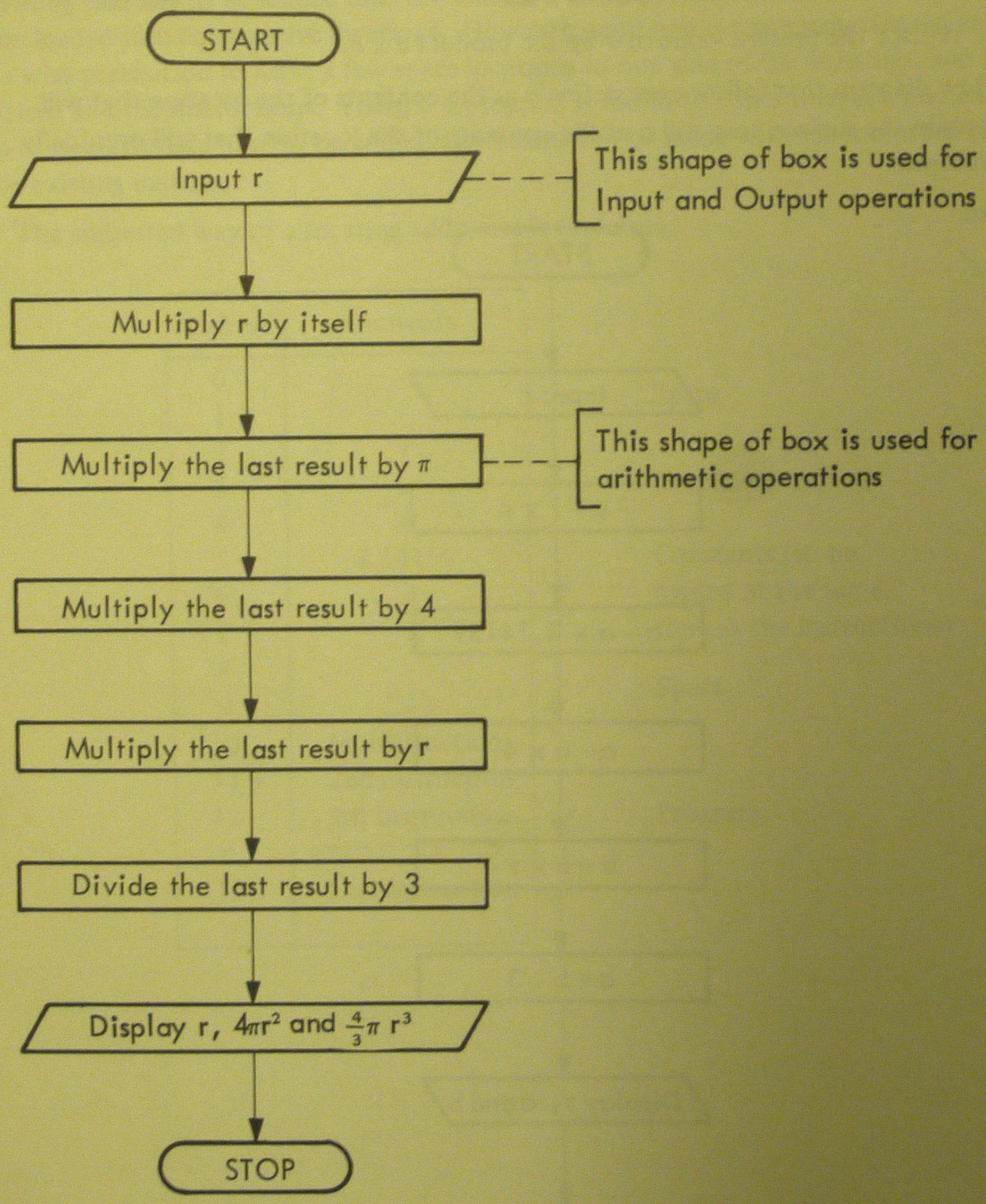
The student is now at the stage where he can write his first program, and the example we shall consider first is the computation of the surface area and volume of a sphere. The first step is to construct a flow diagram. Storage locations are then assigned to the various quantities and finally the program is coded, i.e. the instructions are written down using the numerical code given in Section 6.

If r is the radius of a sphere, the surface area s and the volume v are:-

$$s = 4 \pi r^2$$

$$v = \frac{4}{3} \pi r^3$$

Flow Diagram



This flow diagram shows what operations are to be done and in what order, but it tells us nothing about where the various interim and final results are to be stored. At this stage we do

not need to assign actual storage locations, but it is important to know how many are needed and to be able to distinguish between them. This can be done in the following way. We use letters to represent quantities, but we do not use them in quite the same way as in algebra. In algebra, a letter denotes some factor such as the radius of a circle, the height of a building or simply some mathematical function. In computer usage it denotes the contents of a storage location; *a* will be the contents of one location, *b* the contents of a second location, and so on. This allows us to refer to the various quantities in a computation before actual addresses have been assigned to them.

The point about using letters in this way is that a single letter will represent different numbers at different stages of a computation; it may even represent quite unrelated quantities, when those quantities take turns to use the same store location.

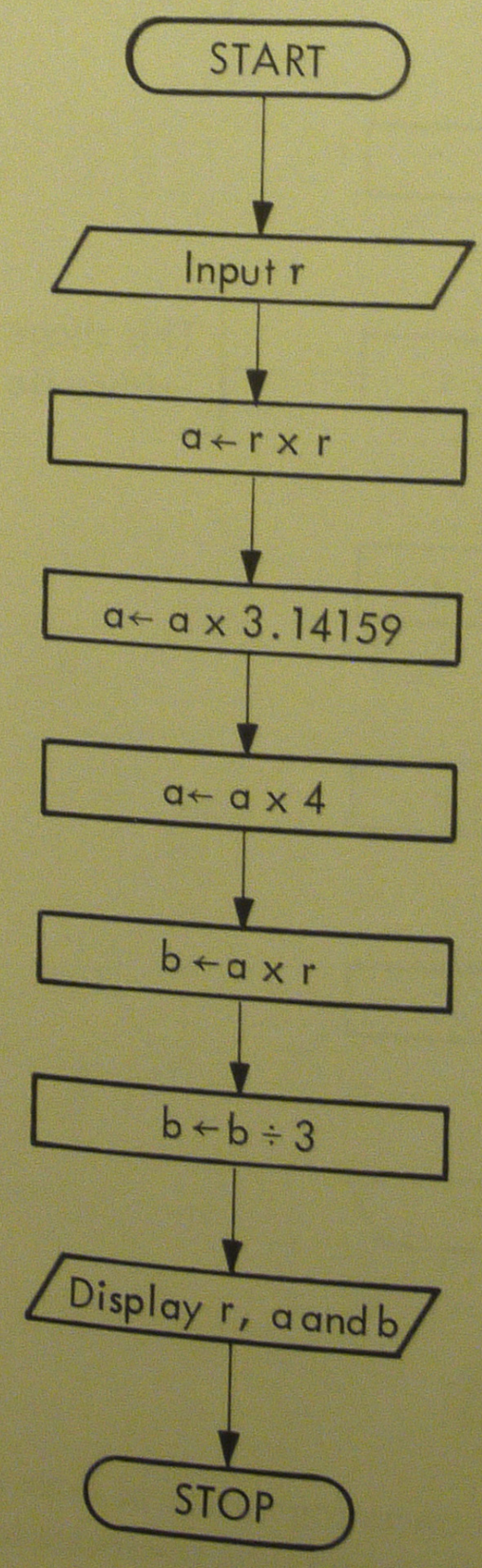
A flow diagram may thus contain expressions of the following type:

$$a \leftarrow a \times b$$

which means: replace the present value of *a* by the product $a \times b$.

In the flow diagram that follows, we define *a* as the contents of the location that will eventually contain the surface area and *b* as the contents of the location that will eventually contain the volume.

Flow Diagram



At this stage we are ready to allocate addresses to the various items of data in the calculation, *r*, *a*, 3.14159, 4, *b* and 3, and also to the seven instructions. The instructions have to be stored in consecutive locations (see Section 7) but the data can be stored wherever we please: in consecutive locations or dotted about the store. Usually, however, the most convenient way is as follows:

Divide the items of data into two groups:

a) the *constants*, in this case 3.14159, 4 and 3, which have to be loaded into the machine at the same time as the instructions, and
b) the *variables*, *r*, *a* and *b*, which are not given any numerical value until the program is actually run.

One possibility would be to store the variables in locations 0, 1 and 2, the constants in 3, 4 and 5 and the instructions in locations 6 to 12. The point in arranging them like this is that everything that has to be loaded into the machine initially, i.e. the constants and the instructions, can be loaded into consecutive locations. However, until one is more experienced at programming, it is a wise precaution to leave a few spare locations to one side of the variables, and between the constants and the instructions. Then, if a program is found to contain errors, and extra instructions or items of data have to be added, alterations can be made without too much re-arrangement of the existing material.

The suggested way of allocating addresses is therefore:

Address	Contents	
0	-	Spare
1	-	
2	*r*	Variables
3	*a*	
4	*b*	
5	3.14159	Constants (to be loaded at the same time as the instructions)
6	4	
7	3	
8	-	Spare
9	-	
10	1st instruction	Program
11	2nd instruction	
12	3rd instruction	
–	– – – – – – – –	
16	Last instruction	

The instructions can now be written in coded form thus:

	Address	Instruction	
		Coded	Meaning
Start →	10	7802 0507	Input r ; display 3.14159 and 3 (Note 1)
	11	7203 0202	$a \leftarrow r \times r$
	12	7203 0305	$a \leftarrow a \times 3.14159$
	13	7203 0306	$a \leftarrow a \times 4$
	14	7204 0302	$b \leftarrow a \times r$
	15	7304 0407	$b \leftarrow b \div 3$
Stop ←	16	7902 0304	Display r, a and b (Note 2)

Notes:

1. Displaying the first and last items in the list of constants is a useful way of checking that the whole list has been entered.
2. A typical output display is shown in Figure 14.

Generally it is best to list the items of data and the instructions in a single table rather than having separate ones as above. The complete documentation of the program will then be as follows:

Program

	Address	Contents	Explanation
	0	-	Spare
	1	-	
	2	r	Variables
	3	a	
	4	b	
	5	3.14159	Constants
	6	4	
	7	3	
	8	-	Spare
	9	-	
Start →	10	7802 0507	Input r ; display 3.14159 and 3
	11	7203 0202	$a \leftarrow r \times r$
	12	7203 0305	$a \leftarrow a \times 3.14159$
	13	7203 0306	$a \leftarrow a \times 4$
	14	7204 0302	$b \leftarrow a \times r$
	15	7304 0407	$b \leftarrow b \div 3$
Stop ←	16	7902 0304	Display r, a and b

10. RUNNING THE PROGRAM

First of all, the constants and instructions must be loaded into the store as described in Section 5. For the program in Section 9, the detailed procedure is as follows.

a) Touch the NORM RESET key.
b) Key-in the address of the first constant (5).
c) Touch LOAD ADDR key.
d) Key-in the value of the first constant (3.14159).
e) Touch the LOAD STORE key.
f) Touch the INCR ADDR key.
g) Load the remaining constants similarly.
h) When the last constant has been loaded, touch the INCR ADDR key three times to skip over the spare locations.
i) Load the instructions in the same way as the constants.

The IAR must then be set to the address of the first instruction. This is done by keying-in the required address (10), and touching the LOAD IAR key. The display then appears as shown in Figure 12. The program can now be started by touching the RUN key or the A, B or C key. The use of these different keys will now be explained.

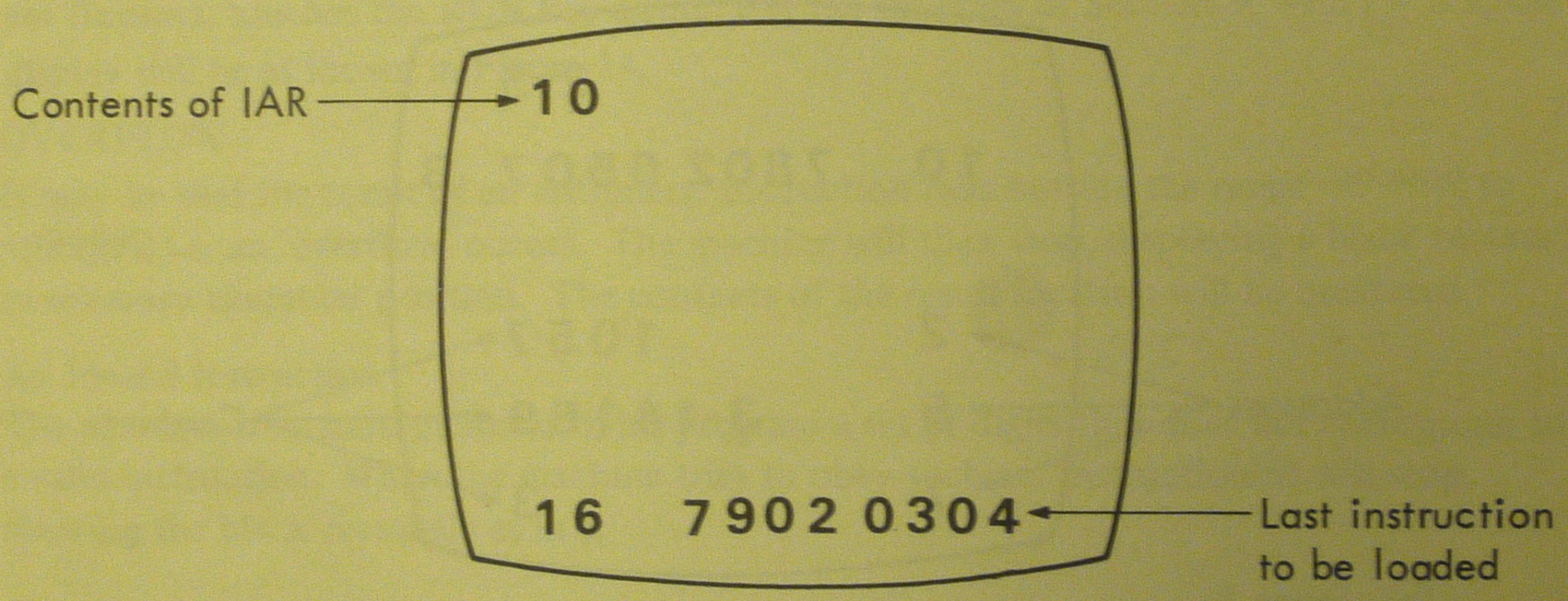
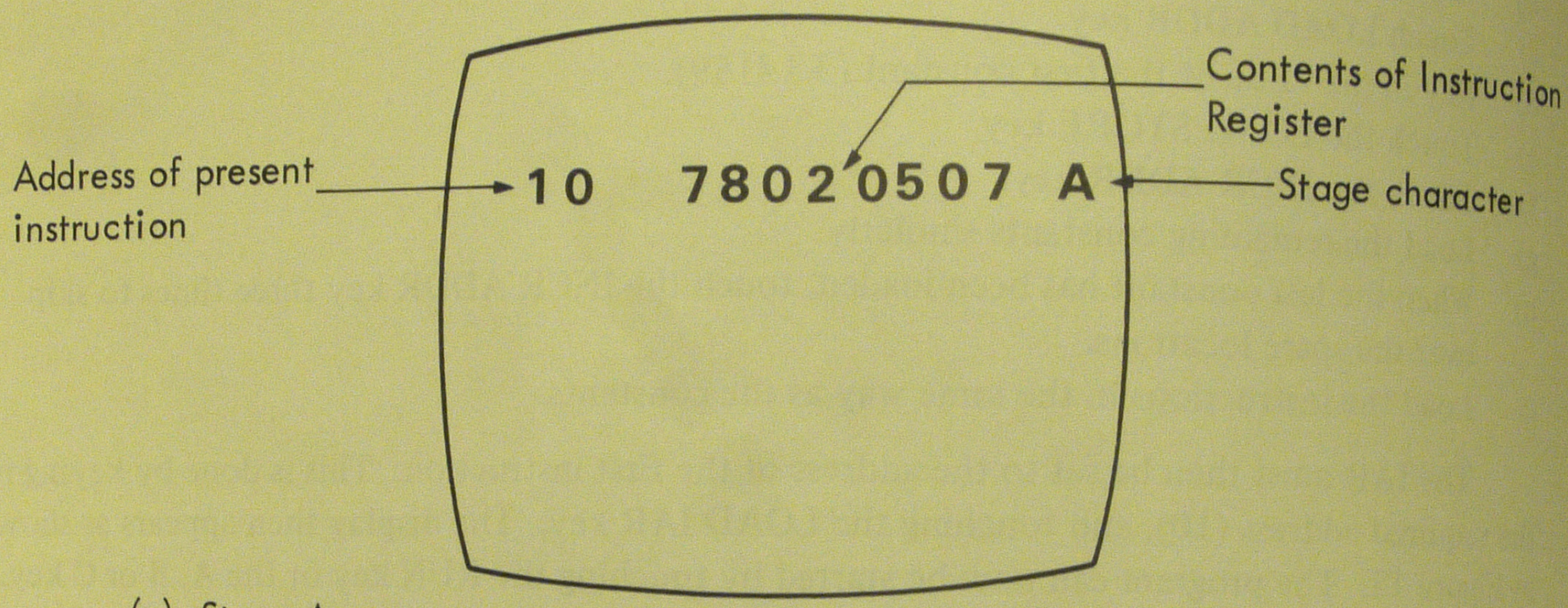


Figure 12. Appearance of display after loading the IAR

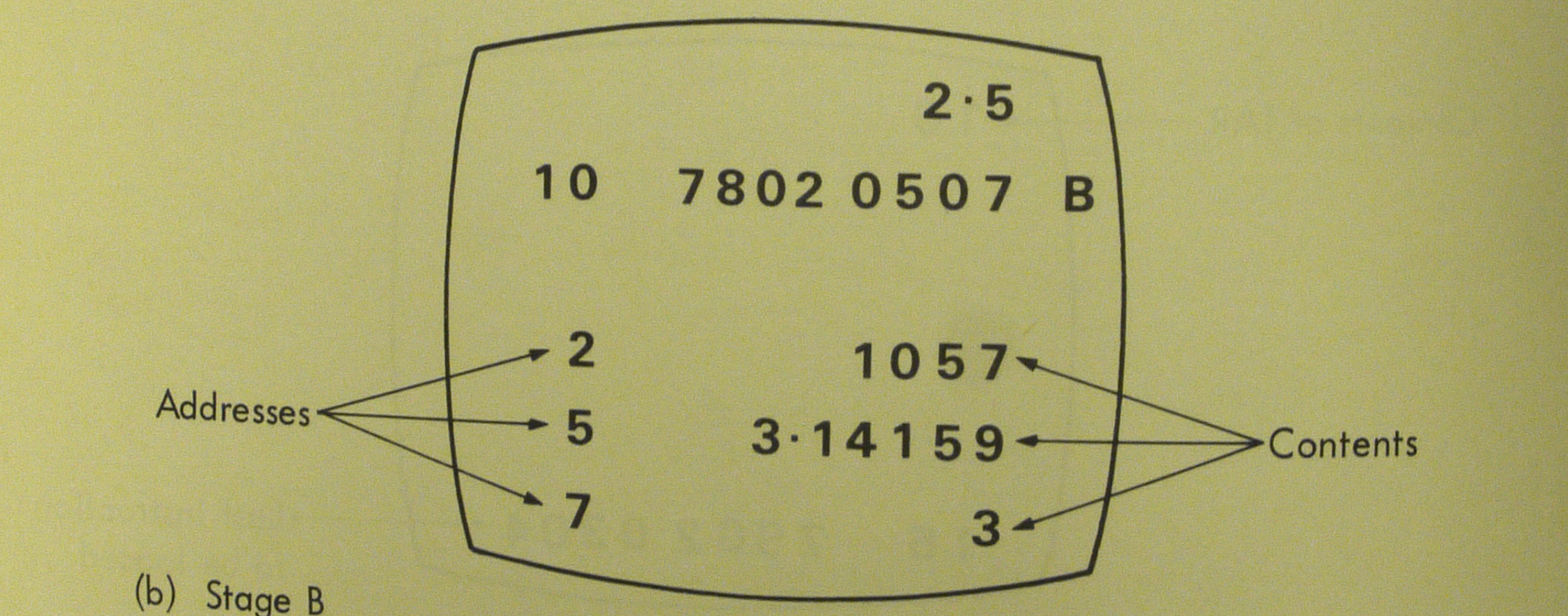
The A, B and C keys are used when one wishes to follow through the program step by step. The three letters represent stages in the processing of an instruction. At stage A, the instruction has been fetched from the store and set up in the instruction register. At stage B, the contents of the three relevant locations are displayed, but the instruction has not yet been carried out. At stage C, the instruction has been completed and the IAR incremented, giving the address of the following instruction. Touching the A key causes the instruction to be processed up to stage A, the B key up to stage B and the C key up to stage C.

Suppose the A key is touched. This causes the first instruction, INPUT *r*, to be read from the store and placed in the instruction register as shown in Figure 13a. The contents of the IAR are not shown at this stage, nor are they at stage B, but the contents of the auxiliary register, i.e. the address from which the instruction was read, appear immediately to the left of the instruction itself. When the B key is touched the existing contents of the three relevant locations will be shown. This is an INPUT instruction, and so the required value of *r* must now be entered on the

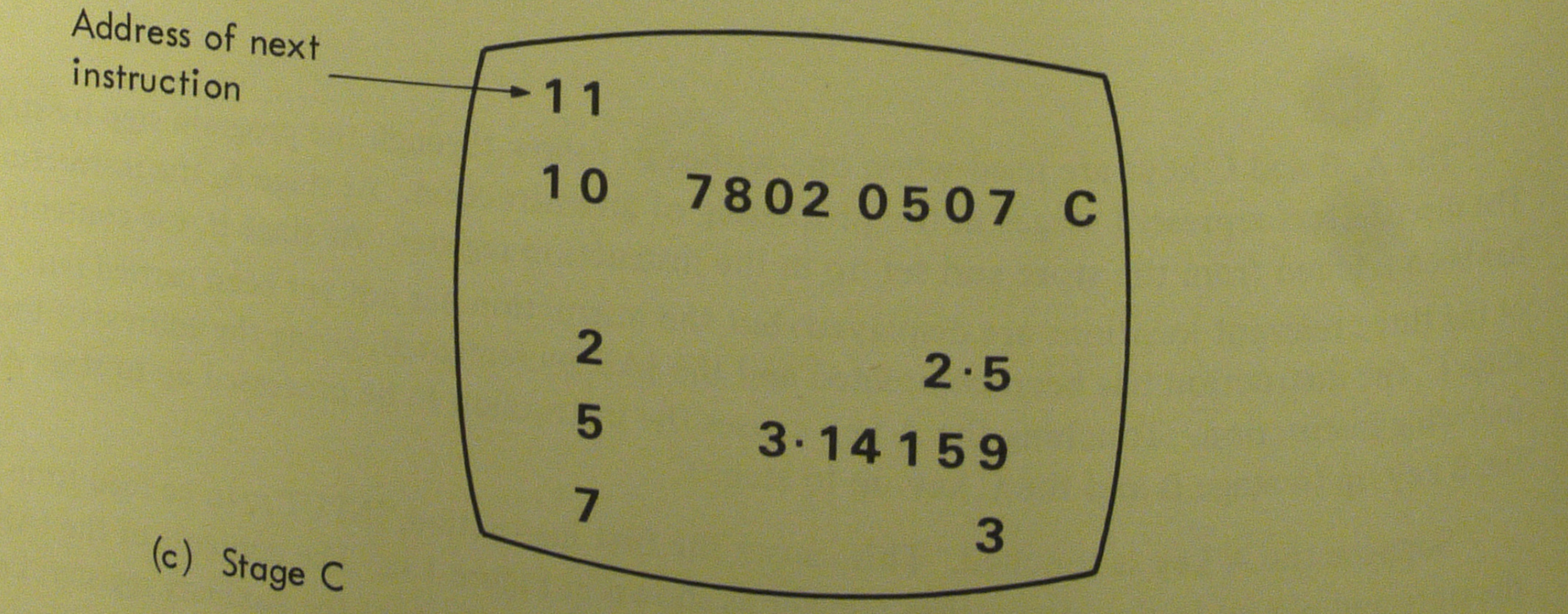
keyboard. Suppose it is to be 2.5; then the display will appear as in Figure 13b. Touching the C key will now cause the 2.5 to be stored in location 2 replacing the number that was there previously. The display at this stage is shown in Figure 13c. Note that the contents of the IAR are again displayed showing the address from which the next instruction will be taken. Touching the A, B or C key again will cause this instruction to be processed up to the corresponding stage, and so on.



(a) Stage A



(b) Stage B



(c) Stage C

Figure 13. Example of display at stages A, B and C of an Input instruction

It may appear from the above description that stage A is superfluous, since everything displayed at that stage is also displayed at stage B. However, in certain instructions, to be introduced later, stage A is used to display additional information.

One may find, at stage A or B, that a programming error has been made and that the instruction fetched from the store is incorrect in some way. If this happens, touch the NORM RESET key and make the necessary corrections to the program as explained in Section 5. When the A or B key is touched again, the corrected instruction will be read out and the program can proceed.

The RUN key is used when one wishes to run a program at full speed. When this key is touched the machine carries out the instructions one after another and does not stop until an instruction is reached that requires the attention of the operator. This happens under the following four circumstances:

a) An INPUT instruction
The machine will stop and display at stage B of the instruction. The operator must then enter the required data on the keyboard and touch the RUN key again to continue the program.

b) A DISPLAY instruction
The machine will stop at stage C. The operator notes the results and if the program is not yet finished, touches the RUN key again. For the program of Section 9, with $r = 2.5$, the display will be as shown in Figure 14.

c) OVERFLOW
It may be that the result of an arithmetic instruction falls outside the range -999999 to +999999 i.e. an 'overflow' occurs. The machine will then stop, displaying a black rectangle in the stage character position. The contents of the result location will be unaltered.

d) An Invalid instruction
The operator may have included in his program a set of digits that does not correspond to a valid instruction. When the machine tries to obey such an 'instruction' it will stop, showing the black rectangle as in c.

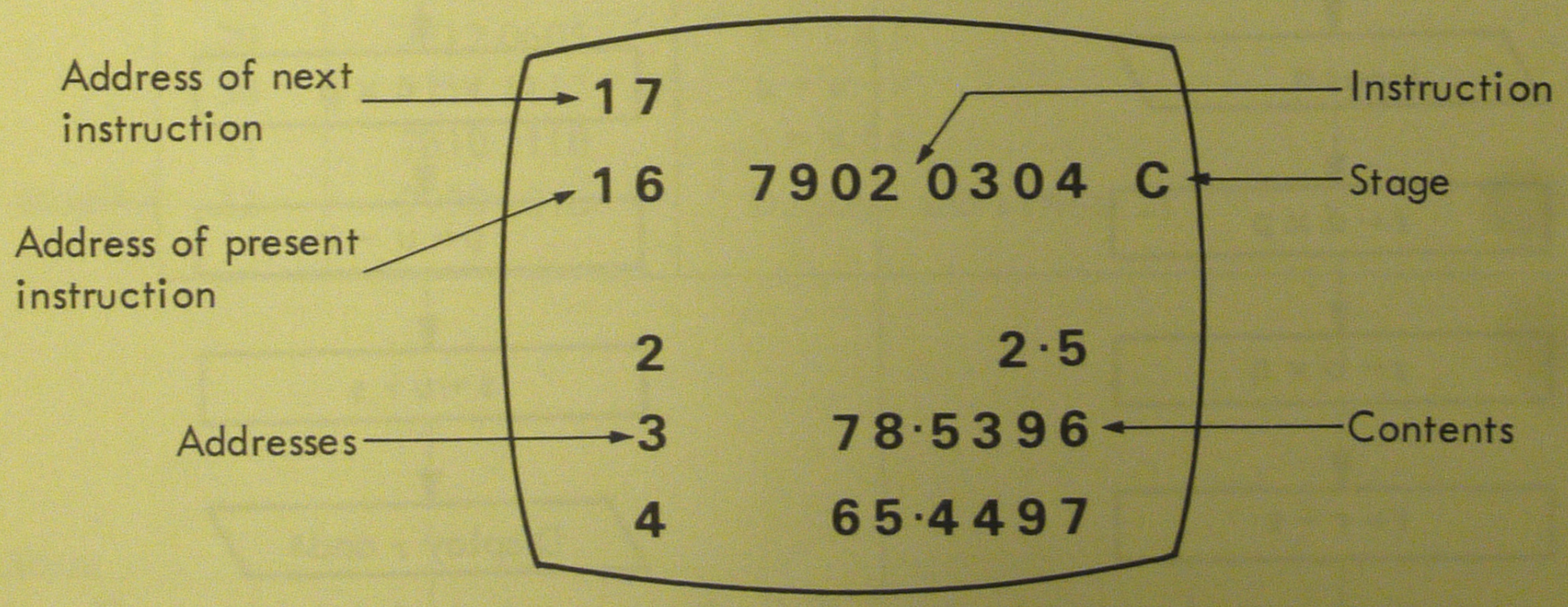


Figure 14. Example of output display

11. SOLUTION OF TWO LINEAR SIMULTANEOUS EQUATIONS

If the equations are of the form

$$ax + by = p$$
$$cx + dy = q$$

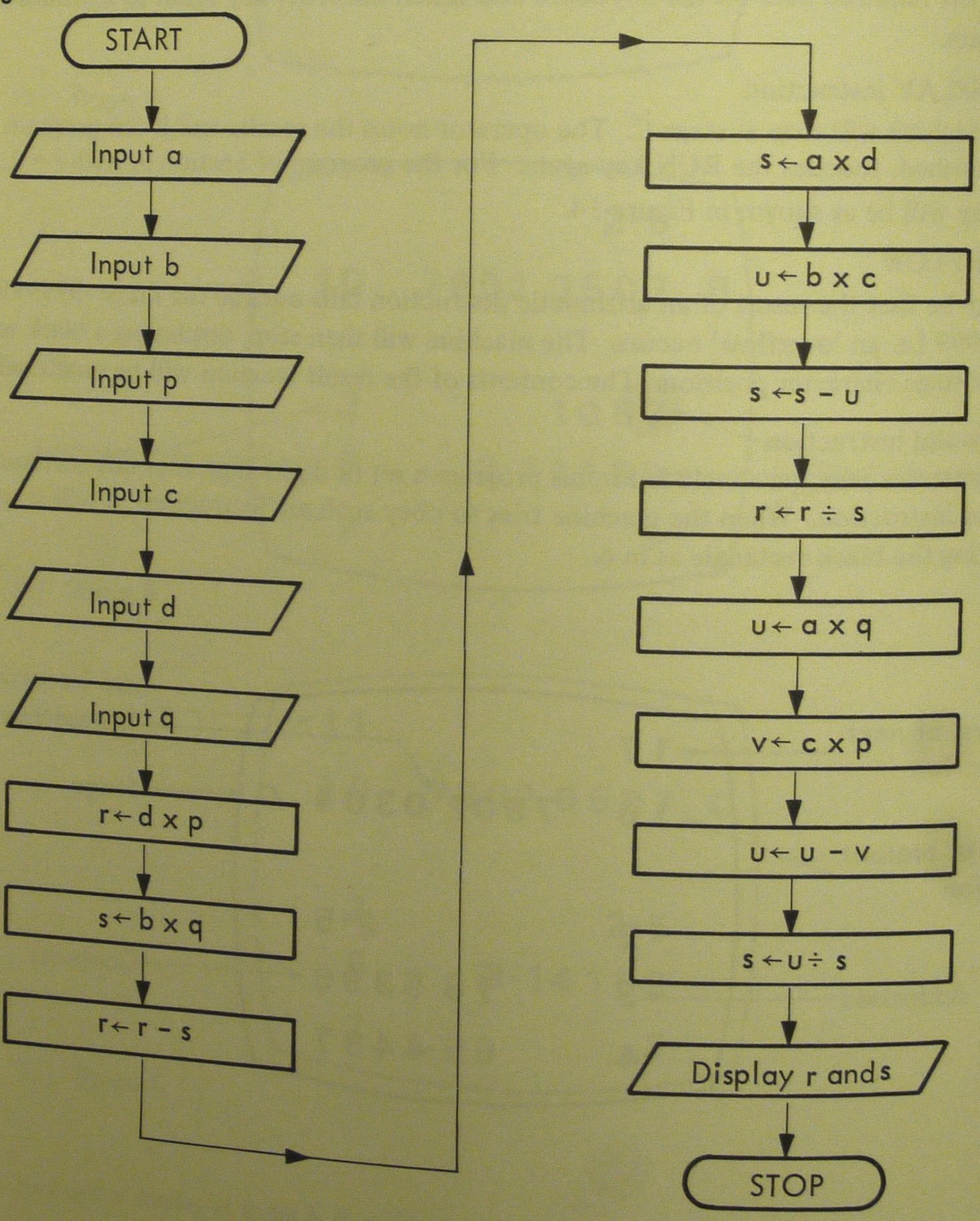
their solution is given by

$$x = \frac{dp - bq}{ad - bc}$$

$$y = \frac{aq - cp}{ad - bc}$$

In the following flow diagram, r is the contents of the location that will eventually contain the value of x, and s is the contents of the location that will eventually contain the value of y. u and v are the contents of two other locations used for storing interim results.

Flow Diagram



Note that in this program the data consists only of variables - there are no constants.

Program

	Address	Contents	Explanation
	0	-	} Spare
	1	-	
	2	-	
	3	*a*	} Variables
	4	*b*	
	5	*p*	
	6	*c*	
	7	*d*	
	8	*q*	
	9	*r*	
	10	*s*	
	11	*u*	
	12	*v*	
	13	-	} Spare
	14	-	
Start →	15	7803 0303	Input *a*
	16	7804 0303	Input *b* (Note 1)
	17	7805 0403	Input *p*
	18	7806 0504	Input *c*
	19	7807 0605	Input *d*
	20	7808 0706	Input *q*
	21	7209 0705	$r \leftarrow d \times p$
	22	7210 0408	$s \leftarrow b \times q$
	23	7109 0910	$r \leftarrow r - s$
	24	7210 0307	$s \leftarrow a \times d$
	25	7211 0406	$u \leftarrow b \times c$
	26	7110 1011	$s \leftarrow s - u$
	27	7309 0910	$r \leftarrow r \div s$
	28	7211 0308	$u \leftarrow a \times q$
	29	7212 0605	$v \leftarrow c \times p$
	30	7111 1112	$u \leftarrow u - v$
	31	7310 1110	$s \leftarrow u \div s$
Stop ←	32	7909 1010	Display *r* and *s* (Note 2)

Notes

1. The second and third addresses in the INPUT instructions can be used to check the variables entered previously.
2. As we are only interested in two quantities, but are obliged to display three, any address may be inserted here to complete the instruction.

12. COMPOUND INTEREST

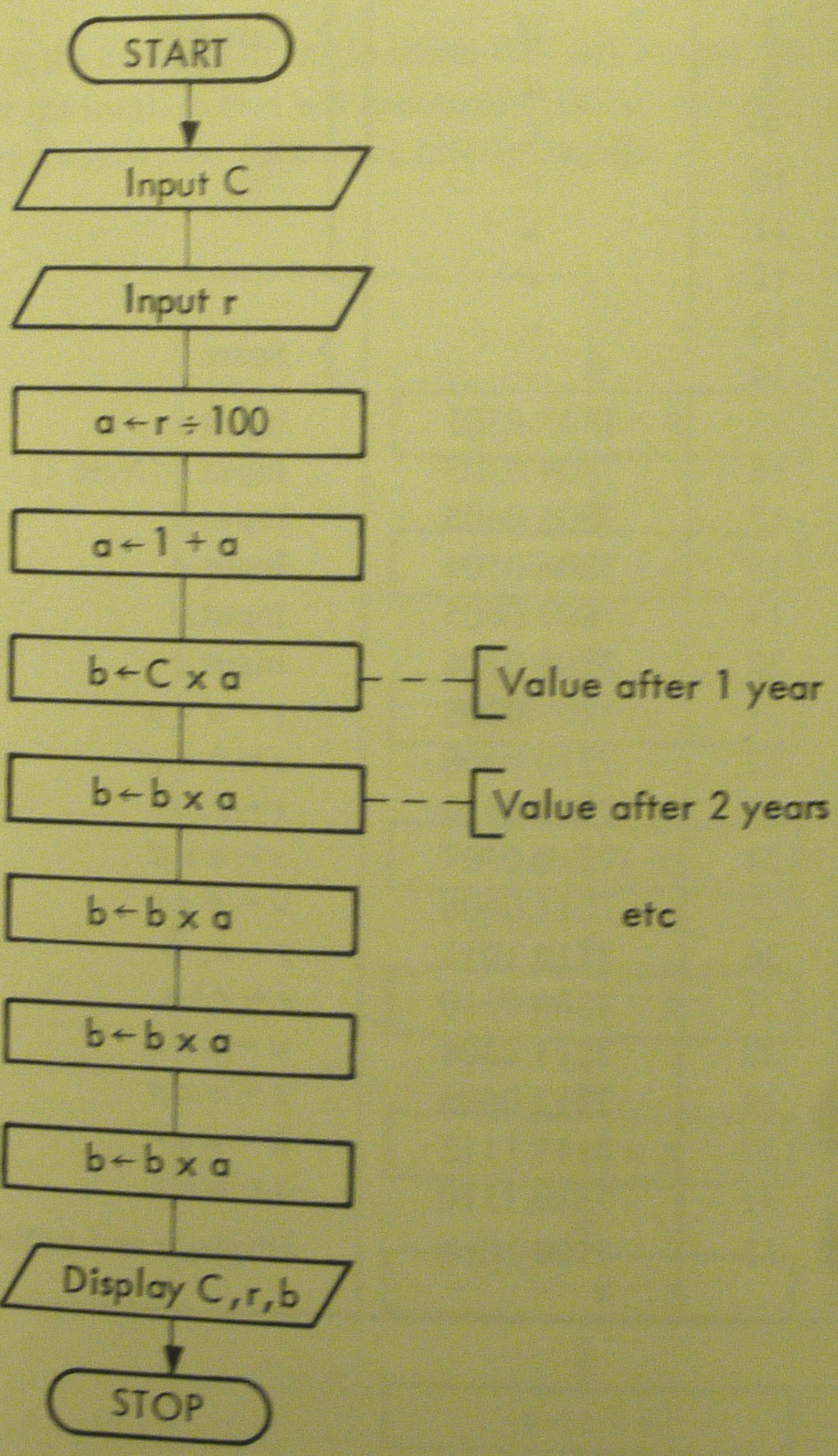
Suppose that a sum of money £C is invested, and interest at r % is added annually. We wish to know the value of the investment after 5 years.

The value will be

$$£C\left(1+\frac{r}{100}\right)^5$$

and the flow diagram for computing it is as follows:

Flow Diagram



Without going to the trouble of coding this program, two features can be clearly seen:

a) The same instruction, $b \leftarrow b \times a$, occurs repeatedly.

b) The number of times it recurs depends on the number of years in question. This means that if we were interested in finding the value after several different numbers of years we

would have to write a new program for each case, and if the number of years were large the program would take up a great deal of space in the store.

It would obviously be much more convenient if the same instruction $b \leftarrow b \times a$ could be used over and over again, but to do this the machine would have to 'know' when the instruction had been used the necessary number of times. This can be done using the CONDITIONAL BRANCH instruction, discussed in the next section.

13. BRANCH INSTRUCTIONS

So far we have confined ourselves to programs in which the instructions are always taken from successive locations in the store. The purpose of branch instructions is to make the machine break out of this simple sequence. Sometimes we want this to happen only if a particular condition is fulfilled; at other times it must happen unconditionally. Hence we have CONDITIONAL and UNCONDITIONAL BRANCH instructions.

13.1 Conditional Branch Instructions

The Schools Computer has three conditional branch instructions.
Referring to Figure 8 they are:

Operation Digits *a*	*b*	Interpretation
7	4	If (*ef*) = (*gh*), replace the contents of the instruction address register by *cd*. Otherwise add 1 to the contents of this register in the normal way. *
7	5	If (*ef*) > (*gh*), replace the contents of the instruction address register by *cd*. Otherwise add 1 to the contents of this register in the normal way.
7	6	This has the same action as code 75, except that the comparison takes no account of the signs of (*ef*) and (*gh*). The modulus of (*ef*) is compared with the modulus of (*gh*).

* The pairs of letters *cd, ef, gh* represent addresses, the first letter of the pair representing the tens digit and the second letter the units digit. Thus, using brackets as in Section 6, (*ef*) represents the number stored at address *ef*, and similarly for the other pairs of letters.

Replacing the contents of the IAR by *cd* causes the machine to take its next instruction from that address. We say that the program 'branches' to address *cd*.

When the machine carries out a conditional branch instruction it first compares (*ef*) and (*gh*) in the arithmetic unit. This indicates to the control unit whether or not branching is to take place, and if it is, digits *cd* are transferred from the instruction register to the IAR. The additional paths required for this transfer are shown in Figure 15.

Conditional Branch instructions have many important uses. If, for example, a sequence of instructions has to be carried out a definite number of times, say *n* times, one location in the store is initially set to *n* and a second is used as a counter. Each time the sequence is carried out a 1 is added to the count and the contents of the two locations are compared, using a conditional branch instruction. If the count is less than *n*, branching takes place to an earlier point in the program, and the sequence is repeated. This is the method used in Section 14.1 to compute compound interest, and in Section 14.3 to determine the area under a curve. Conditional Branch instructions also play an important part in data-processing applications such as the Electricity Account example given in Section 14.2.

In mathematical work, one of their main uses is in successive approximation. This too involves using the same sequence of instructions over and over again, but instead of stopping after a prescribed number of times, the sequence must be repeated until the latest approximation is equal

to the previous one, to the accuracy of the machine. Examples are given in Section 14.4 where a square root is determined by the Newton-Raphson method, and in Section 14.5 where sin x is computed by summing the terms of a convergent series.

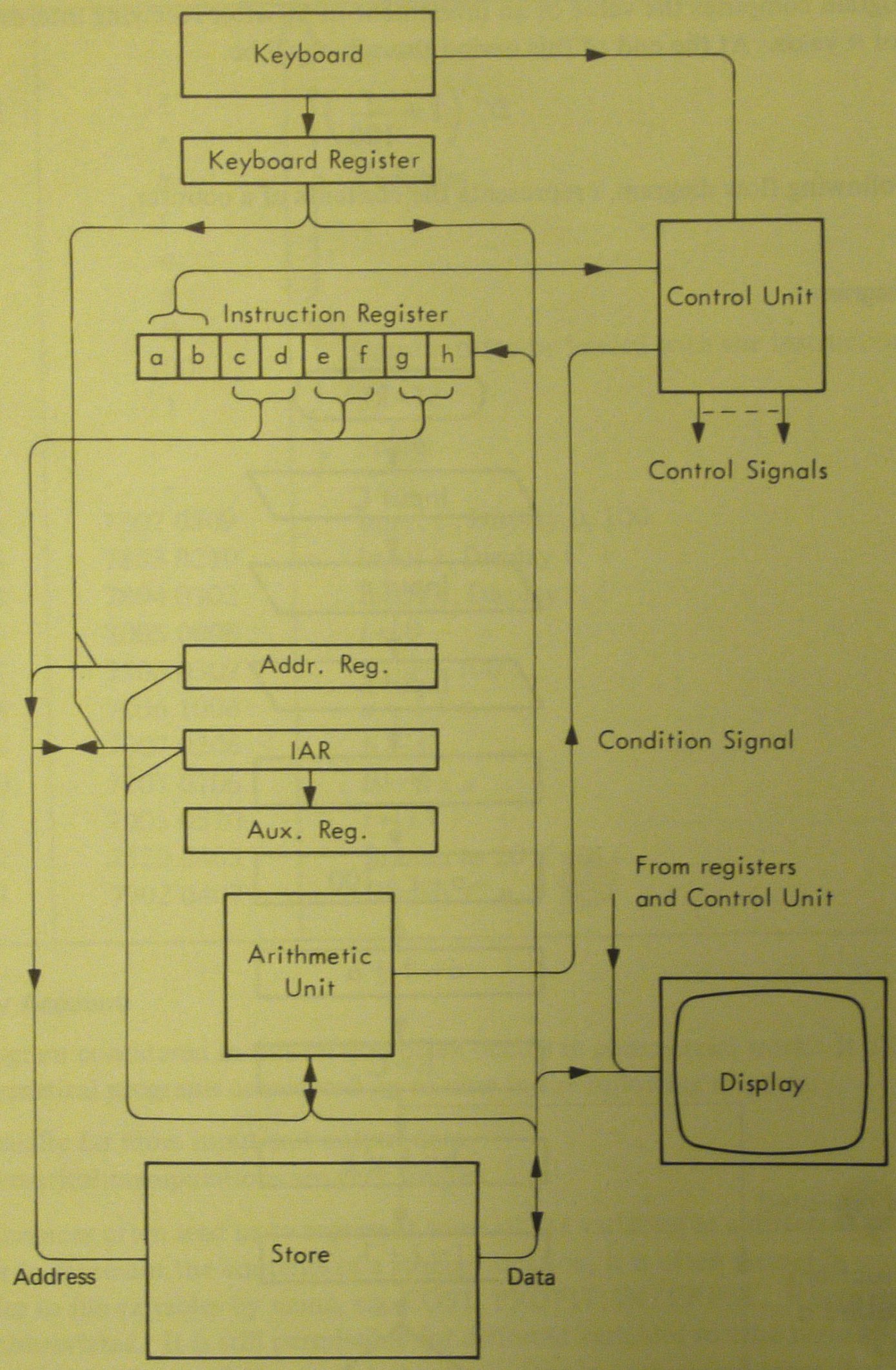


Figure 15. Data-flow diagram showing additional paths used in BRANCH instructions

13.2 Unconditional Branch

In the Schools Computer, unconditional branching can be done by using the 74 instruction (Section 13.1) and making address *gh* the same as *ef*. Any address may be chosen provided it appears in both these positions. The contents of the chosen location will then be tested to see whether it is equal to itself, and since this must be so, branching will occur.

14. APPLICATIONS OF CONDITIONAL BRANCH INSTRUCTIONS

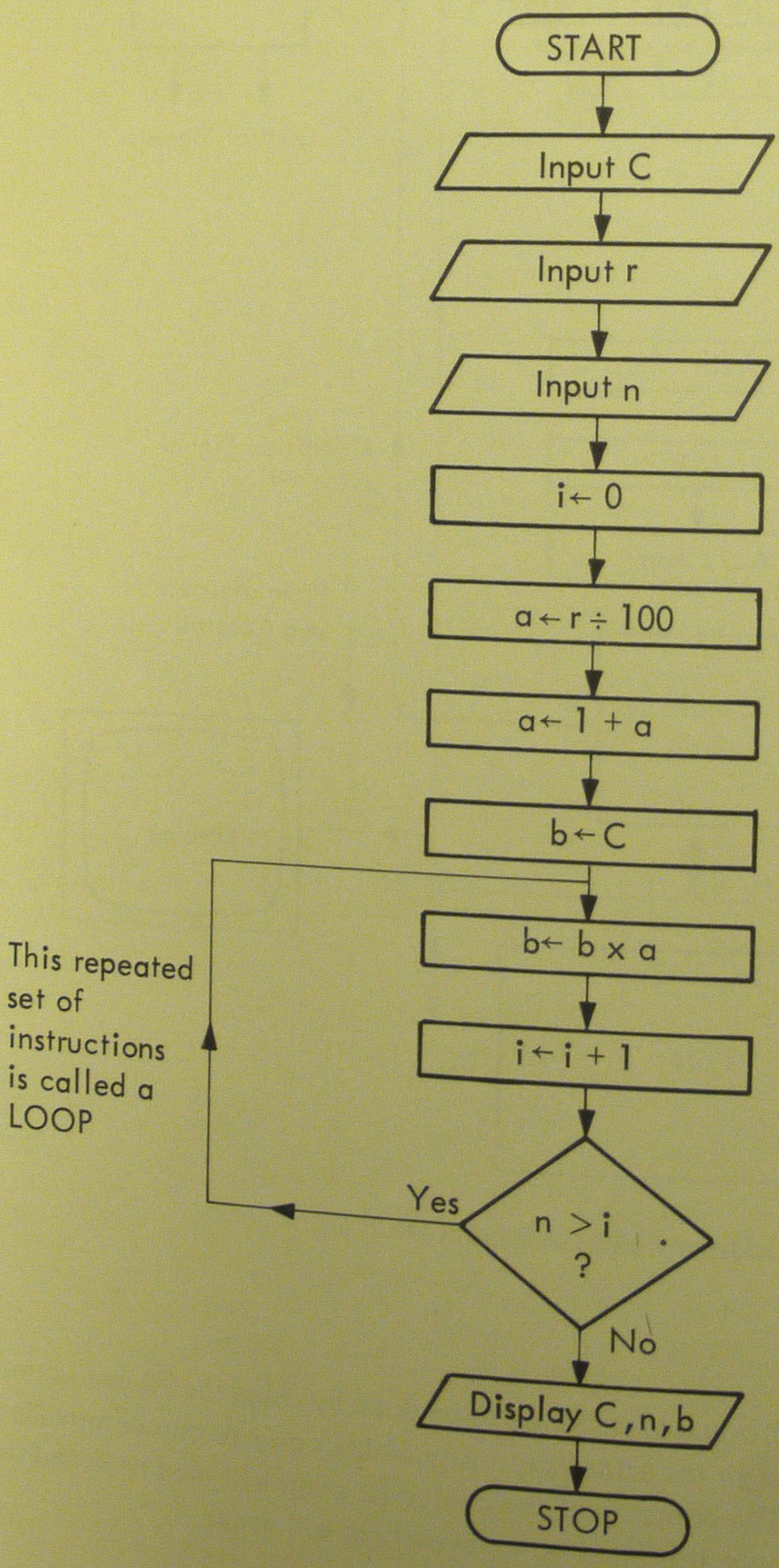
14.1 Compound Interest

This program computes the value of an investment of £C after receiving interest at r % for a period of n years. At the end of this period the value will be:

$$£C\left(1+\frac{r}{100}\right)^{n}$$

In the following flow diagram, i represents the contents of a counter.

Flow Diagram



Program

	Address	Contents	Explanation
	0	-	} Spare
	1	-	
	2	*C*	} Variables
	3	*r*	
	4	*n*	
	5	*i*	
	6	*a*	
	7	*b*	
	8	0	} Constants (to be loaded with the instructions)
	9	100	
	10	1	
	11	-	} Spare
	12	-	
Start →	13	7802 0809	Input *C*; Display 0, 100
	14	7803 0210	Input *r*; Display *C*, 1
	15	7804 0302	Input *n*; Display *r*, *C*
	16	7005 0808	$i \leftarrow 0$
	17	7306 0309	$a \leftarrow r \div 100$
	18	7006 1006	$a \leftarrow 1 + a$
	19	7007 0208	$b \leftarrow C$
↑	20	7207 0706	$b \leftarrow b \times a$
↑	21	7005 0510	$i \leftarrow i + 1$
↑	22	7520 0405	Branch to 20 if $n > i$
Stop ←	23	7902 0407	Display *C*, *n*, *b*

14.2 Electricity Account

The type of program considered in this section often occurs in commercial work. It differs from the more mathematical programs considered up to now in the following respects:

a) There is usually far more input and output data.
b) The mathematical manipulations are very simple.

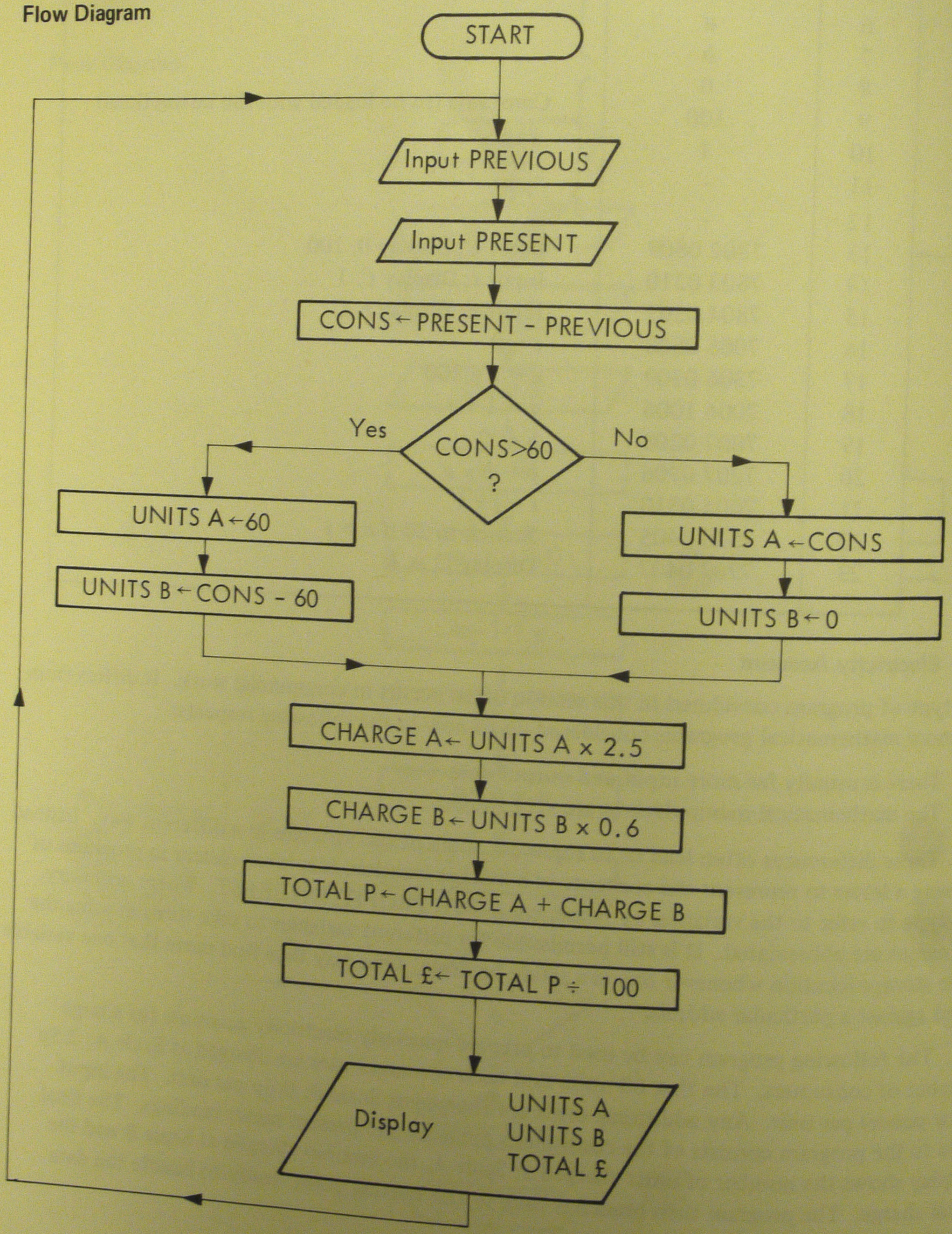
These differences often lead us to represent some of the variables in a different way. Instead of using a letter to represent the contents of a storage location, it is often clearer in programs of this type to refer to the variables by name, e.g. COST, PROFIT, INTEREST. Where necessary, the names are abbreviated. It is still permissible for different variables to take turns in using the same storage location whenever this can safely be done. One may thus find more that one variable listed against a particular address.

The following program can be used to prepare quarterly electricity accounts for a large number of consumers. The first 60 units that each consumer uses are charged at Scale A: 2.5p (new pence) per unit. Any additional units are charged at Scale B: 0.6p per unit. The input data to the program consists of the consumer's previous and present meter readings. The final display shows the number of units charged at Scale A, the number charged at Scale B and the total charge. The program then branches back to the starting point, ready to handle the data relating to the next consumer.

The following abbreviations are used:

Previous meter reading	PREVIOUS	Charge for units at Scale A	CHARGE A
Present meter reading	PRESENT	Charge for units at Scale B	CHARGE B
Total consumption	CONS	Total charge in new pence	TOTAL P
Number of units to be charged at Scale A	UNITS A	Total charge in pounds	TOTAL £
Number of units to be charged at Scale B	UNITS B		

Flow Diagram



Program

Address	Contents	Explanation	
0	-		
1	-		
2	PREVIOUS	Previous meter reading (units)	Variables
	CONS	Total consumption (units)	
	UNITS B	No. of units charged at Scale B	
3	PRESENT	Present meter reading (units)	
	UNITS A	No. of units charged at Scale A	
4	CHARGE A	Charge for units at Scale A	
	TOTAL P	Total charge in new pence	
	TOTAL £	Total charge in pounds	
5	CHARGE B	Charge for units at Scale B	
6	60	Max. No. of units charged at Scale A	Constants
7	0		
8	2.5	Charge per unit at Scale A	
9	0.6	Charge per unit at Scale B	
10	100		
11	-		
12	-		
Start → 13	7802 0610	Input PREVIOUS ; Display 60, 100	
14	7803 0207	Input PRESENT ; Display PREVIOUS, 0	
15	7102 0302	CONS ← PRESENT – PREVIOUS	
16	7520 0206	Branch to 20 if CONS > 60	
17	7003 0207	UNITS A ← CONS	
18	7002 0707	UNITS B ← 0	
19	7422 0707	Unconditional branch to 22	
20	7003 0607	UNITS A ← 60	
21	7102 0206	UNITS B ← CONS - 60	
22	7204 0308	CHARGE A ← UNITS A × 2.5	
23	7205 0209	CHARGE B ← UNITS B × 0.6	
24	7004 0405	TOTAL P ← CHARGE A + CHARGE B	
25	7304 0410	TOTAL £ ← TOTAL P ÷ 100	
26	7903 0204	Display UNITS A, UNITS B, TOTAL £	
27	7413 0707	Unconditional branch to 13	

14.3 Area Under a Curve

The area under any curve, such as that shown in Figure 16, can be found to a good degree of accuracy by dividing it into strips and approximating each strip by a trapezium (See School Mathematics Project Book 5 p10). If there are n strips, each of width d, the area, a, is given by:

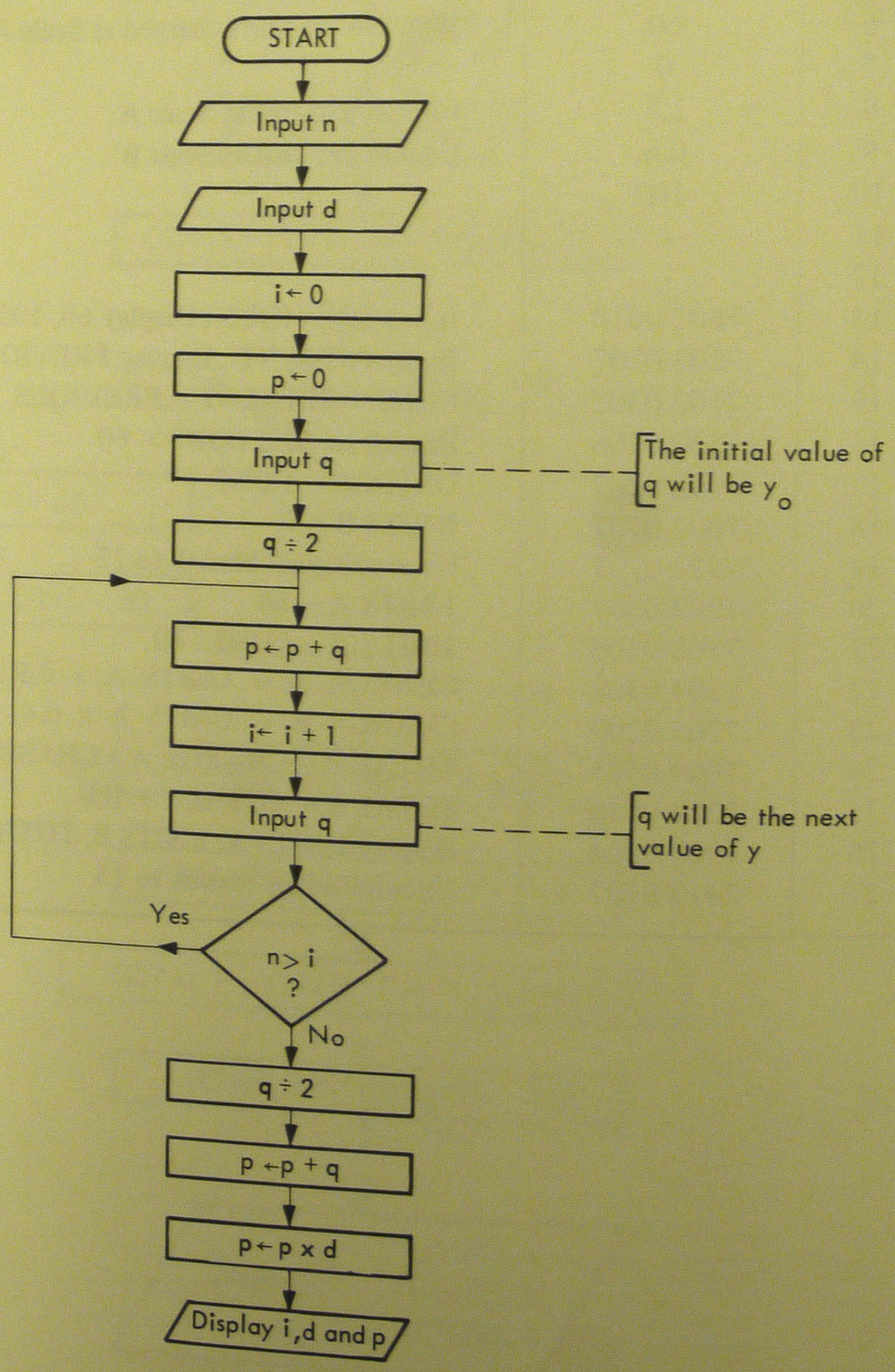
$$a = d\left\{\frac{y_0}{2} + y_1 + y_2 + \ldots\ldots + y_i + \ldots\ldots + y_{n-1} + \frac{y_n}{2}\right\}$$

Flow Diagram

i is the number of strips so far considered.

p is first used for the running total of the quantity in the brackets, then the area.

q is the latest value of y_i to be fed in.



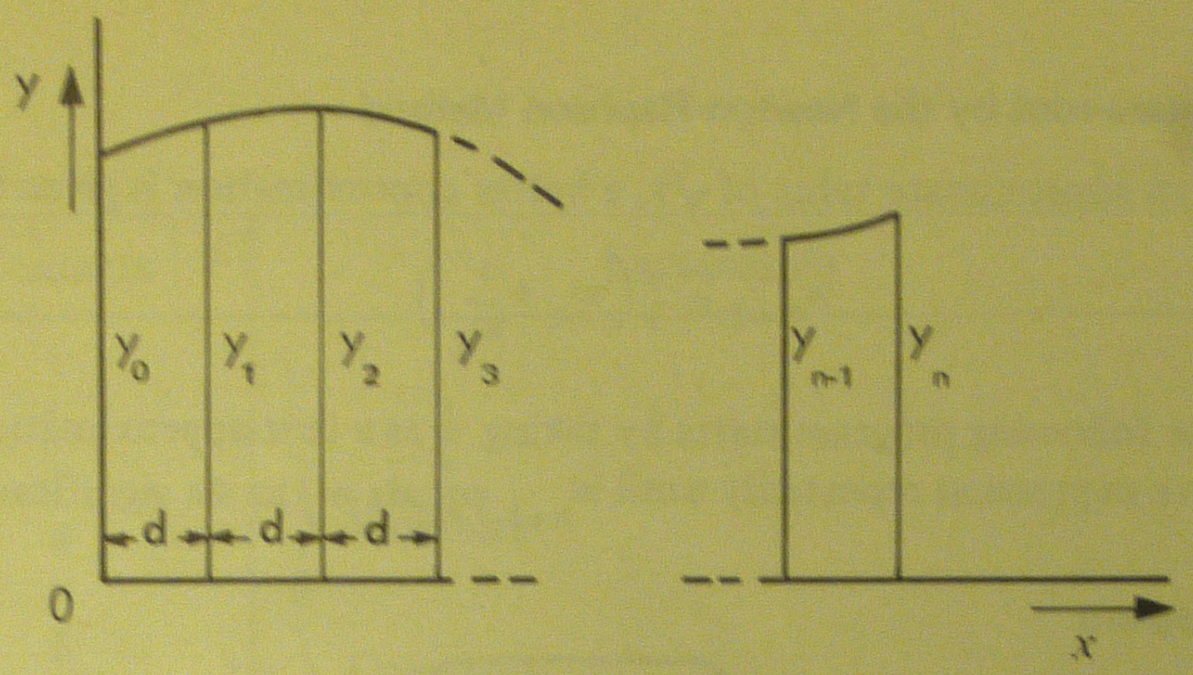


Figure 16. Area Under a Curve

Program

	Address	Contents	Explanation
	0	-	} Spare
	1	-	
	2	*n*	} Variables
	3	*d*	
	4	*i*	
	5	*p*	
	6	*q*	
	7	0	} Constants
	8	1	
	9	2	
	10	-	
	11	-	
Start →	12	7802 0709	Input *n* ; Display 0 and 2
	13	7803 0207	Input *d* ; Display *n* and 0
	14	7104 0404	$i \leftarrow 0$ (Note 1)
	15	7105 0505	$p \leftarrow 0$
	16	7806 0402	Input *q* ; Display *i* and *n* (Note 2)
	17	7306 0609	$q \leftarrow q \div 2$
	18	7005 0506	$p \leftarrow p + q$
	19	7004 0408	$i \leftarrow i + 1$
	20	7806 0402	Input *q* ; Display *i* and *n* (Note 3)
	21	7518 0204	Branch to 18 if $n > i$
	22	7306 0609	$q \leftarrow q \div 2$
	23	7005 0506	$p \leftarrow p + q$
	24	7205 0503	$p \leftarrow p \times d$
Stop ←	25	7904 0305	Display *i*, *d*, *p*

Notes

1) The SUBTRACT instruction is often a convenient way of setting a variable to 0. The variable is subtracted from itself.
2) This INPUT instruction gives *q* the value y_0.
3) This INPUT instruction gives *q* the value y_i and displays the current value of *i* together with *n*. By such a choice of display the operator can see what stage he has reached during a long input procedure.

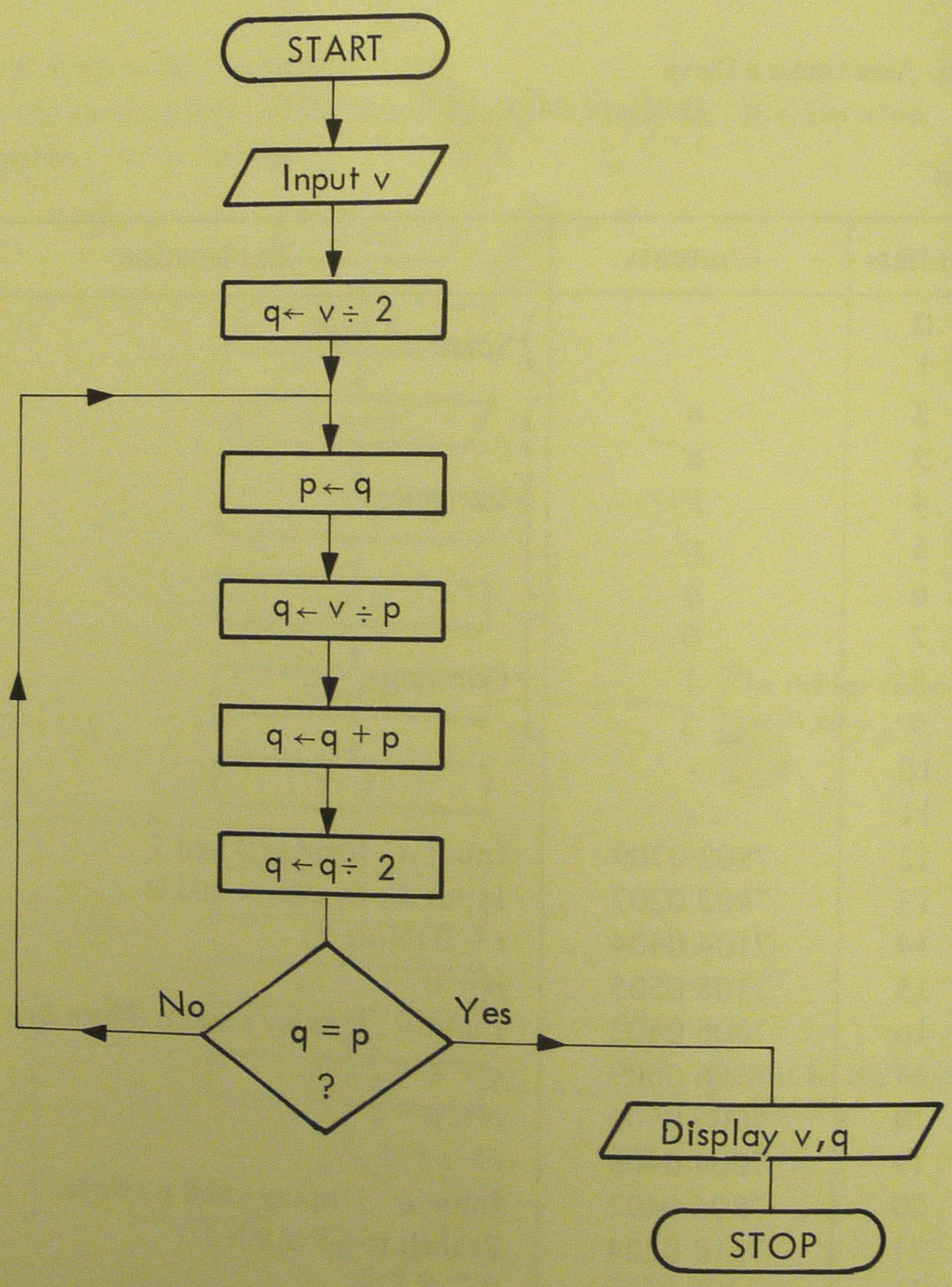
14.4 Square-root by the Newton-Raphson Method

If w_n is an approximate value of $\sqrt{v}$, a better approximation is given by:

$$w_{n+1} = \frac{1}{2}\left(w_n + \frac{v}{w_n}\right)$$

The following program starts by taking $\frac{v}{2}$ as a first approximation to $\sqrt{v}$. It then applies the above expression repeatedly until w_{n+1} equals w_n to six significant figures.

Flow Diagram



Program

	Address	Contents	Explanation
	0	-	
	1	-	
	2	*v*	Variables
	3	*q*	
	4	*p*	
	5	2	Constants
	6	0	
	7	-	
Start →	8	7802 0506	Input *v*; Display 2, 0
	9	7303 0205	$q \leftarrow v \div 2$
	10	7004 0306	$p \leftarrow q$
	11	7303 0204	$q \leftarrow v \div p$
	12	7003 0304	$q \leftarrow q + p$
	13	7303 0305	$q \leftarrow q \div 2$
	14	7416 0304	Branch to 16 if $q = p$
	15	7410 0505	Unconditional branch to 10
Stop ←	16	7902 0304	Display *v*, *q*, *p*.

Sometimes in programs of this kind, the latest and previous approximations never reach equality owing to round-off errors. Each oscillates between two values which differ by 1 in the least significant digit. When there is a risk of this happening, the two approximations should not be tested for equality, but their difference should be tested to see if it is less than some small specified value.

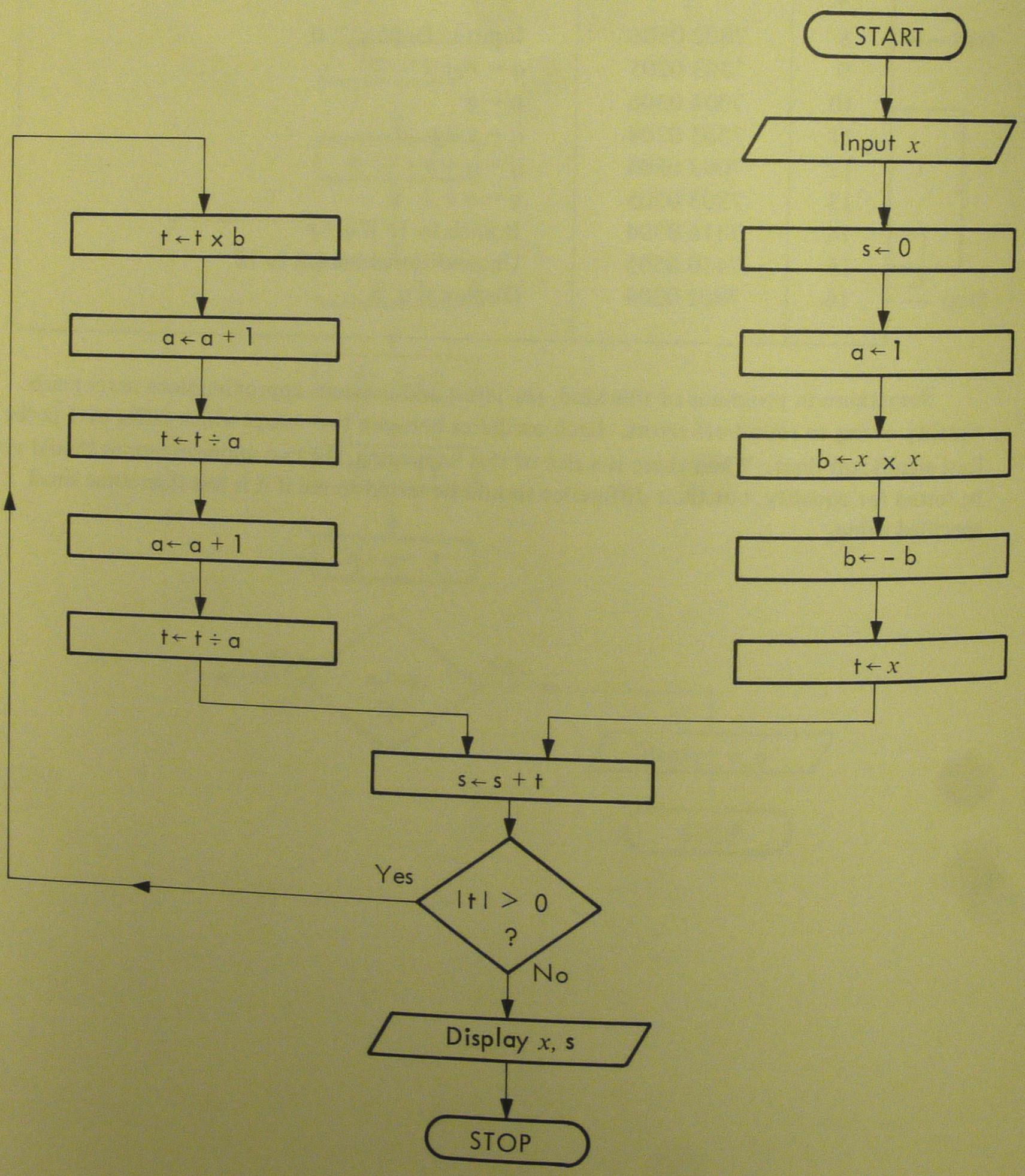
14.5 Sin x by Summation of a Series

The series for sin x is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$$

The following program computes each term of the series by carrying out a fixed sequence of operations on the previous term. When the latest term is found to be zero, to the accuracy of the machine, the loop is terminated and the latest value of the sum, s, is displayed.

Flow Diagram



Program

	Address	Contents	Explanation
	0	-	
	1	-	
	2	x	} Variables
	3	s	
	4	a	
	5	b	
	6	t	
	7	0	} Constants
	8	1	
	9	-	
Start →	10	7802 0708	Input x ; Display 0 and 1
	11	7003 0707	$s \leftarrow 0$
	12	7004 0807	$a \leftarrow 1$
	13	7205 0202	$b \leftarrow x \times x$
	14	7105 0705	$b \leftarrow -b$
	15	7006 0207	$t \leftarrow x$
	16	7422 0808	Unconditional branch to 22
	17	7206 0605	$t \leftarrow t \times b$
	18	7004 0408	$a \leftarrow a + 1$
	19	7306 0604	$t \leftarrow t \div a$
	20	7004 0408	$a \leftarrow a + 1$
	21	7306 0604	$t \leftarrow t \div a$
	22	7003 0306	$s \leftarrow s + t$
	23	7617 0607	Branch to 17 if $\|t\| > 0$
Stop ←	24	7902 0307	Display x, s, 0

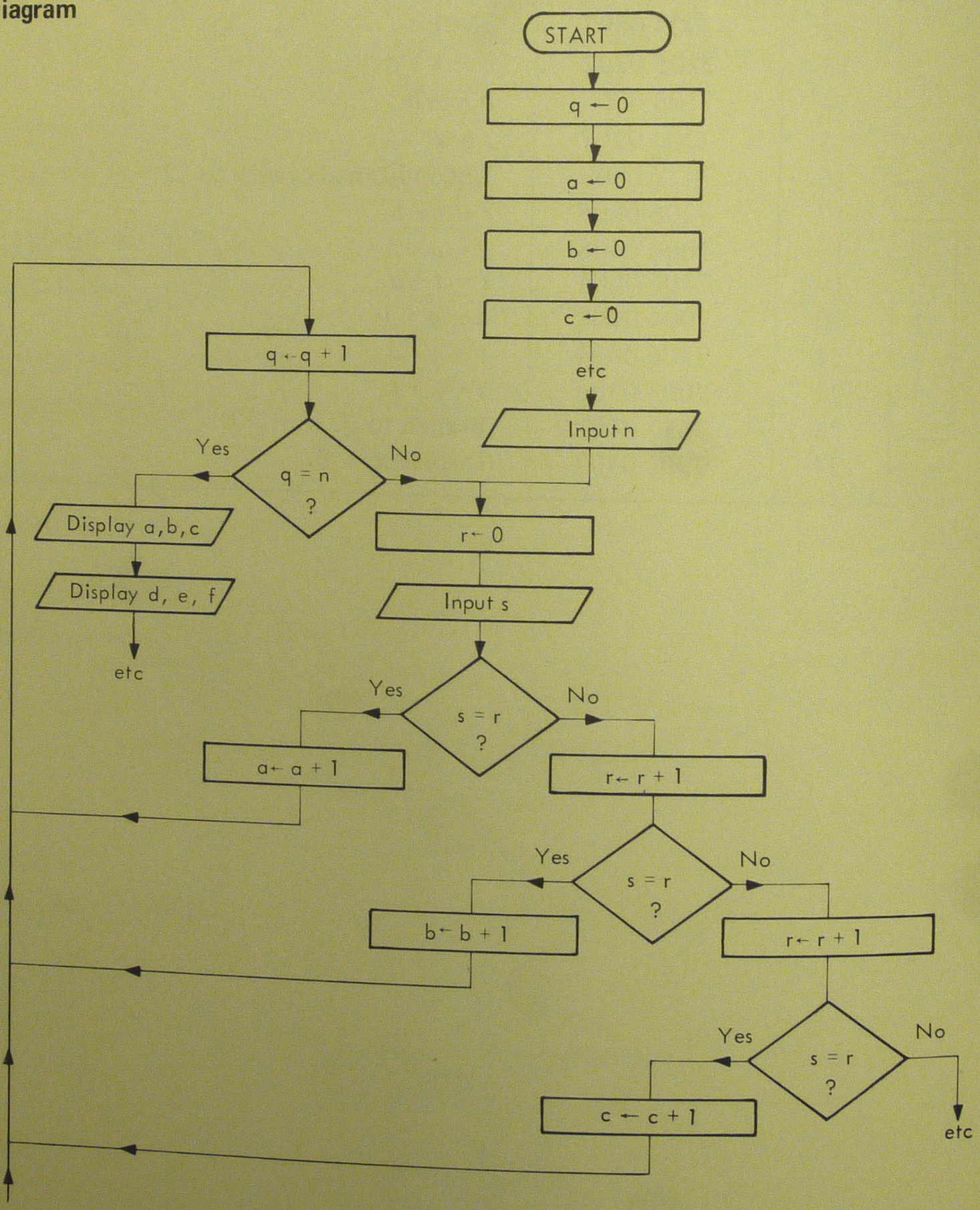
15. A PROGRAM FOR CONSTRUCTING A BAR CHART

Suppose that we are interested in the distance that the children in a particular school have to travel every day from their homes. We can display this information very simply and clearly by means of a bar chart (See School Mathematics Project, Book 2 p23).

Let n be the total number of children in the school,
q, the number of children so far considered,
r, the contents of a counter (see Flow Diagram)
s, the distance (to the nearest mile) travelled by the child under consideration ,
a, the number of children so far considered who travel 0 miles,
b, the number of children so far considered who travel 1 mile,
c, the number of children so far considered who travel 2 miles, etc.

Using the instructions we have already met, the flow diagram is as follows:

Flow Diagram



This is a very clumsy program for the following reasons:

a) It has to use a separate instruction to clear each column of the bar chart to zero initially: $a \leftarrow 0, b \leftarrow 0$, etc.

b) It has to use a separate instruction to increment each column of the bar chart: $a \leftarrow a + 1$, $b \leftarrow b + 1$ etc.

c) Each of the increment instructions requires a branch instruction to reach it, and a second branch instruction to return to the common part of the program.

d) Several instructions are needed to display the output information.

There are two powerful techniques for overcoming these drawbacks: RELATIVE ADDRESSING, mentioned in the preface, and INDIRECT ADDRESSING, which is explained in the next section.

16. INDIRECT ADDRESSING

It was explained in Section 6 that every instruction consists of two parts: an operation part and an address part, the address part containing one or more addresses. In all the instructions we have met so far, each set of address digits has represented the address where some required operand or instruction was stored. We can say that the address digits represented the ABSOLUTE or TRUE ADDRESS of that operand or instruction.

When indirect addressing is used this is no longer so. The address digits in an instruction now represent not the true address of the operand but the address at which the true address will be found. Thus, if the address digits in an instruction are 37, and location 37 contains the number 62, then 37 is the INDIRECT ADDRESS of the operand and 62 is its TRUE ADDRESS.

In the Schools Computer, indirect addressing is specified by changing the first digit of the instruction. In the case of the instructions already discussed, the initial 7 is replaced by a 9. Thus, 90 means Add, 91 Subtract, 98 Input and so on. (Compare with Sections 6 and 13).

When indirect addressing is used, two extra address registers are required besides the one shown in Figure 15. The purpose of the three address registers, shown in Figure 17, is to store the true addresses of the operands. After the instruction has been set up in the instruction register and the control unit has recognised that indirect addressing is required, the store is addressed by digits *cd* from the instruction register, and the contents of this location, i.e. (*cd*), are read out into True Address Register 1 (TAR1). The store is then addressed by digits *ef* and (*ef*) is read out into TAR2. Similarly, (*gh*) is read out into TAR3. The instruction is then carried out in the normal way, the store being addressed by the true address registers to obtain the operands. If the instruction is a branch instruction and the branching condition is fulfilled, the contents of TAR1 are taken to the instruction address register.

In describing how individual instructions can be followed on the display (Section 10), it was mentioned that with certain instructions additional information is displayed at stage A. The instructions referred to there are the ones using indirect addressing. The purpose of displaying at stage A is to allow the indirect and true addresses to be seen. At stage A the indirect addresses appear at the left of the display, and their contents, i.e. the true addresses, appear at the right. An illustration is shown in Figure 18, in which the instruction held at address 76 is

9046 5103

This is an instruction to add, but the operands are indirectly addressed. 46, 51 and 3 are the indirect addresses and these hold the true addresses which are respectively 83, 15 and 35.

The instruction

9046 5103

is therefore equivalent in this case to

7083 1535

It will cause the contents of locations 15 and 35 to be added and the result placed in location 83.

As another illustration of how indirect addressing is used, consider a program containing several variables, *a*, *b* and *c*, used as counters. In such a program we need to be able to add 1 to any of these variables and the technique of indirect addressing can be used to determine which of the variables it is to be.

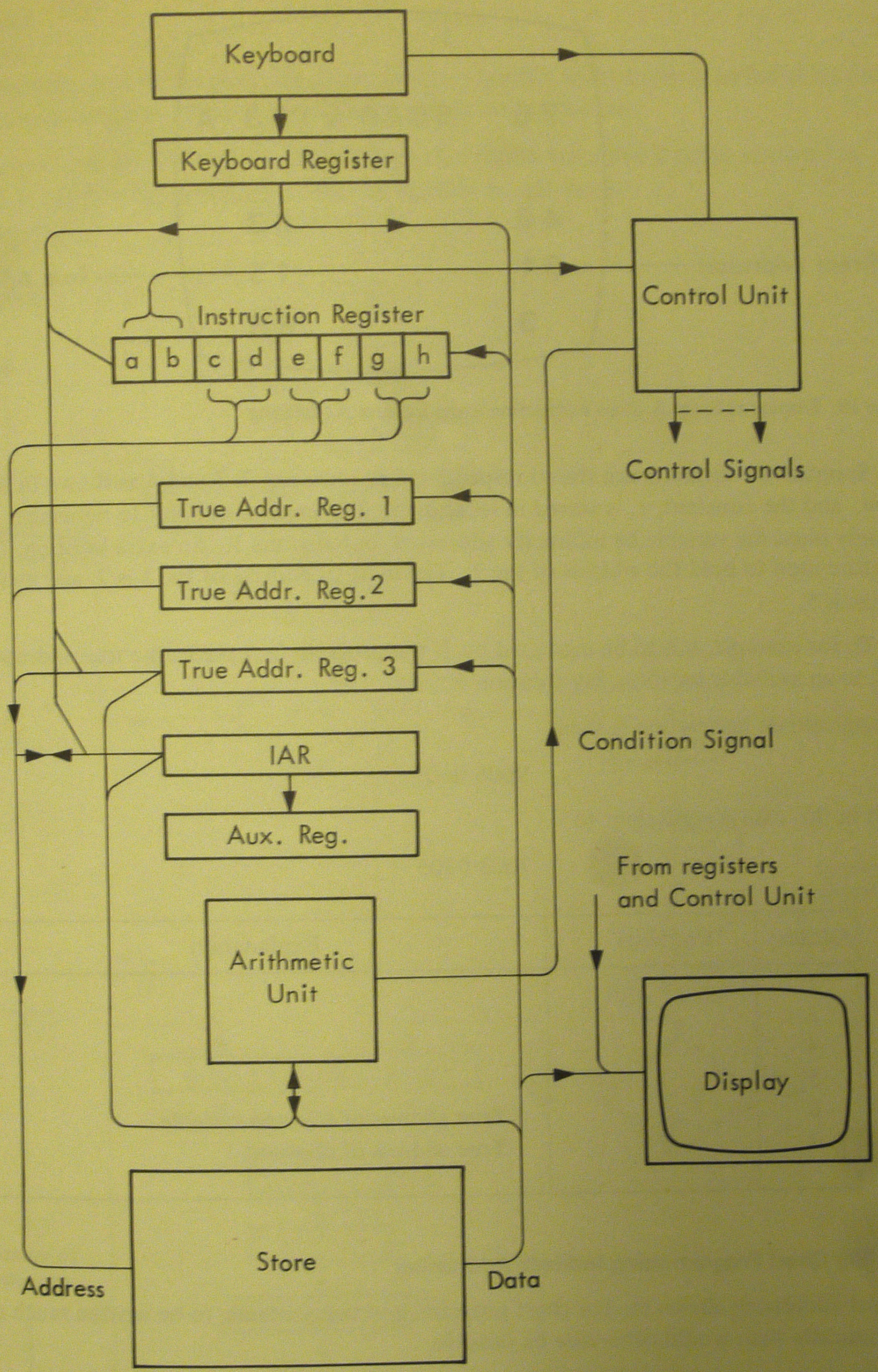


Figure 17. Final Data-flow diagram

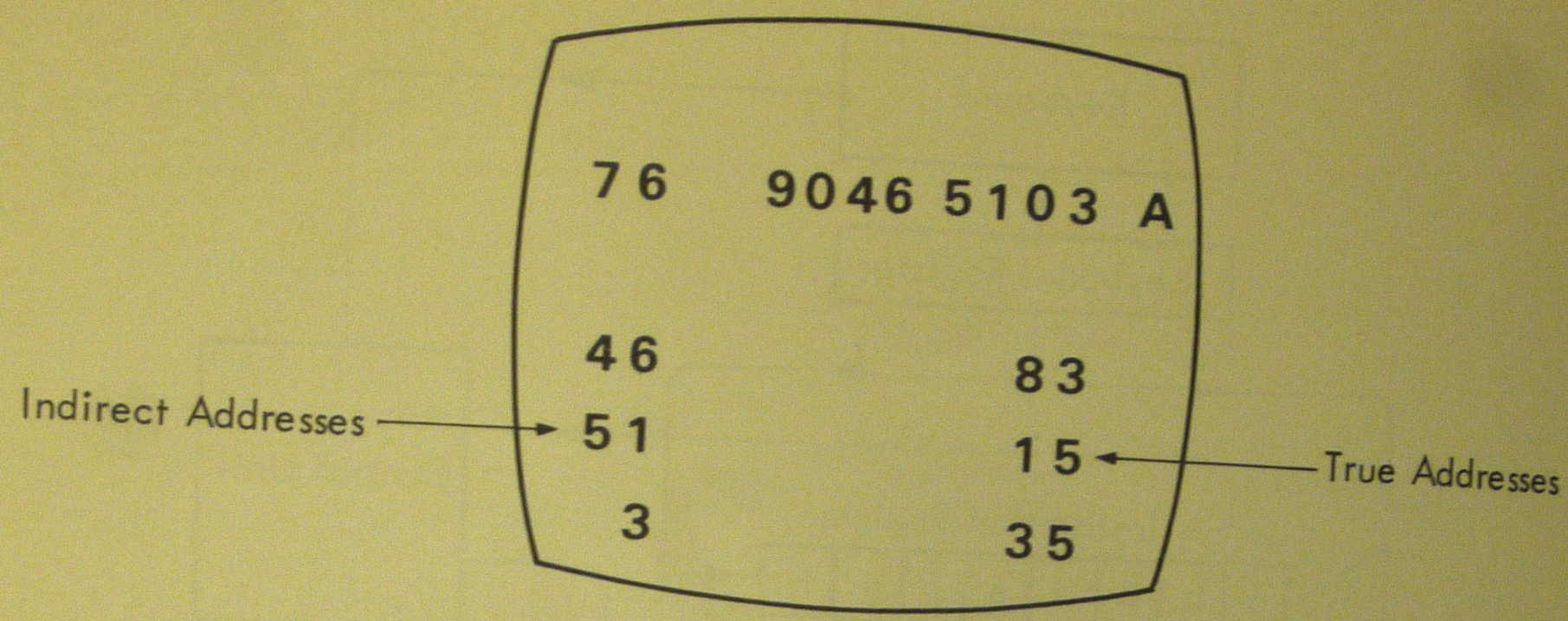


Figure 18. Display at Stage A of an Instruction using Indirect Addressing

Suppose the variables are stored respectively at addresses 2, 3 and 4 as shown in the table below, and the constant, 1, is stored in location 5. The important point to remember is that not only must the variable be indirectly addressed, but also the 1. An extra location, 7, is therefore used to hold the address of the 1. The indirect address of the 1 is 7, and its true address is 5.

If, for example, *b* is to be increased by 1, the program first places the true address of *b* i.e. 3, in an available location, say location 6.

The appropriate instruction is then

9006 0607

which in this case is equivalent to

7003 0305

Address	Contents	Explanation
2	*a*	
3	*b*	
4	*c*	
5	1	
6	3	True address of selected variable
7	5	True address of constant 1

16.1 Bar Chart Program using Indirect Addressing

Indirect addressing allows the bar chart program, and many others, to be written much more economically than would otherwise be possible.

The technique is used first of all to clear all the columns of the bar chart to zero, and then, as each mileage figure is entered, to add 1 to the appropriate column. After the final mileage figure has been entered, indirect addressing is used to give an economical display routine.

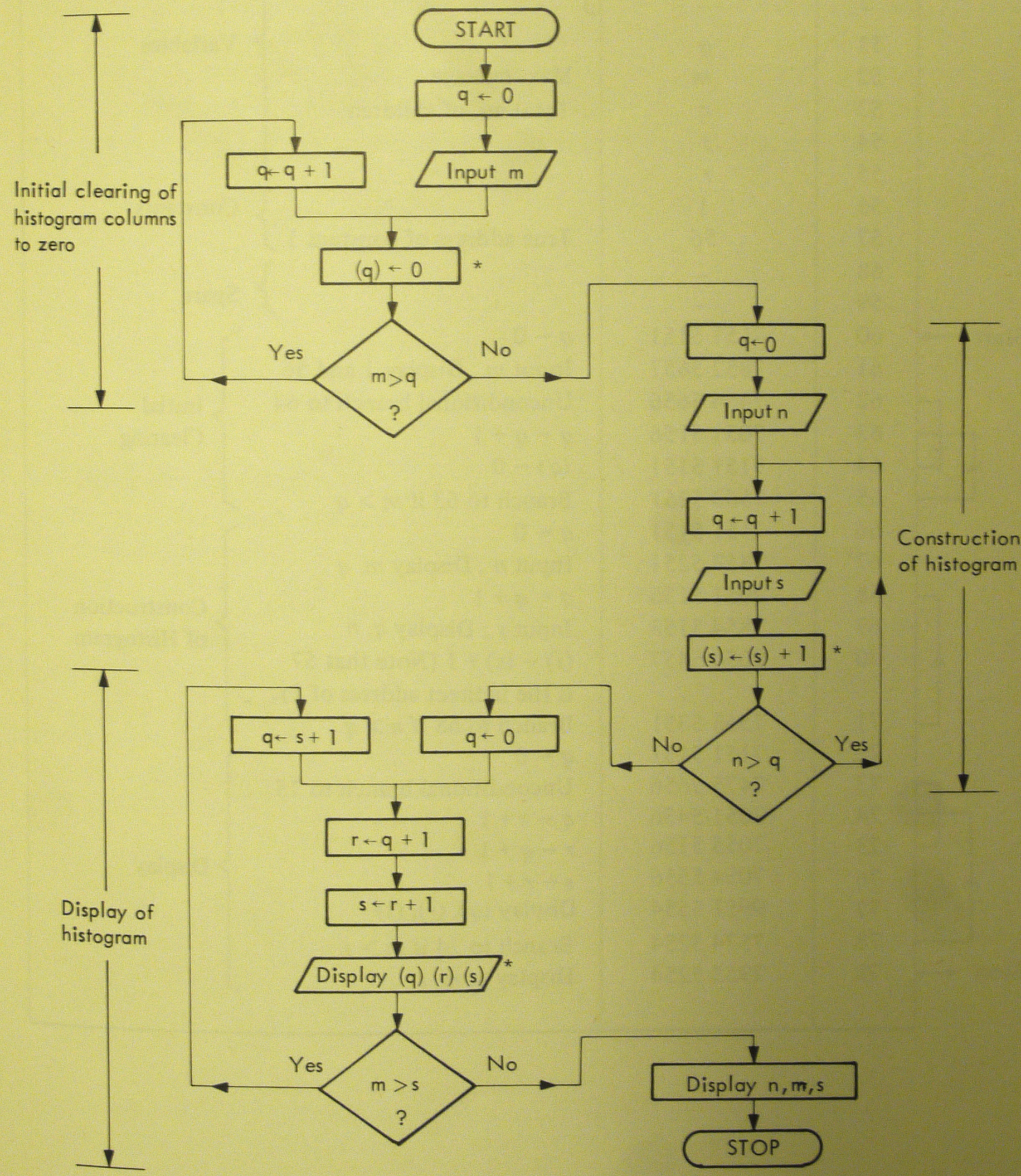
In the following flow diagram:

m = maximum mileage to be allowed for in the program
n = total number of children.

q is a counter
r is a counter
s, during the main part of the program, is the distance (to the nearest mile), travelled by the child under consideration. During the display routine it is a counter.

To keep the program simple, *a*, the number of children travelling 0 miles, is stored in location 0, *b*, the number travelling 1 mile, in location 1, and so on.

Flow Diagram



* These instructions use indirect addressing

Program (This will cater for a maximum distance of 50 miles)

	Address	Contents	Explanation	
	0	*a*		Variables
	1	*b*		
	2	*c*	Bar chart Columns	
	3	*d*		
	–	–		
	–	–		
	–	–		
	51	*q*		
	52	*m*	Max. mileage	
	53	*n*	Total no. of children	
	54	*s*		
	55	*r*		
	56	1		Constants
	57	56	True address of constant 1	
	58	–		Spare
	59	–		
Start →	60	7151 5151	$q \leftarrow 0$	Initial Clearing
	61	7852 5657	Input m ; Display 1 and 56	
	62	7464 5656	Unconditional branch to 64	
	63	7051 5156	$q \leftarrow q + 1$	
	64	9151 5151	$(q) \leftarrow 0$	
	65	7563 5251	Branch to 63 if $m > q$	
	66	7151 5151	$q \leftarrow 0$	Construction of Histogram
	67	7853 5251	Input n ; Display m, q	
	68	7051 5156	$q \leftarrow q + 1$	
	69	7854 5153	Input s ; Display q, n	
	70	9054 5457	$(s) \leftarrow (s) + 1$ (Note that 57 is the indirect address of 1).	
	71	7568 5351	Branch to 68 if $n > q$	
	72	7151 5151	$q \leftarrow 0$	Display
	73	7475 5656	Unconditional branch to 75	
	74	7051 5456	$q \leftarrow s + 1$	
	75	7055 5156	$r \leftarrow q + 1$	
	76	7054 5556	$s \leftarrow r + 1$	
	77	9951 5554	Display (q), (r), (s)	
	78	7574 5254	Branch to 74 if $m > s$	
Stop ←	79	7953 5254	Display n, m, s	

The economy of this program, compared with the one in Section 14, can be clearly seen. Even if there were only twelve columns to the bar chart, the program in Section 14 would take 60 instructions; the present one takes only 20, no matter how many columns there may be.

It is worth pointing out that a loop containing an INPUT instruction with indirect addressing is a good way of loading several items of data. The method is often used in larger machines where the data is read in from paper tape or punched cards. (See Appendix C).

17. SUBROUTINES

So far we have confined ourselves to very simple programs but in practice, programs are often long and complex. They may contain thousands of instructions and have many loops and branches.

The task of writing and checking such a program is enormous, but there are various ways in which it can be made easier. One way is to break it down into sections so that several programmers can work on it at once. The individual sections are often known as ROUTINES and are so arranged that each carries out some specific task.

Another way is to use, whenever possible, routines that already exist. For instance, if we are writing a program which at some stage requires the extraction of a square root, we can use the routine from Section 14.4. When, as in this case, a routine can be used in many different programs, or at several places in the same program it is called a SUBROUTINE.

The following examples show how subroutines are used.

17.1 Solution of a Quadratic Equation

We shall solve the equation

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

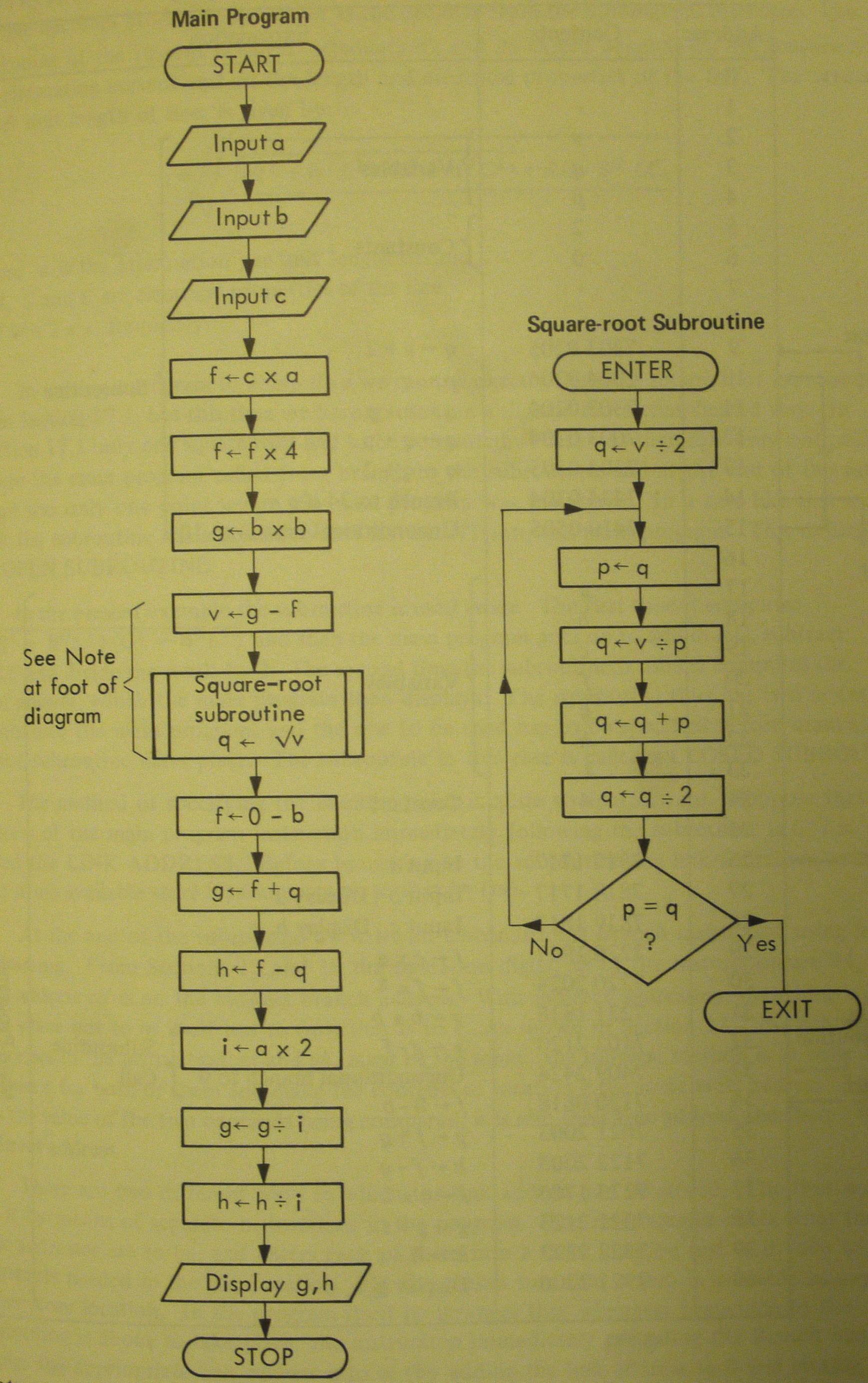
restricting ourselves for simplicity to the case where the roots are real. The solution is:

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

The program first computes $b^2 - 4ac$ and then uses this as input data for the square root program of Section 14.4, used here as a subroutine. Obviously we do not want to alter this subroutine each time we use it in a different program. This means storing it in the addresses already allocated, i.e. 2 to 16, and reaching it by means of an Unconditional Branch instruction. Slight alterations, however, must be made: the 'Input v' and the 'Display' instructions are no longer needed. In this example they can simply be omitted, and the branch address in the Conditional Branch instruction altered to the address at which the main program is re-entered.

Note
The p
the su

Flow Diagram



Note

The process of assigning a value to *v* and branching to the subroutine is known as 'CALLING the subroutine'.

	Address	Contents	Explanation	
	0	-		
	1	-		
	2	*v*	Variables	
	3	*q*		
	4	*p*		
	5	2	Constants	
	6	0		
	7	-		
	8	-		
Enter →	9	7303 0205	$q \leftarrow v \div 2$	Subroutine
	10	7004 0306	$p \leftarrow q$	
	11	7303 0204	$q \leftarrow v \div p$	
	12	7003 0304	$q \leftarrow q + p$	
	13	7303 0305	$q \leftarrow q \div 2$	
	14	7434 0304	Branch to 34 if $q = p$	
	15	7410 0505	Unconditional branch to 10	
Exit →	16	-		
	17	*a*	Variables	Main Program
	18	*b*		
	19	*c*		
	20	*f*		
	21	*g*		
	22	*h*		
	23	*i*		
	24	4	Constant	
	25	-		
Start →	26	7817 1717	Input *a*	
	27	7818 1717	Input *b* ; Display *a*	
	28	7819 1817	Input *c* ; Display *b*, *a*	
	29	7220 1917	$f \leftarrow c \times a$	
	30	7220 2024	$f \leftarrow f \times 4$	
To Subroutine →	31	7221 1818	$g \leftarrow b \times b$	Subroutine Call
	32	7102 2120	$v \leftarrow g - f$	
← Return	33	7409 2424	Unconditional branch to 9	
	34	7120 0618	$f \leftarrow 0 - b$	
	35	7021 2003	$g \leftarrow f + q$	
	36	7122 2003	$h \leftarrow f - q$	
	37	7223 1705	$i \leftarrow a \times 2$	
	38	7321 2123	$g \leftarrow g \div i$	
	39	7322 2223	$h \leftarrow h \div i$	
Stop ←	40	7921 2206	Display *g*, *h*, 0	

17.2 Attenuation in a Telephone Line – Open and Closed Subroutines

When an alternating current is applied to one end of a telephone line, the current flowing in the line becomes progressively weaker as the distance from the sending-end increases. This weakening of the current is called 'attenuation', and its extent depends on the frequency of the alternating current and on the length and electrical properties of the line. The attenuation, a, per unit length of line, is given by:

$$a = \sqrt{\frac{\sqrt{(R^2 + W^2L^2)(G^2 + W^2C^2)} + RG - W^2LC}{2}}$$

Where a is the attenuation per unit length of line,
R, L, G and C are electrical properties of the line,
and $W = 2\pi \times$ frequency.

In writing a program to compute a we can use exactly the same square root subroutine as in Section 17.1, but this time we have to use it in a slightly more complicated way. In Section 17.1 only one square root had to be computed. There was therefore only one point where the main program called for a branch to the subroutine, and at the end of the subroutine there was only one point where the main program was re-entered. In a case like this we say that the subroutine has a fixed RE-ENTRY POINT into the main program. This defines it as an OPEN SUBROUTINE.

In the present example the subroutine is used twice. The first time it computes $\sqrt{(R^2 + W^2L^2)(G^2 + W^2L^2)}$, and then the main program goes on to add in RG, subtract W^2LC and divide the result by 2. The second time the subroutine is used it computes the final answer, which the main program then displays. The subroutine thus has two re-entry points into the main program, and the one to be used has to be specified before branching to the subroutine takes place. The subroutine in this case is called a CLOSED SUBROUTINE.

The method of specifying the re-entry point is quite straightforward. Suppose that the address of the main program instruction immediately following the subroutine is l. l is called the LINK ADDRESS. Before branching to the subroutine the link address is written into some available store location, say m, so that $l = (m)$.

At the end of the subroutine we write an Unconditional Branch instruction using indirect addressing. From Sections 13 and 16 the operation digits, ab of the instruction are 94, and the first address cd is m, the indirect branch address. With indirect addressing, branching will take place not to m itself but to (m), that is to l. As shown in Section 13.2 the other two addresses in the instruction, ef and gh, must be the same: the simplest method is to write some address z for both of them and make the contents of location z the same as its address. Then z is the value of the two operands being compared; it is also their true address and their indirect address.

There are two different ways in which the link address can be stored. The first way is to do it by means of separate instructions in the program. The programmer notes what the link addresses are to be, and enters each of them into a store location just as if they were constants needed in the computation. He also makes sure that a zero is available in some other store location. In the program itself he arranges that whenever branching to the subroutine is about to take place, the instruction immediately preceding the Branch instruction copies the appriopriate link address into m (by adding the link address to 0 and storing the result in m). This method needs no instructions beyond those we have already met.

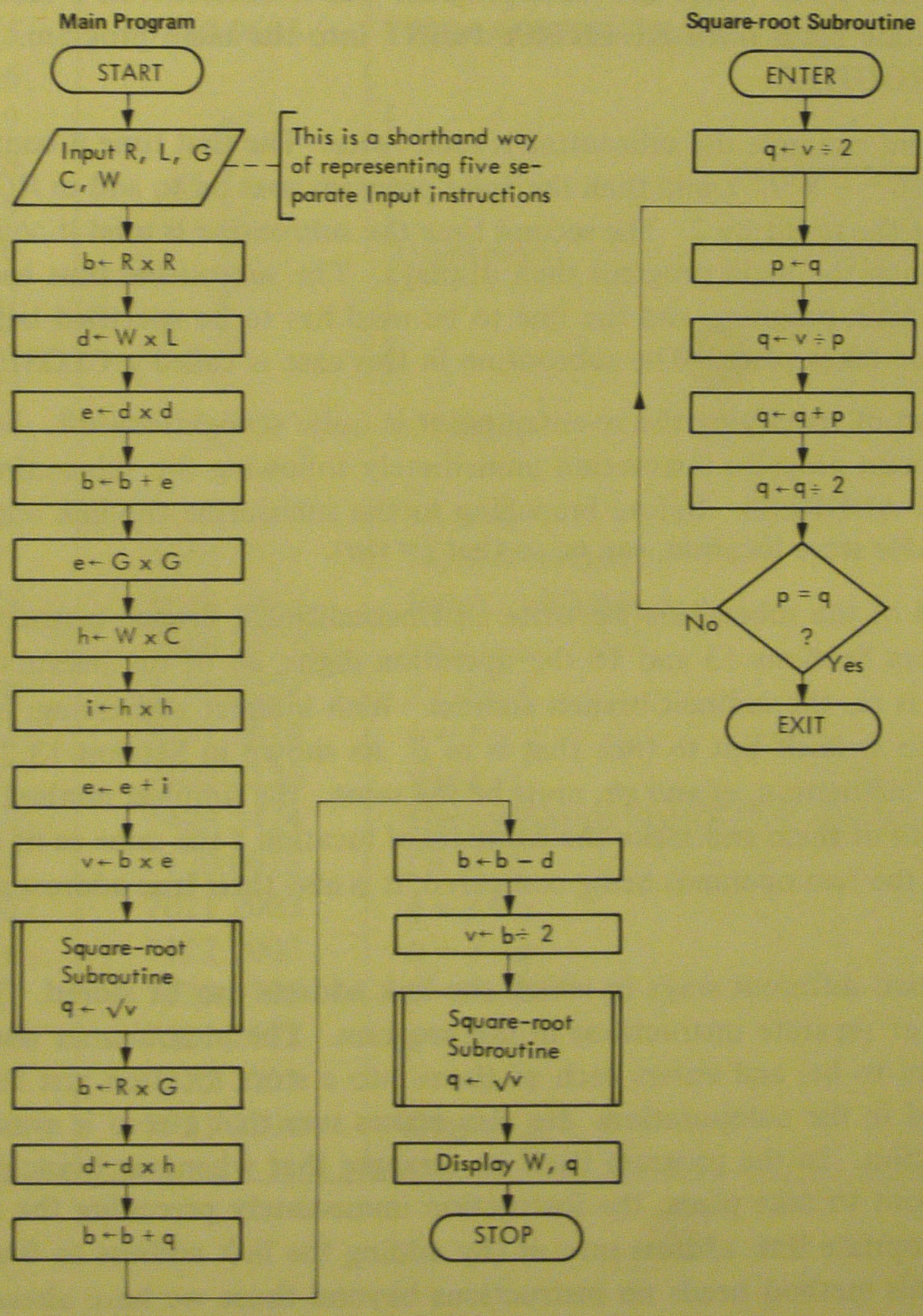
The second way of storing the link address is to use an instruction designed specially for the purpose; it is called the BRANCH AND LINK instruction. This instruction first adds 1 to the IAR in the normal way, to form the link address. It then stores the link address in the required location and causes the appropriate branch to take place. Referring to Figure 8, on page 10, the operation digits and interpretation are as follows:

Operation Digits *a* *b*	Interpretation
7 7	Add 1 to the instruction address register in the normal way. Then if (*gh*) ≠ 0, store (IAR) in location *ef* and branch to *cd*.

(IAR) is the link address, i.e. one greater than the address of the present instruction;
ef is address *m*, the address at which the link address is stored;
cd is the address of the first instruction in the subroutine.

Normally, the conditional property of this instruction is not needed, and so *gh* can be any address whose contents are known to be non-zero.

Flow Diagram



Program

	Address	Contents	Explanation	
	0	-		
	1	Link Address		Subroutine
	2	v		Variables
	3	q		
	4	p		
	5	2		Constants
	6	0		
	7	7	(Note 1)	
Enter	8	-		
	9	7303 0205	$q \leftarrow v \div 2$	
	10	7004 0306	$p \leftarrow q$	
	11	7303 0204	$q \leftarrow v \div p$	
	12	7003 0304	$q \leftarrow q + p$	
	13	7303 0305	$q \leftarrow q \div 2$	
	14	7416 0304	Branch to 16 if $q = p$ (Note 2)	
	15	7410 0505	Unconditional branch to 10	
Exit	16	9401 0707	Unconditional branch to Link Address (Note 1)	
	17	-		
	18	R		Variables
	19	L		
	20	G		
	21	C		
	22	W		
	23	b		
	24	d		
	25	e		
	26	h		
	27	i		
	28	-		
Start →	29	7818 1818	Input R	
	30	7819 1818	Input L ; Display R	
	31	7820 1918	Input G ; Display L, R	
	32	7821 2019	Input C ; Display G, L	
	33	7822 2120	Input W ; Display C, G	
	34	7223 1818	$b \leftarrow R \times R$	
	35	7224 2219	$d \leftarrow W \times L$	
	36	7225 2424	$e \leftarrow d \times d$	
	37	7023 2325	$b \leftarrow b + e$	
	38	7225 2020	$e \leftarrow G \times G$	
	39	7226 2221	$h \leftarrow W \times C$	
To Sub-routine	40	7227 2626	$i \leftarrow h \times h$	
	41	7025 2527	$e \leftarrow e + i$	
	42	7202 2325	$v \leftarrow b \times e$	
Return →	43	7709 0105	Store Link Addr. in location 1; Branch to 9 (Note 3)	
	44	7223 1820	$b \leftarrow R \times G$	
	45	7224 2426	$d \leftarrow d \times h$	

(Continued)

Program (continued)

	Address	Contents	Explanation
To Sub-routine	46	7023 2303	$b \leftarrow b + q$
	47	7123 2324	$b \leftarrow b - d$
	48	7302 2305	$v \leftarrow b \div 2$
	49	7709 0105	Store Link Addr. in location 1; Branch to 9 (Note 4)
Return	50	7922 0306	Display W, q, 0.

Notes

1) The contents of location 7 are made equal to its address. This is a convenient way of providing an indirectly-addressed constant for use in Unconditional Branch instructions. (See the earlier part of this section, and Section 13.2).

2) Apart from returning to the main program, the instruction at address 14 is the last instruction in the subroutine to be obeyed. Instead of first carrying out a conditional branch to 16 and then branching unconditionally to the Link Address, we can make the instruction at address 14 branch straight to the Link Address by specifying indirect addressing. The two quantities to be compared, $q$and p, must then be indirectly addressed too. The following table shows the changes that would be made.

Address	Contents	Explanation
-	-	
-	-	
-	-	
7	3	True address of q
8	4	True address of p
-	-	
-	-	
14	9401 0708	Branch to Link Address if $q = p$
-	-	
-	-	
16	-	(Instruction no longer required)

3) The Link Address on this occasion will be 44.

4) The Link Address on this occasion will be 50.

18. PROBLEM-ORIENTED LANGUAGES

In working through the various programs in the preceding sections and in writing his own programs, the student may well have thought how much less work it would be if programs could be written entirely in symbolic form, as on the flow diagrams. This would save him the trouble of assigning storage locations to the various quantities, and of translating the instructions into the numerical code used by the machine.

Programs can indeed be prepared in this way, by writing them in what is called a PROBLEM-ORIENTED LANGUAGE. This consists of a succession of STATEMENTS very like the symbolic instructions used here. Various problem-oriented languages have been devised, each for a particular purpose. Well-known ones are FORTRAN and ALGOL for mathematical work, and COBOL for use in commerce. A newer one for both applications is PL/I.

These languages save work in another important way. So far our mathematical instructions have each been limited to a single operation, add, subtract, multiply or divide. With problem-oriented languages, on the other hand, a single statement can contain a complex mathematical expression, calling for many separate operations. Besides the basic arithmetic operations it can include raising of numbers to powers, trigonometric functions, etc. As an example, here is the ALGOL statement for one root of a quadratic equation:

$$x := (-b + sqrt\ (b \uparrow 2 - 4 \times a \times c)\ ) / (2 \times a)\ ;$$

Before a computer can carry out a program written in a problem-oriented language it first has to translate the program into ordinary machine instructions like the ones we have used throughout this course. To do this, the machine is loaded with a COMPILER program, and the initial program in problem-oriented language is fed in as data. The compiler program then performs the following operations:

a) it assigns addresses to the various factors involved:
b) it breaks down the long statements into sequences of normal machine-code instructions:
c) it selects any necessary subroutines: and
d) it puts in the appropriate Branch and Link instructions.

The end product is a complete program in machine code which is recorded, usually on magnetic tape or punched cards, and can then be used in the normal way.

19. CONCLUSION

At the begining of this course the student was introduced to the basic ideas underlying computers: the concept of a stored program, representation of instructions in coded form, how a store is used, and so on. The course has gone on to cover simple programming using the 3-address instructions summarised on page 83. The topics covered have included the use of Branch instructions, indirect addressing and subroutines.

Having completed this course the student should have a clear idea of what a computer can do, and how it does it. He can now learn more about the basic operations of a machine by studying the use of register and single-address instructions (See Preface), or he can move on to the use of problem-oriented languages.

APPENDICES

Supplementary Examples and Exercises

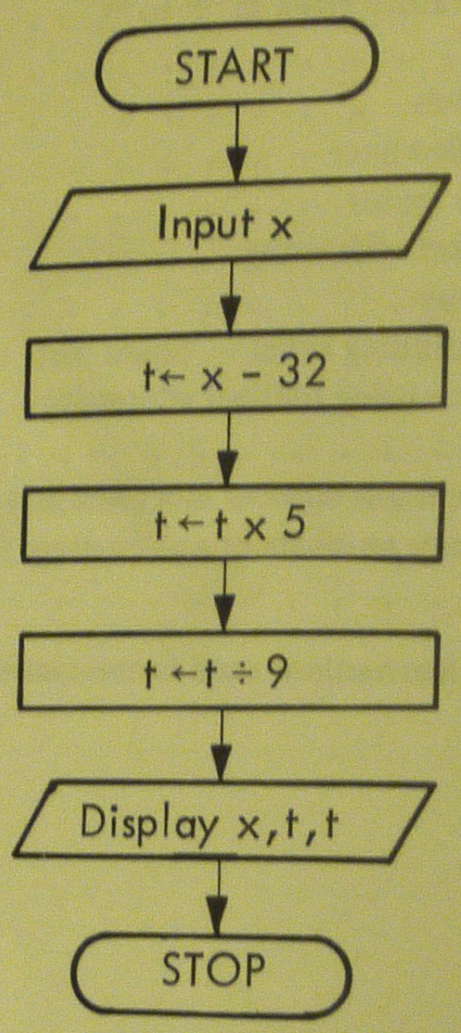
A. Programs without branches 56
A.1 Convert a temperature given in ^{o}F to ^{o}C 56
A.2 Daily shop sales 57
B. Programs with branches 62
B.1 Motor mileage allowance 62
B.2 Highest common factor 64
C. Programs using indirect addressing 66
C.1 Processing an array 66
C.2 Sorting into descending order – 'Shunt sort' 72
C.3 Daily shop sales – using indirect addressing 74
D. Subroutines 78
D.1 Finding the maximum number in a given array 78
D.2 Finding how many numbers in a given array fall within a given range 79

Summary of 3-address instructions used in this course 83

A. PROGRAMS WITHOUT BRANCHES

A.1 Convert a temperature given in °F to °C

Flow Diagram



Program

	Address	Contents	Explanation
	0	-	
	1	-	
	2	*x*	} Variables
	3	*t*	
	4	32	} Constants (to be loaded at the same time as the instructions)
	5	5	
	6	9	
	7	-	
	8	-	
Start →	9	7802 0406	Input *x* ; Display 32, 9
	10	7103 0204	$t \leftarrow x - 32$
	11	7203 0305	$t \leftarrow t \times 5$
	12	7303 0306	$t \leftarrow t \div 9$
Stop ←	13	7902 0303	Display *x*, *t*,*t*

Notes

1. During the INPUT instruction the display is used to check the constants held in locations 4 and 6.
2. The program may be adapted to return automatically to the starting point by adding an unconditional branch instruction:

14	7409 0404	Unconditional branch to 9

In this way, several different temperature conversions can be made, without having to reload the IAR from the keyboard for each run of the program.

Exercises

i) Adapt the above program to convert from °C to °F.

ii) Write a program to convert a distance given in miles into one given in kilometers.

iii) Write a program to read a distance given in yards, and to display that distance in yards, in inches and also in centimeters. (Use the three rows of a 79 display instruction to show all these equivalent lengths).

iv) Write a program to read a price given in Dollars, and to display that price in Dollars and also in French Francs and German Marks. (This problem arises when Americans are eating in a restaurant in Strasbourg!) Obtain the current exchange rates from the financial section of a newspaper.

A.2 Daily shop sales

Records are being kept of the sales of four food items sold by the School Shop during the month of June. The items are Ice Cream (I), Lemonade (L), Toffee Apples (T) and Packets of Nuts (N).

At the beginning of the month the existing stock of each item is recorded on a card thus:

Month : June		
Stocks held on first day of the month.	I :	250
	L :	144
	T :	158
	N :	400
Signature : AB		

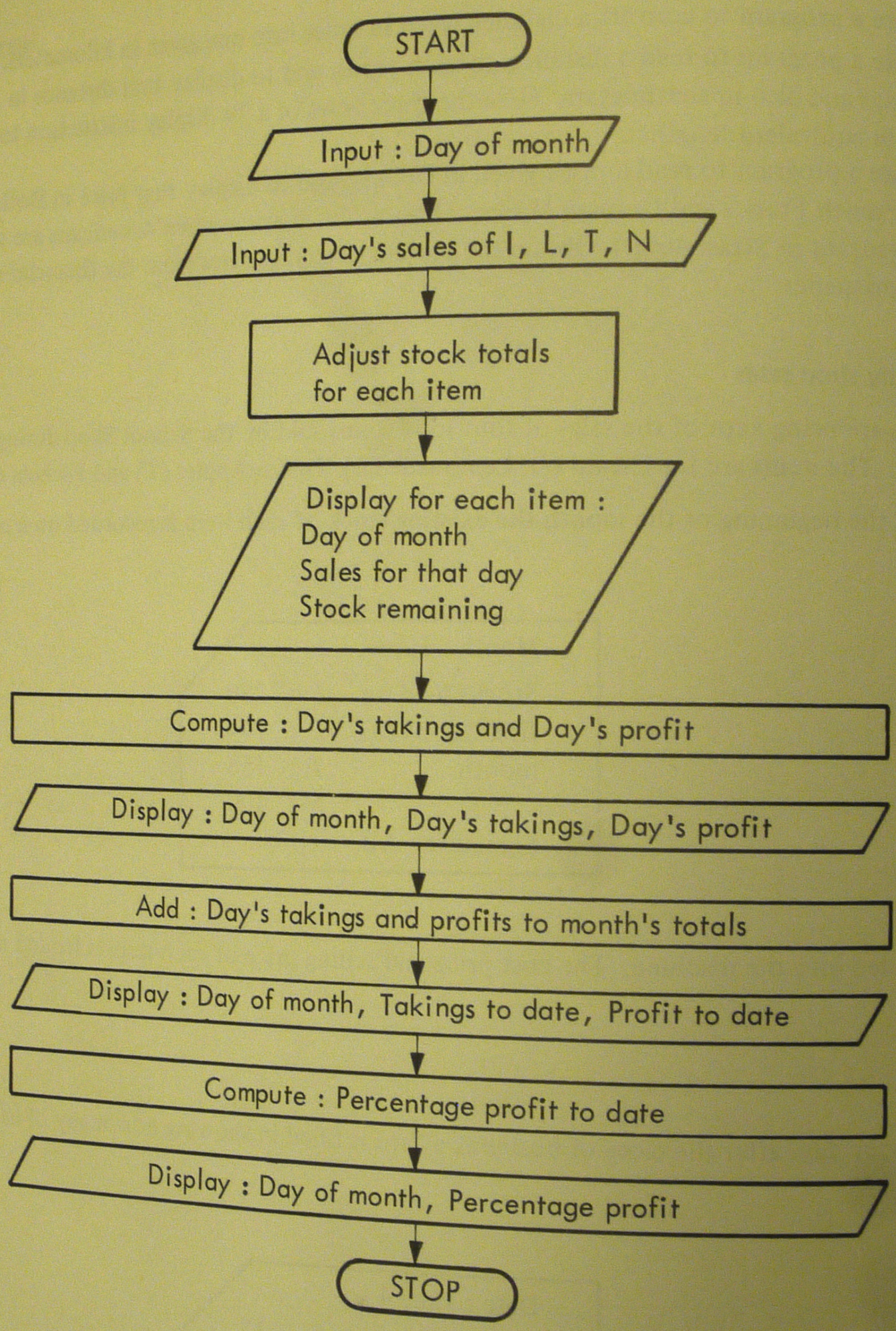
and loaded into the machine. The cost price and selling price of each item is loaded at the same time.

Every day, after the close of business, a card is filled in showing how many of each item were sold:

Date : June 4		
	I :	15
Day's sales	L :	12
	T :	25
	N :	13
Signature : AB		

Write programs to accept the daily sales figures and to keep a record of the current numbers of each item in stock. The program should also calculate the total value of sales for the day (as a check on the day's takings) and for the month so far, and should show the corresponding profit figures and the percentage profit to date.

Flow Diagram



Program

Address	Contents	Explanation		
0	-			
1	-			
2	Day of the month			Variables
3	Day's sales of I			
4	Day's sales of L			
5	Day's sales of T			
6	Day's sales of N			
7	Item takings			
8	Item profit			
9	Day's takings ; Cost of items sold to date			
10	Day's profit ; Percentage profit			
11	Total takings to date	Set to 0 at beginning of month		
12	Total profit to date			
13	Current stock of I	Set to existing stock at beginning of month		
14	Current stock of L			
15	Current stock of T			
16	Current stock of N			
17	Cost price of I	Constants	To be loaded at the beginning of the month	
18	Cost price of L			
19	Cost price of T			
20	Cost price of N			
21	Selling price of I			
22	Selling price of L			
23	Selling price of T			
24	Selling price of N			
25	100			
26	-			
27	-			
Start → 28	7802 0202	Input Day of month		
29	7803 0202	Input Day's sales of I ; Display Day of month		
30	7804 0302	Input Day's sales of L ; Display Day's sales of I, Day of the month		
31	7805 0403	Input Day's sales of T ; Display Day's sales of L,I		
32	7806 0504	Input Day's sales of N ; Display Day's sales of T,L		
33	7113 1303	Current stock ← Current stock – Day's sales		
34	7114 1404			
35	7115 1505			
36	7116 1606			

(continued)

Program (continued)

Address	Contents	Explanation
37	7902 0313	Display : Day of month
38	7902 0414	Day's sales
39	7902 0515	Current stock
40	7902 0616	
41	7109 0909	Day's takings ← 0
42	7110 1010	Day's profit ← 0
43	7207 2103	Day's takings on *I*
44	7108 2117	Profit per ice cream
45	7208 0803	Day's profit on *I*
46	7009 0907	Add Day's takings on *I* to Day's total takings
47	7010 1008	Add Day's profit on *I* to Day's total profit
48	7207 2204	
49	7108 2218	As addresses 43 to 47
50	7208 0804	but for Lemonade
51	7009 0907	
52	7010 1008	
53	7207 2305	
54	7108 2319	As above but for Toffee Apples
55	7208 0805	
56	7009 0907	
57	7010 1008	
58	7207 2406	
59	7108 2420	
60	7208 0806	As above but for Nuts
61	7009 0907	
62	7010 1008	
63	7902 0910	Display Day of month, Day's takings, Day's profit
64	7011 1109	Add Day's takings to Total takings to date
65	7012 1210	Add Day's profit to Total profit to date
66	7902 1112	Display day of month, Total takings and Profit to date
67	7109 1112	Cost of items sold to date
68	7310 1209	Profit to date ÷ Cost of items sold to date
69	7210 1025	Percentage profit to date
70	7902 1025	Display Day of month, Percentage profit to date, 100.

Notes

i) At the beginning of the month locations 11 to 70 are loaded as detailed in the table. The contents of the store are then recorded on tape. (See Installation and Operating Instructions).

ii) After the close of business for each day, the store is reloaded from the tape, the program run and the displayed data copied into the ledger.

iii) The contents of the store are then recorded again, to preserve the program and the running totals of takings and profit.

iv) The prices of each item may be varied at any time during the month by altering the contents of addresses 21 – 24. Similarly, the stock figures and the cost price may be altered. If this is done, a record of the changes should be kept in the ledger, so that the final figures may be properly checked.

v) The program is long because of the repetition of the same steps for each item of food. A shorter program, which allows for a greater number of stock items, can be prepared using indirect addressing, and an example of this method is given later.

Exercises

a) Adapt the above program to deal with six items, and prepare suitable daily record cards and sales ledgers.

b) Add a program to display for each item the number of days of stock remaining, assuming that sales remain constant and equal to the current day's figures.

c) Add a program to update the stock values when a new delivery arrives. A record card should be prepared as follows:

Date 12.6.69	I :	144
Delivery of stock	L :	72
items added to store	T :	50
	N :	0
Signature: AB		

B. PROGRAMS WITH BRANCHES

B.1 Motor mileage allowance

In some firms, cars are provided for employees' use on company business. In others an allowance is paid to an employee using his own car. A typical car allowance is discribed as follows:

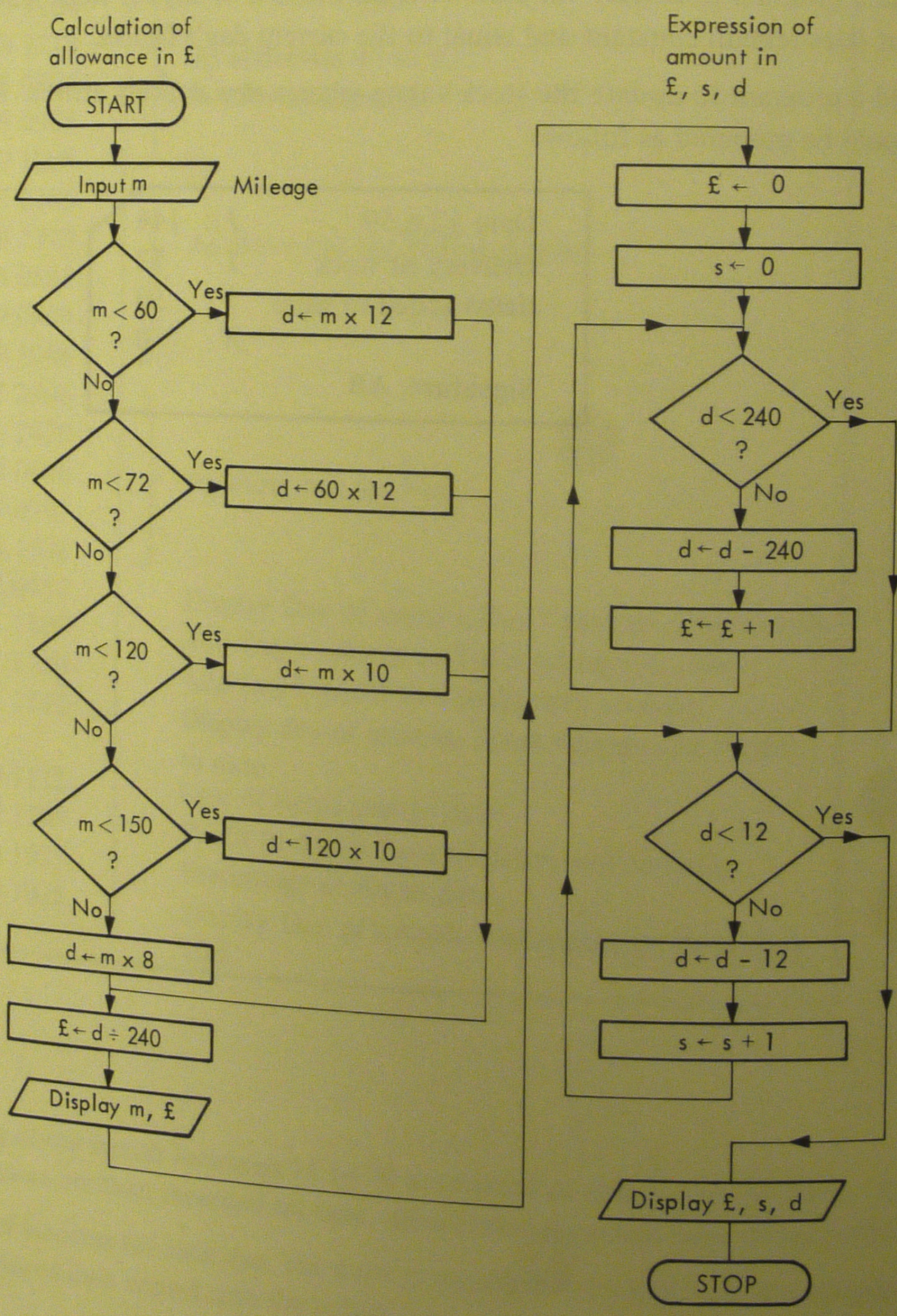
i) If the weekly mileage is less than, or equal to, 60 miles, then the allowance is 1/- per mile.

ii) For mileage between 60 and 120, the allowance is 10d per mile, but the minimum refund is £3.

iii) For mileage in excess of 120 the refund is 8d, per mile with a minimum of £5.

Write a program to calculate the total refund for a given weekly mileage, expressing the result first as a decimal of a pound, and secondly in pounds, shillings and pence.

Flow Diagram



Program

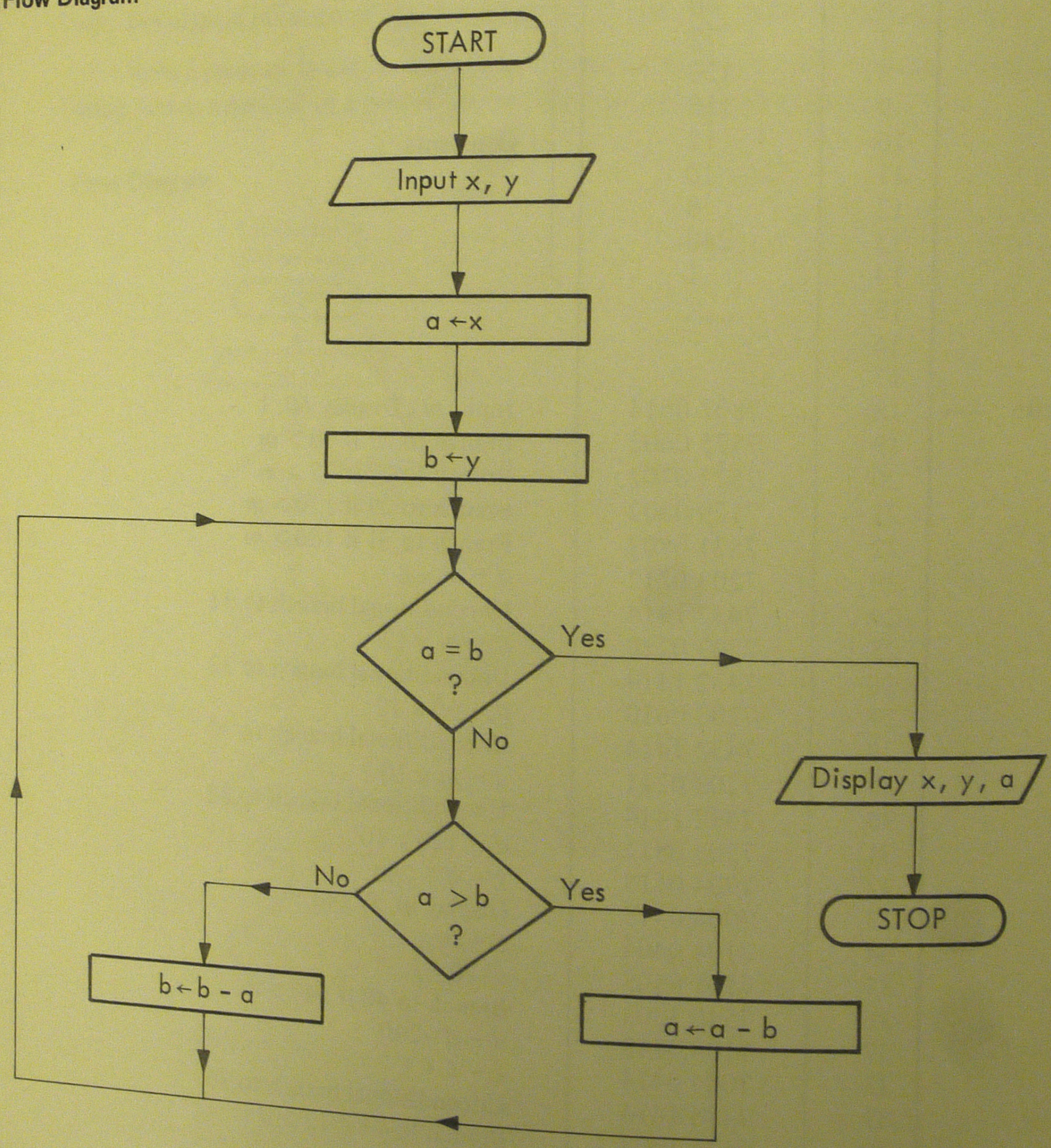
	Address	Contents	Explanation
	0	-	
	1	-	
	2	*m*	} Variables
	3	*d*	
	4	£	
	5	*s*	
	6	60	} Constants
	7	72	
	8	120	
	9	150	
	10	12	
	11	10	
	12	8	
	13	240	
	14	1	
	15	-	
	16	-	
	17	-	
Start →	18	7802 0614	Input *m* ; Display 60, 1
	19	7525 0602	Branch to 25 if 60 > *m*
	20	7527 0702	Branch to 27 if 72 > *m*
	21	7529 0802	Branch to 29 if 120 > *m*
	22	7531 0902	Branch to 31 if 150 > *m*
	23	7203 0212	*d* ← *m* x 8
	24	7432 1414	Unconditional branch to 32
	25	7203 0210	*d* ← *m* x 12
	26	7432 1414	Unconditional branch to 32
	27	7203 0610	*d* ← 60 x 12
	28	7432 1414	Unconditional branch to 32
	29	7203 0211	*d* ← *m* x 10
	30	7432 1414	Unconditional branch to 32
	31	7203 0811	*d* ← 120 x 10
	32	7304 0313	£ ← *d* ÷ 240
	33	7902 0413	Display *m*, £, 240
	34	7104 0404	£ ← 0
	35	7105 0505	*s* ← 0
	36	7540 1303	Branch to 40 if 240 > *d*
	37	7103 0313	*d* ← *d*−240
	38	7004 0414	£ ← £ + 1
	39	7436 1414	Unconditional branch to 36
	40	7544 1003	Branch to 44 if 12 > *d*
	41	7103 0310	*d* ← *d*−12
	42	7005 0514	*s* ← *s* + 1
	43	7440 1414	Unconditional branch to 40
Stop ←	44	7904 0503	Display £, *s*, *d*

B.2 Highest common factor

The highest common factor (H.C.F) of two numbers is the greatest number which will divide exactly into the two given numbers. If two numbers are equal, then each is equal to the H.C.F.

This program uses the property that the H.C.F. of two unequal numbers is also a factor of their difference. After the two numbers have been read, a loop is formed in which the smaller is subtracted from the larger until equality is achieved. This final value is then displayed as the H.C.F. together with the two given numbers.

Flow Diagram



Program

	Address	Contents	Explanation
	0	-	
	1	-	
	2	*x*	Variables
	3	*y*	
	4	*a*	
	5	*b*	
	6	0	Constant
	7	-	
	8	-	
Start →	9	7802 0606	Input *x* ; Display 0
	10	7803 0206	Input *y* ; Display *x*, 0
	11	7004 0206	$a \leftarrow x$
	12	7005 0306	$b \leftarrow y$
	13	7419 0405	Branch to 19 if $a = b$
	14	7517 0405	Branch to 17 if $a > b$
	15	7105 0504	$b \leftarrow b - a$
	16	7413 0606	Unconditional branch to 13
	17	7104 0405	$a \leftarrow a - b$
	18	7413 0606	Unconditional branch to 13
Stop ←	19	7902 0304	Display *x*, *y*, *a*

C. PROGRAMS USING INDIRECT ADDRESSING

In commercial applications, the computer is often used to perform the same set of operations on a large number of different items of data. For example, in wages calculation, the same simple arithmetic processes will be applied for every worker in a large factory.

The Schools Computer is able to handle problems of this type by the use of indirect addressing, whereby the same set of instructions can be made to operate on data held in a set of several different storage locations.

In the programs that follow, we shall call the numbers in a set of storage locations an 'array', and we shall refer to each number in the set by means of a 'subscripted variable' of the form A [i]. The subscript i can be used to represent the address of a particular number in the array. For example, if storage locations 30 to 47 hold a set of 18 numbers, then we can refer to the whole set as

$$\text{A}\,[i]\ ,\quad 30 \leqslant i \leqslant 47$$

where 30 is the lower bound of the array subscript and 47 is the upper bound.

C.1 Processing an array

The following program contains several routines joined together in sequence. Each routine uses the technique of indirect addressing to operate on the array

$$\text{A}\,[i]\ ,\quad 1 \leqslant i \leqslant n$$

i.e. on the set of numbers contained in locations 1 to n of the store. n can have any value from 2 to 50.

Routine 1 reads the array into the store. Routine 2 searches the array for the largest and smallest numbers ('MAX' and 'MIN'), and Routine 3 carries out a transformation on the numbers to make the smallest zero and the largest 100. The transformation consists of subtracting 'MIN' from each number and multiplying the result by

$$\frac{100}{\text{MAX}-\text{MIN}}$$

Routine 4 displays the numbers three at a time so that they may be recorded. This set of routines could be used to process the examination results of a class.

The technique of indirect addressing enables us to write a single instruction such as

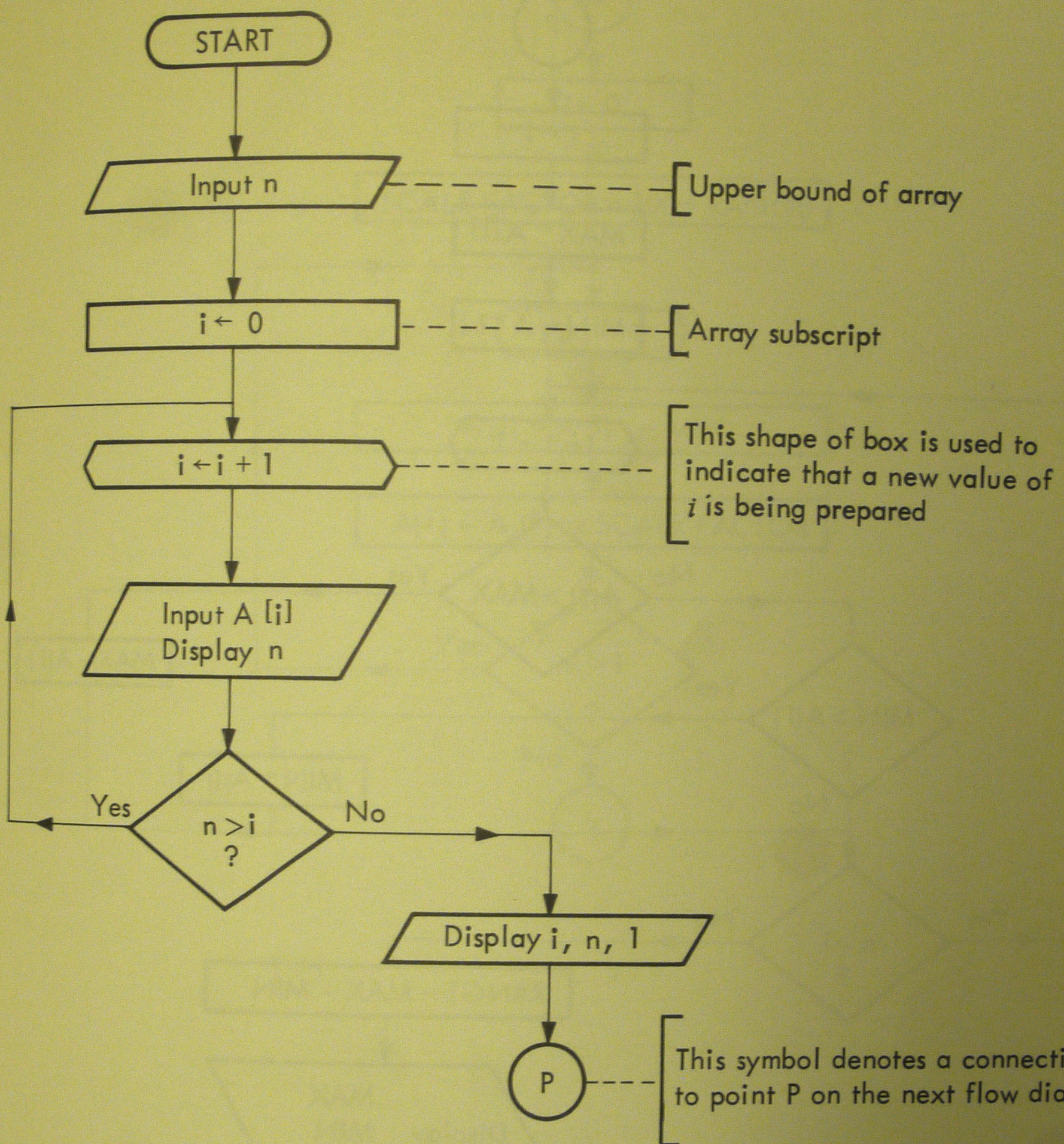
$$\text{A}\,[i] \leftarrow \text{A}\,[i] - \text{MIN}$$

and then to change the value of i before looping back to obey the same instruction. As in section 16, it is essential that all the operands of an indirect addressing instruction should be indirectly addressed. For example, in the following program, the coding for A [i] ← A [i] − MIN is 9152 5262 (see Address 91). This is equivalent to the instruction 71 i i 54 where i is the subscript indicating the current address under consideration. Beginners often forget to allocate an indirect address to operands which are not subscripts – the location for MIN in this case.

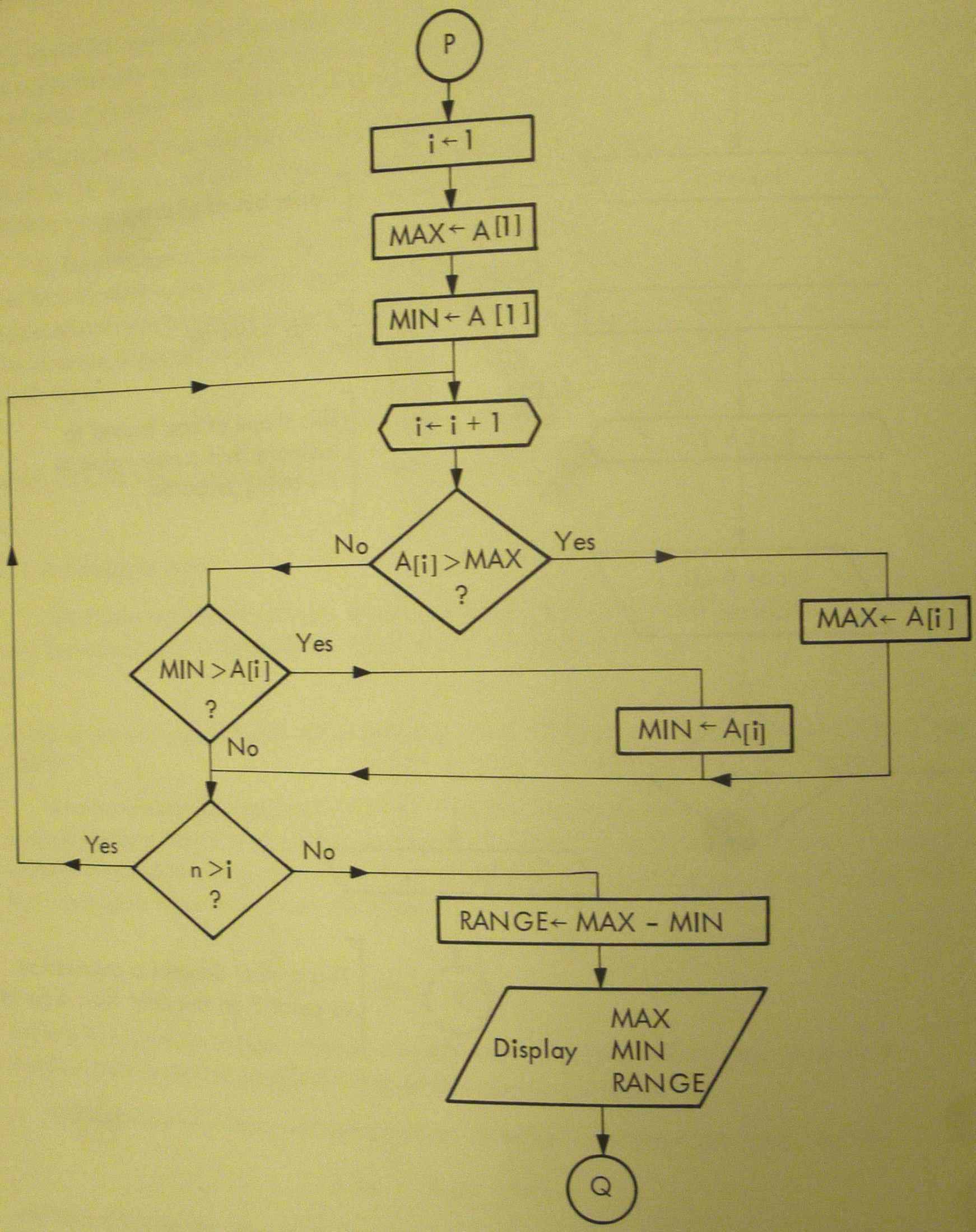
Similarly in Branch instructions using indirect addressing the branch address has to be expressed indirectly as well as the addresses of the quantities being compared. Thus the instruction at address 78, i.e. 9564 5261 is equivalent to 7584 i 53.

Flow Diagrams

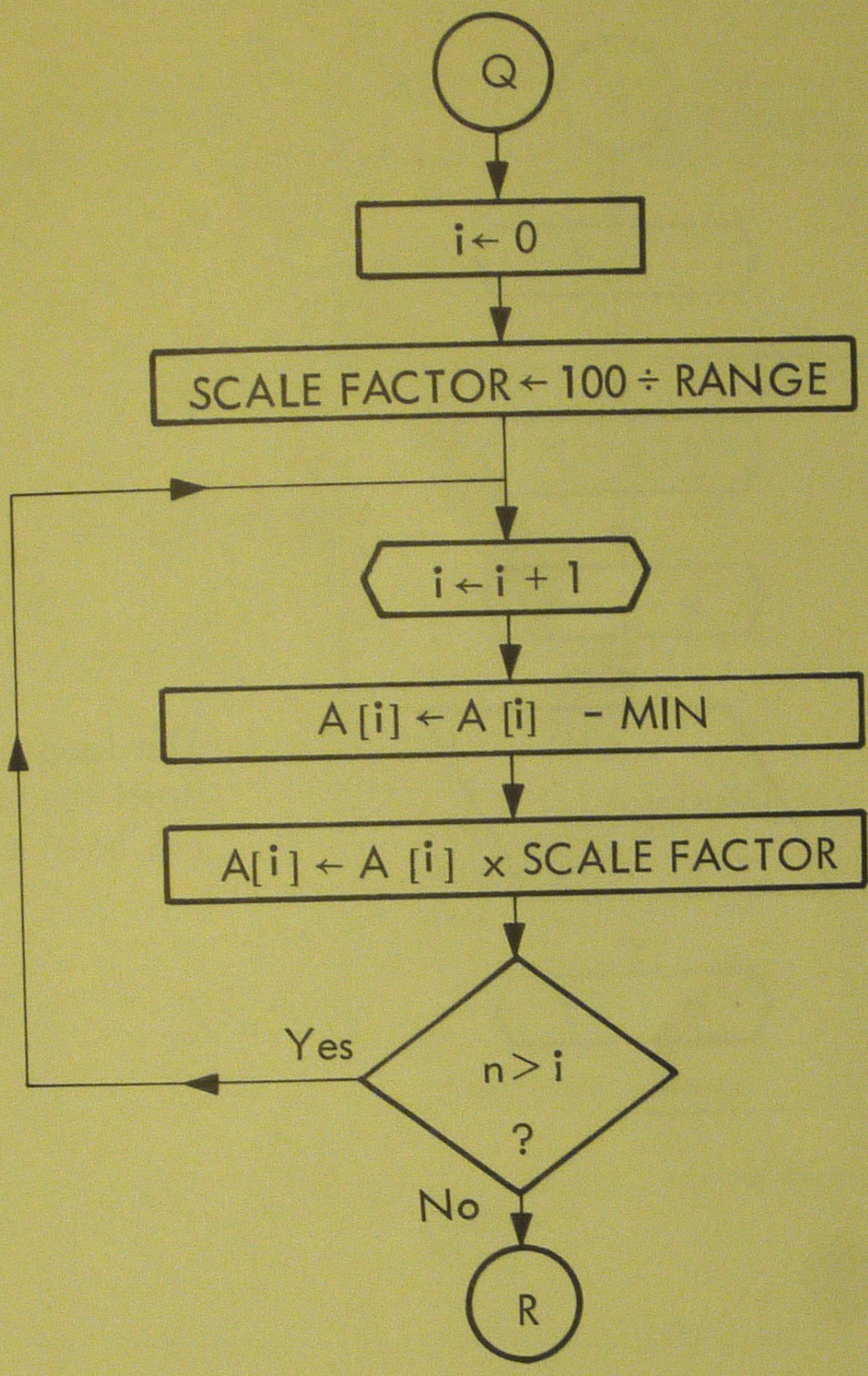
Routine 1 – INPUT



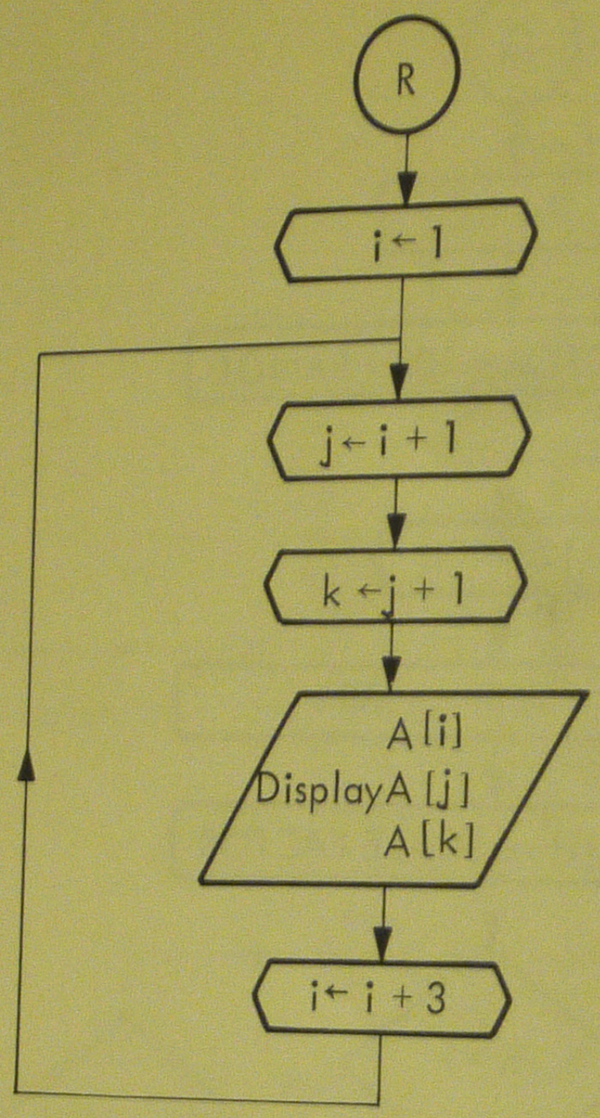
Routine 2 – MAX AND MIN



Routine 3 – SCALE



Routine 4 – DISPLAYED SCALED NUMBERS



The loop is repeated until the whole array has been displayed

Program

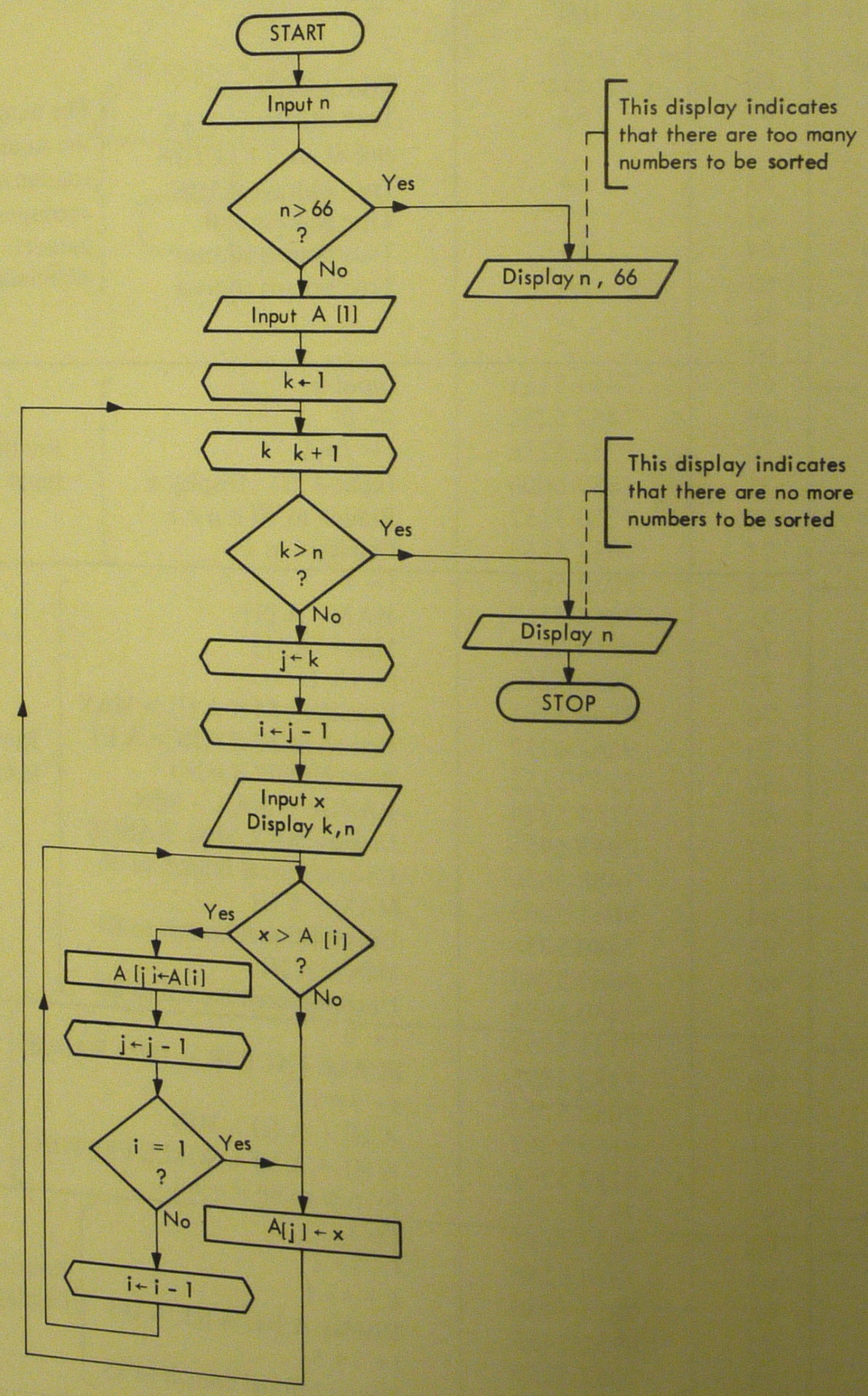
	Address	Contents	Explanation		
	0	-			
	1	A [1]	Array		Variables
	2	A [2]			
	–	–			
	–	–			
	50	A [50]			
	51	*n*	Upper bound of array		
	52	*i*	Array subscript		
	53	MAX;Scale factor			
	54	MIN; *j*			
	55	Range; *k*			
	56	1			Constants
	57	0			
	58	100			
	59	3			
	60	51	True address of *n*	The numbers in the Address column are the corresponding indirect addresses.	
	61	53	True address of MAX and SCALE FACTOR		
	62	54	True address of MIN		
	63	57	True address of 0		
	64	84	True branch address		
	65	86	True branch address		
	66	-			
	67	-			
Enter →	68	7851 5151	Input *n*		
	69	7152 5252	$i \leftarrow 0$		Routine 1
	70	7052 5256	$i \leftarrow i + 1$		Input
	71	9852 6060	Input A [*i*] ; Display *n*		
	72	7570 5152	Branch to 70 if $n > i$		
Exit ←	73	7952 5156	Display *i*, *n*, 1		
Enter →	74	7052 5657	$i \leftarrow 1$		
	75	7053 0157	MAX ← A [1]		
	76	7054 0157	MIN ← A [1]		
	77	7052 5256	$i \leftarrow i + 1$		Routine 2
	78	9564 5261	Branch to 84 if A [*i*] > MAX		MAX and MIN
	79	9565 6252	Branch to 86 if MIN > A [*i*]		
	80	7577 5152	Branch to 77 if $n > i$		
	81	7155 5354	RANGE ← MAX – MIN		
	82	7953 5455	Display MAX, MIN, RANGE		
Exit ←	83	7488 5656	Unconditional branch to 88		
	84	9061 5263	MAX ← A [*i*]		
	85	7480 5656	Unconditional branch to 80		
	86	9062 5263	MIN ← A [*i*]		
Enter →	87	7480 5656	Unconditional branch to 80		
	88	7152 5252	$i \leftarrow 0$		
	89	7353 5855	SCALE FACTOR ← 100 ÷ RANGE		Routine 3
	90	7052 5256	$i \leftarrow i + 1$		Scale
	91	9152 5262	A [*i*] ← A [*i*] – MIN		
Exit ←	92	9252 5261	A [*i*] ← A [*i*] x SCALE FACTOR		
Enter →	93	7590 5152	Branch to 90 if $n > i$		
	94	7052 5657	$i \leftarrow 1$		
	95	7054 5256	$j \leftarrow i + 1$		Routine 4
	96	7055 5456	$k \leftarrow j + 1$		Display
	97	9952 5455	Display A [*i*], A [*j*], A [*k*]		
	98	7052 5259	$i \leftarrow i + 3$		
	99	7495 5656	Unconditional branch to 95		

C.2 Sorting into descending order — 'Shunt sort'

This program sorts n numbers into descending order; n can have any value up to 66. The program first reads in the value of n and compares it with 66 to check that the store is large enough to hold all the data.

The first number is then read-in and stored in location 1. The second number is read-in and compared with the first. If the second number is the greater of the two, the first number is moved to location 2 and the second placed in location 1. Otherwise, the second number is placed in location 2. As each subsequent number is read-in, it is compared with the numbers already stored. Any numbers which are smaller are moved to the next higher address, leaving a space for the new number at the appropriate place. Thus, when the n^{th} number has been read-in, the highest number will be found in location 1 and the lowest in location n.

Flow Diagram



Program

	Address	Contents	Explanation	
	0	-		
	1	A [1]	Array	Variables
	2	A [2]		
	-	-		
	-	-		
	66	A [66]		
	67	*n*		
	68	*k*		
	69	*j*		
	70	*i*		
	71	*x*		
	72	66		Constants
	73	1		
	74	0		
	75	71	True address of *x*	
	76	74	True address of 0	
	77	92	True branch address	
	78	-		
	79	-		
Start →	80	7867 6767	Input *n*	
	81	7598 6772	Branch to 98 if $n > 66$	
	82	7801 0101	Input A [1]	
	83	7068 7374	$k \leftarrow 1$	
	84	7068 6873	$k \leftarrow k + 1$	
	85	7597 6867	Branch to 97 if $k > n$	
	86	7069 6874	$j \leftarrow k$	
	87	7170 6973	$i \leftarrow j - 1$	
	88	7871 6867	Input *x* ; Display *k*, *n*	
	89	9577 7570	Branch to 92 if $x >$ A [*i*]	
	90	9069 7576	A [*j*] $\leftarrow x$	
	91	7484 7373	Unconditional branch to 84	
	92	9069 7076	A [*j*] $\leftarrow$ A [*i*]	
	93	7169 6973	$j \leftarrow j - 1$	
	94	7490 7073	Branch to 90 if $i = 1$	
	95	7170 7073	$i \leftarrow i - 1$	
	96	7489 7373	Unconditional branch to 89	
Stop ←	97	7967 6767	Display *n*	
	98	7967 7272	Display *n*, 66	

C.3 Daily shop sales — using indirect addressing

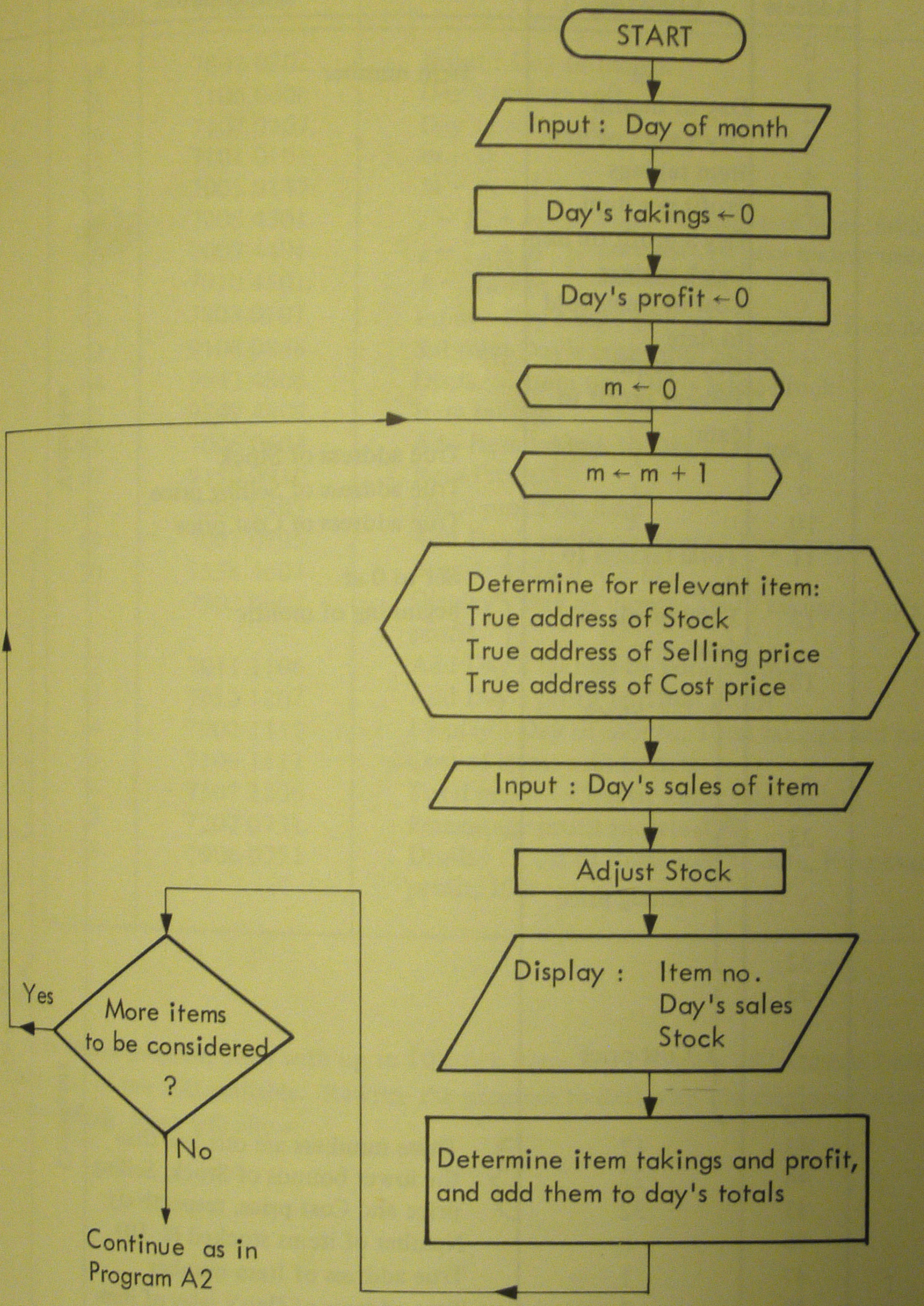
In program A2 some of the instructions were repeated for each item of stock. Therefore if many items had to be considered the program would be very lengthy. By using the technique of indirect addressing, programs of this kind can be written much more economically. The instructions that have to be repeated are written only once, but they are contained within a loop and operate on each stock item in turn.

In the following program the current stock figures, selling prices and cost prices are treated as arrays. For example, the current stock figures form an array:

$$A\,[i]\ ,\ 13 \leqslant i \leqslant 22$$

i.e. they are held in locations 13 to 22 of the store. Before carrying out the repeated instructions the true address, i, of the relevant stock figure is formed by adding the constant 12, i.e. one less than the lower bound of the array, to the variable m which represents the number of the item under consideration. The true addresses of the selling and cost prices are formed in a similar way.

Flow Diagram



Program

Address	Contents	Explanation	
0	-	Item number	Variables
1	*m*		
2	Day of month		
3	Day's sales of item		
4	Item takings		
5	Profit per item		
	Day's profit on item		
6	Day's takings		
	Cost of items sold to date		
7	Day's profit		
	Percent profit to date		
8	*i*	True address of Stock	
9	*j*	True address of Selling price	
10	*k*	True address of Cost price	
11	Total takings to date	Set to 0 at beginning of month	
12	Total profit to date		
13 - - - 22	Stock		Constants
23 - - - 32	Selling price		
33 - - - 42	Cost price		
43	12	These numbers are one less than the lower bounds of Stock, Selling price and Cost price, respectively.	To be loaded at the beginning of the month
44	22		
45	32		
46	*n*	Number of items stocked ($\leqslant$ 10)	
47	1	True address of Item number	
48	3	True address of Day's sales of item	
49	4	True address of Item takings	
50	5	True address of Profit per item	
51	100		
52	-		
53	-		

Continued

Program (continued)

	Address	Contents	Explanation
Start→	54	7802 0202	Input : Day of month
	55	7106 0606	Day's takings ← 0
	56	7107 0707	Day's profit ← 0
	57	7101 0101	$m \leftarrow 0$
	58	7001 0147	$m \leftarrow m + 1$
	59	7008 4301	$i \leftarrow 12 + m$ } Form true address of Stock, Selling
	60	7009 4401	$j \leftarrow 22 + m$ } price and Cost price for item under
	61	7010 4501	$k \leftarrow 32 + m$ } consideration
	62	7803 0101	Input : Day's sales of item ; Display : Item no.
	63	9108 0848	Subtract Day's sales from Stock
	64	9947 4808	Display : Item no., Day's sales, Stock
	65	9249 4809	Item takings
	66	7006 0604	Add Item takings to Day's takings
	67	9150 0910	Item Profit
	68	7205 0503	Day's profit on item
	69	7007 0705	Add Day's profit on item to Day's profit
	70	7558 4601	Branch to 58 if $n > m$
	71	7902 0607	Display : Day of month, Day's takings, Day's profit
	72	7011 1106	Add Day's takings to Total takings
	73	7012 1207	Add Day's profit to Total profit
	74	7902 1112	Display : Day of month, Total takings and profit
	75	7106 1112	Cost of items sold to date
	76	7307 1206	Total profit ÷ Cost of items sold to date
	77	7207 0751	Percentage profit to date
Stop←	78	7906 0751	Display : Cost of items sold to date, Percentage profit, 100.

Exercise

The above program deals with up to 10 stock items and occupies 78 locations (including 3 spare) out of the 100 available. Rewrite the program to deal with the maximum number of items that the storage space will allow.

D. SUBROUTINES

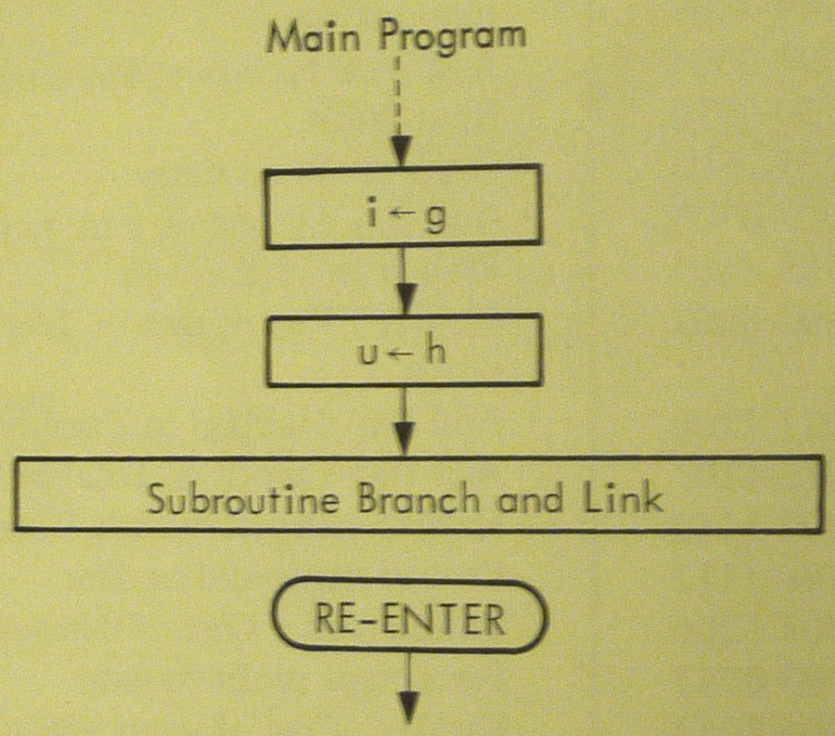
D.1 Finding the maximum number in a given array

This is written as a closed subroutine, i.e. it could be used at several places in one program, or by several unrelated programs. Before branching to this subroutine the program must provide it with the upper and lower bounds of the array under consideration. The process of providing this preliminary information and branching to the entry point of the subroutine is referred to as CALLING the subroutine.

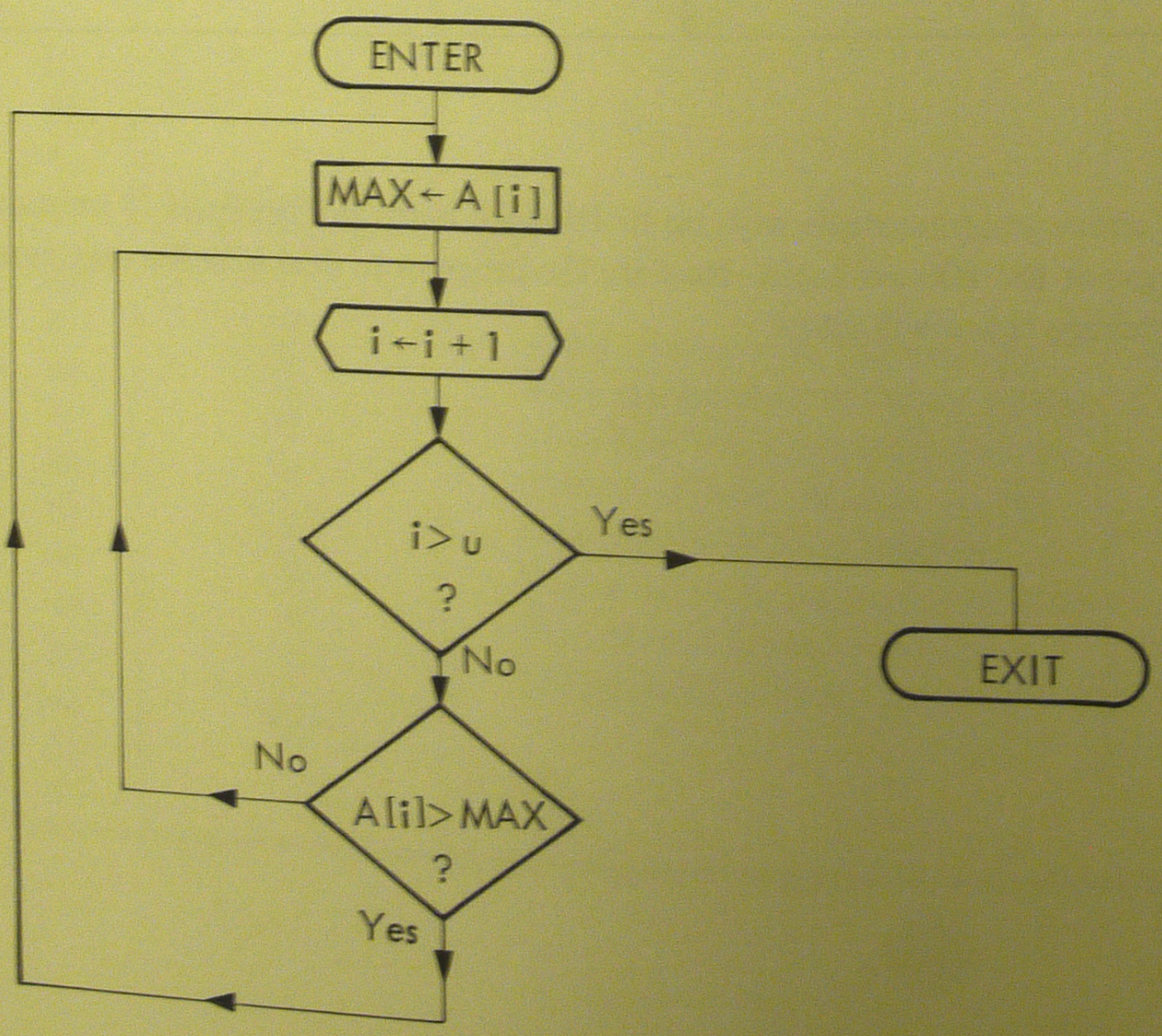
In the present example, the array is defined as

$$A[i], \quad g \leqslant i \leqslant h$$

i.e. g is the lower bound and h the upper bound. The calling sequence is as follows, i and u being subroutine variables:



Flow Diagram



	Address	Contents	Explanation	
	83	Link address		Variables
	84	*u*		
	85	*i*		
	86	MAX		
	87	0		Constants
	88	1		
	89	86	True address of MAX	
	90	87	True address of 0	
	91	93	True branch address	
	92	-		
Enter →	93	9089 8590	MAX ← A [*i*]	
	94	7085 8588	$i \leftarrow i + 1$	
	95	7598 8584	Branch to 98 if $i > u$	
	96	9591 8489	Branch to 93 if A [*i*] > MAX	
	97	7494 8787	Unconditional branch to 94	
Exit ←	98	9483 9090	Unconditional branch to link address (i.e. return to main program)	

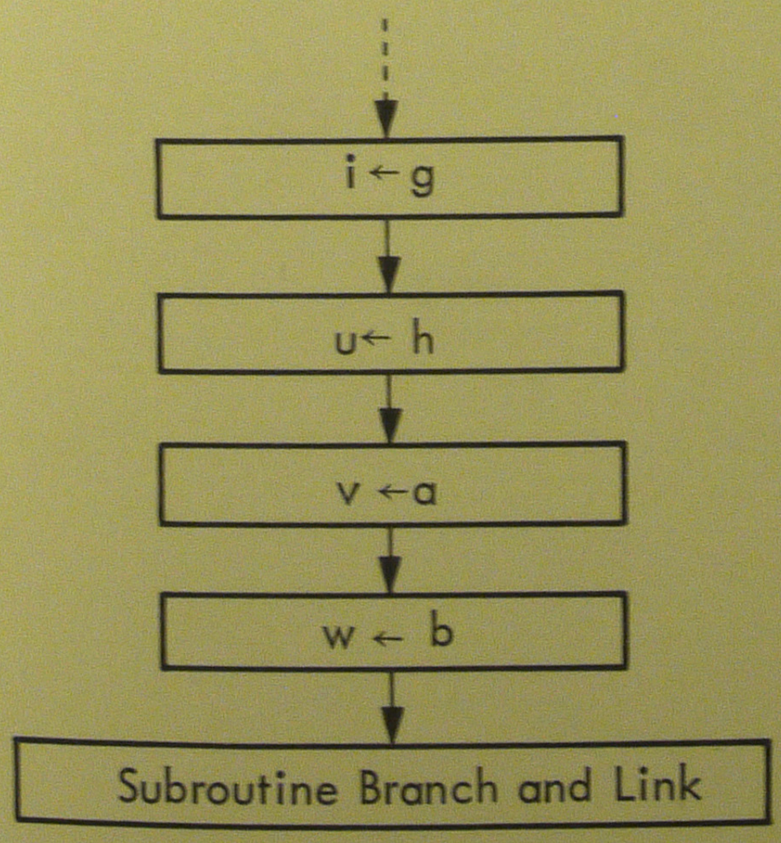
D.2 Finding how many numbers in a given array fall within a given range

The array is defined by

$$A[i],\ g \leqslant i \leqslant h$$

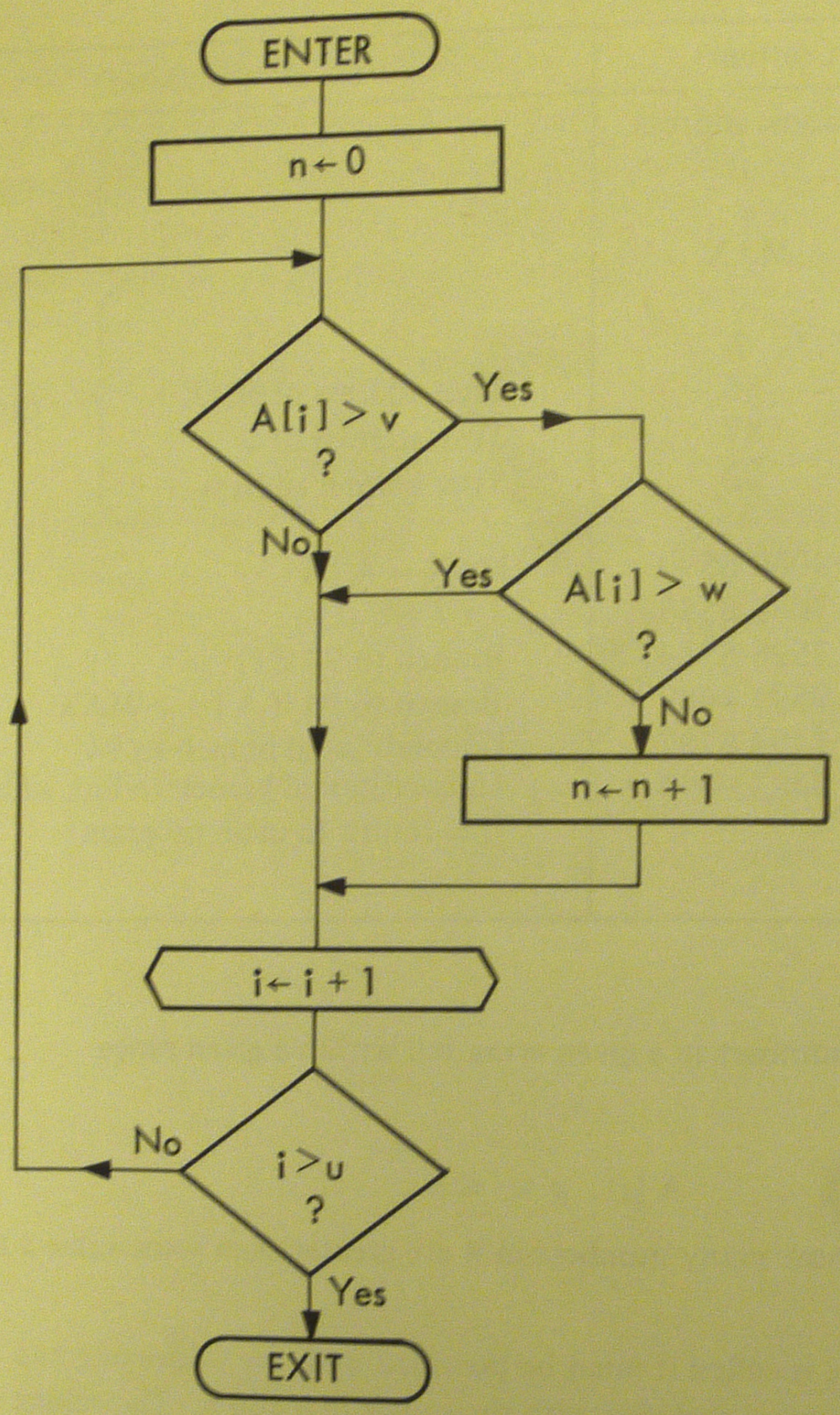
and we wish to find out how many numbers in it are greater than some value a but not greater than a second value b.

When the subroutine is called it must be provided with the values of g and h, i.e. the lower and upper bounds of the array, and also with the values of a and b. The calling sequence is:



i, *u*, *v*, *w* and *n* are subroutine variables. *n* represents the count of the numbers falling within the given specification.

Flow Diagram



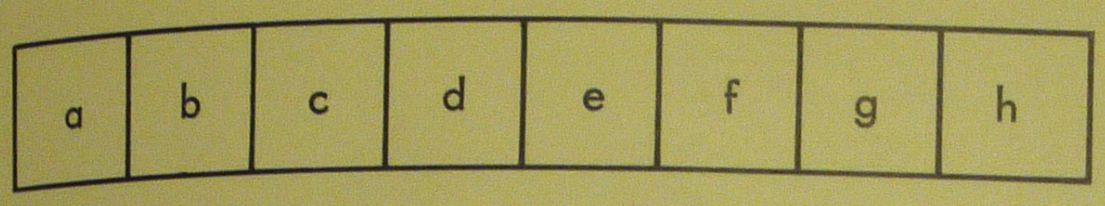
Subroutine

	Address	Contents	Explanation	
	60	Link address		
	61	*i*		
	62	*u*		Variables
	63	*v*		
	64	*w*		
	65	*n*		
	66	1		
	67	63	True address of *v*	
	68	64	True address of *w*	Constants
	69	77	True branch address	
	70	74	True branch address	
	71	-		
Enter →	72	7165 6565	$n \leftarrow 0$	
	73	9569 6167	Branch to 77 if A $[i] > v$	
	74	7061 6166	$i \leftarrow i + 1$	
	75	7580 6162	Branch to 80 if $i > u$	
	76	7473 6666	Unconditional branch to 73	
	77	9570 6168	Branch to 74 if A $[i] > w$	
	78	7065 6566	$n \leftarrow n + 1$	
	79	7474 6666	Unconditional branch to 74	
Exit ←	80	9460 6767	Unconditional branch to link address	

Exercises

a) Write a closed subroutine to find the minimum number in a given array.
b) Write a closed subroutine to find the range of values in a given array.
c) Write a closed subroutine to sort a given array into ascending order.

SUMMARY OF 3-ADDRESS INSTRUCTIONS USED IN THIS COURSE



ab is the 2-digit operation code
cd, ef and gh are the three 2-digit addresses

Codes (for direct addressing)

ab	
70	(cd) ← (ef) + (gh)
71	(cd) ← (ef) - (gh)
72	(cd) ← (ef) x (gh)
73	(cd) ← (ef) ÷ (gh)
74	Branch to cd if (ef) = (gh)
75	Branch to cd if (ef) > (gh)
76	Branch to cd if |(ef)| > |(gh)|
77	If (gh) ≠ 0 branch to cd and store link address in ef
78	Input to cd; display (ef) and (gh)
79	Display (cd), (ef) and (gh)

The type of addressing is determined by the digit a:

Digit a	Type of addressing	True addresses
7	Direct	cd, ef, gh
9	Indirect	(cd), (ef), (gh)