J. Jeenel

Programs as a Tool for Research in Systems Organization

Abstract: A program for the solution of a problem by a data-processing system constitutes a conceptual link between the problem and the machine. It is proposed that both problems and machine organization be studied in terms of programs. A data-processing system may for this purpose be considered a collection of units such as arithmetic units and stores of different types and characteristics. A standard representation of programs is proposed for studying the organization of systems composed of processing units and stores. This approach may lead to more efficient and systematic design of data-processing systems as well as to improved programming methods for existing machines.

1. Introduction

A computer program for the solution of a problem is frequently viewed only as a control device which causes the machine to perform a desired task. This, of course, is the primary purpose of programs in present-day data processing. It is felt, however, that programs may also be considered a conceptual link between the problems and the machines, and therefore merit attention from a more general point of view. A study of problems as well as systems organization in terms of programs appears to be a promising approach to the formulation of a theory of automatic data processing. Such a theory could be expected to lead to more efficient systems design and also to more efficient programming techniques.

Most current efforts in the field of programming research seem to be directed either toward specific systems or particular classes of applications. The majority of the effort appears to be concentrated in the area of non-machine languages, or shorthand codes, with the objective of making the program a better control device, that is, to make it easier to write and more readily understandable for the machine. The languages thus constitute an answer to the question "How do I talk to the machine?" rather than "What should I tell the machine?" To best serve this purpose the languages are descriptive rather than functional. Essentially, they permit a large amount of intricate control information to be summed up in compact form, but they do not permit formal manipulation of their symbols according to a theory for optimizing the use of the processing units and stores. The research proposed in this paper is concerned with the primary functions of data-processing equipment, the processing and storing of data, rather than with the necessary subordinate function of controlling the equipment.

To be useful as a research tool, programs should be represented in a uniform and compact manner. A standard representation of programs as a hierarchy of "basic forms" would facilitate the process of programming. This subject as well as much of the material in the present paper is treated in detail by the author in a forthcoming book on programming for stored-program calculators. The representation of a program in terms of basic forms emphasizes the storage of the problem data and the flow of this information through the system, two particularly important aspects of the machine solution of a problem.

This method of program representation has also been found applicable to systems other than conventional stored-program calculators; it has provided some insight into the functioning of a "decentralized" system of card-handling data-processing machines, and it has been applied successfully to a hypothetical machine in which the data are identified by means essentially different from conventional addresses.

A data-processing system in this paper is considered to be a collection of arithmetic units and stores. This paper shows, through examples, how programs might be used as a research tool in studying the organization of machine systems represented in this manner.

2. Basic form and address patterns

As a first example, consider a problem which requires the elements of two 3-dimensional arrays $f_{i,j,k}$ and $g_{i,j,k}$ to be added in pairs producing a 3-dimensional array

$r_{i,j,k}$. The elements of each array may be visualized to be associated with the nodes of a cube as illustrated by Figure 1. The problem requires the elements from the same relative positions in the f- and g-cubes to be added in pairs to form the elements of the r-cube.

- *Statement of the problem*

$$r_{i,j,k}=f_{i,j,k}+g_{i,j,k} \qquad \text{for } \begin{matrix} i \\ j \\ k \end{matrix} \Big\} 0, 1, \ldots, 9.$$

- *Address assignments*

$$L(f_{i,j,k})=1000+100i+10j+\quad k,$$

$$L(g_{i,j,k})=2000+\quad i+10j+100k,$$

$$L(r_{i,j,k})=3000+100i+10j+\quad k.$$

- *Flow chart*

See Figure 2, pages 108 and 109.

The arrays $f_{i,j,k}$ and $g_{i,j,k}$ are assumed to be available in a random-access store. The elements of the result array $r_{i,j,k}$ will be placed in the same store. The addresses, or locations, $L(f_{i,j,k})$, $L(g_{i,j,k})$, and $L(r_{i,j,k})$ of the elements of the arrays are given as linear functions of the indices i, j, and k. As indicated by these address patterns, the first input array $f_{i,j,k}$ and the output array $r_{i,j,k}$ are both stored in order by i, j, and k, whereas the second input array $g_{i,j,k}$ is available in the opposite order by k, j, and i.

The program is represented as a basic form by the flow chart in Figure 2. The boxes on the flow chart are arranged in columns from left to right by increasing frequency of execution, which also corresponds to the different levels of control implied by the system of "loops within loops." The k-level represented by the rightmost column represents the innermost loop; it is of the lowest logical order and has the highest relative frequency of 10^3 executions. The frequencies of execution for the j- and i-levels, which are of higher logical orders, are 10^2 and 10, respectively. The level of highest logical order on the extreme left of the basic form is "open," that is, it does not contain a loop since it is executed only once per execution of this program.

The computations $r_{i,j,k}=f_{i,j,k}+g_{i,j,k}$ are performed in Box C on the k-level. The instructions in this box contain the variable addresses $L(f_{i,j,k})$, $L(g_{i,j,k})$, and $L(r_{i,j,k})$. The address $L(g_{i,j,k})$ is stepped with respect to k, j, and i in Boxes B, E, and G on the k-, j-, and i-levels, respectively. It is initially set in Box J on the open level. The addresses $L(f_{i,j,k})$ and $L(r_{i,j,k})$ are set initially also on the open level in Box I. Unlike $L(g_{i,j,k})$, the addresses $L(f_{i,j,k})$ and $L(r_{i,j,k})$ are stepped with respect to all three indices on the k-level.

Stepping $L(f_{i,j,k})$ with respect to k, for instance, may be described by the following recursion formula of the address pattern $L(f_{i,j,k})$:

$$L(f_{i,j,k+1})=L(f_{i,j,k})+1.$$

Similarly, stepping with respect to j and at the same time resetting k from its maximum value 9 to its initial value 0 is described by

$$L(f_{i,j+1,0})=L(f_{i,j,9})+1.$$

The right-hand members of these two recursion formulas contain the same constant, 1. Since this constant is the stepping constant to be applied to $L(f_{i,j,k})$ on the k-level, Box A on this level will also serve to step $L(f_{i,j,k})$ with respect to j, simultaneously resetting the index k contained in $L(f_{i,j,k})$. The corresponding recursion formulas of the address pattern $L(g_{i,j,k})$ are

$$L(g_{i,j,k+1})=L(g_{i,j,k})+100$$

and

$$L(g_{i,j+1,0})=L(g_{i,j,9})-890.$$

The different constants in the right-hand members of these formulas indicate that the variable address $L(g_{i,j,k})$ must be operated upon separately on the k- and j-levels.

The indices were assigned in the order i, j and k to the levels of the basic form from left to right. If they were assigned in the opposite order k, j, and i, that is, in the order in which the array $g_{i,j,k}$ is stored, the situation with respect to the generation of the variable addresses would be reversed. $L(g_{i,j,k})$ could then be stepped with respect to all three indices on the rightmost level whereas $L(f_{i,j,k})$ and $L(r_{i,j,k})$ would have to be operated upon on the center levels of the basic form.

Each series of consecutive executions of a level is terminated by a terminating box (Boxes D, F, and H) at the bottom of the level. Program execution enters in the upper left-hand corner of a basic form and eventually leaves it by the lower left-hand corner. The execution pattern of many basic forms is somewhat analogous to the action of a counter. The rightmost level would be executed repeatedly until the associated index reaches its maximum value. Then the next level to the left is executed once before the rightmost level is again executed repeatedly. This corresponds to a carry into an adjacent counter position. When the indices of several adjacent levels have reached their maximum values, program execution proceeds through the terminating boxes from right to left across the basic form in much the same way a carry propagates across several counter positions.

Programs of greater complexity may be represented in a standard manner by a hierarchy of basic forms as indicated by Figure 3. In such a hierarchy, basic forms contain on their levels lower-order basic forms as details in serial and parallel combinations. The different stages of such a hierarchy again reflect different levels of control. From the point of view of sequencing, a hierarchy of basic forms helps to separate the two fundamental types of sequencing elements, loops and branches, which represent repetition and decision, respectively. On a given detail in a hierarchy of basic forms, one may have extensive branching but only trivial looping represented by a

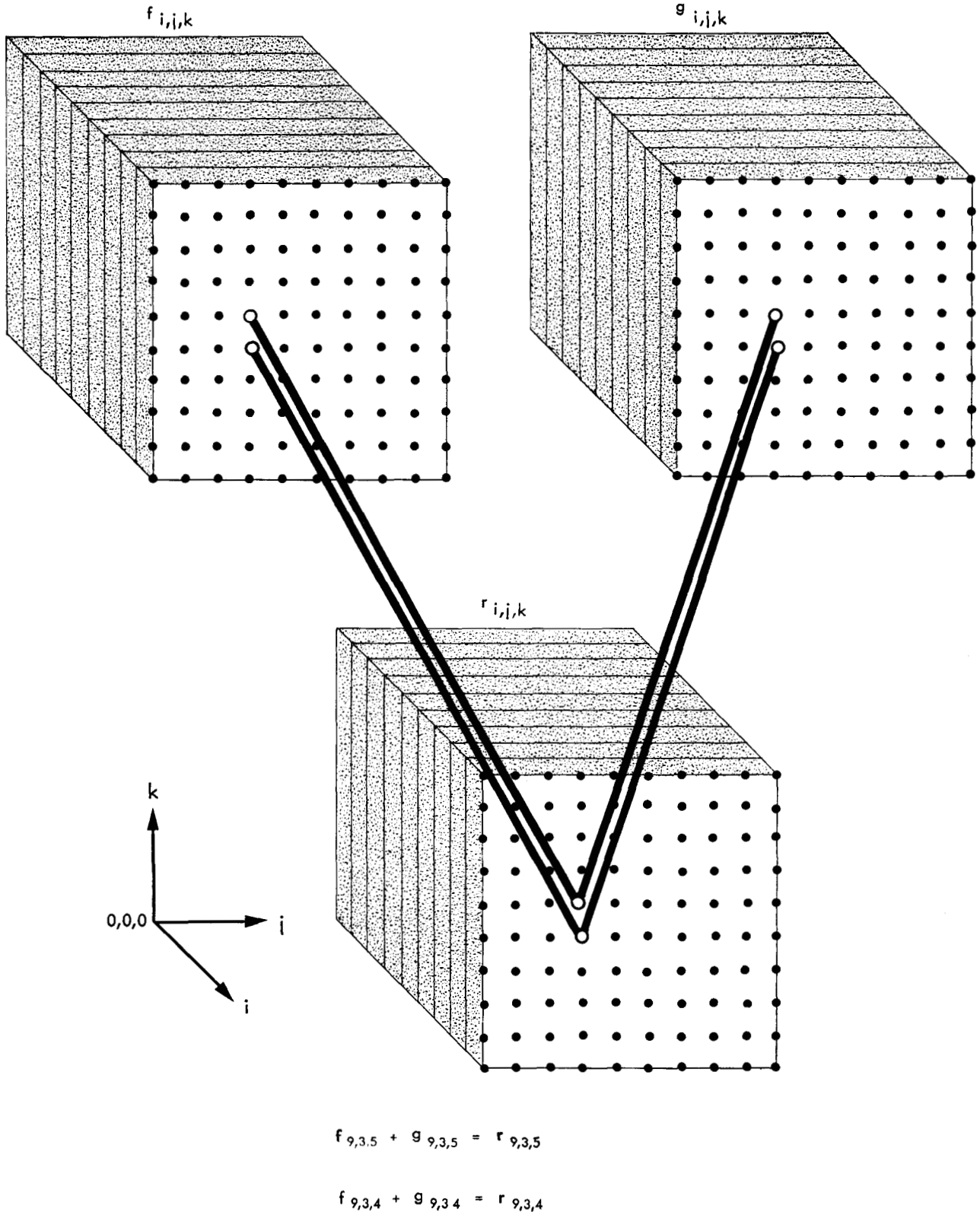


Figure 1 **The elements of 3-dimensional arrays arranged at the nodes of cubes. The elements $r_{i,k,j}$ are to be generated as the sum of the corresponding elements $f_{i,j,k}$ and $g_{i,j,k}$.**

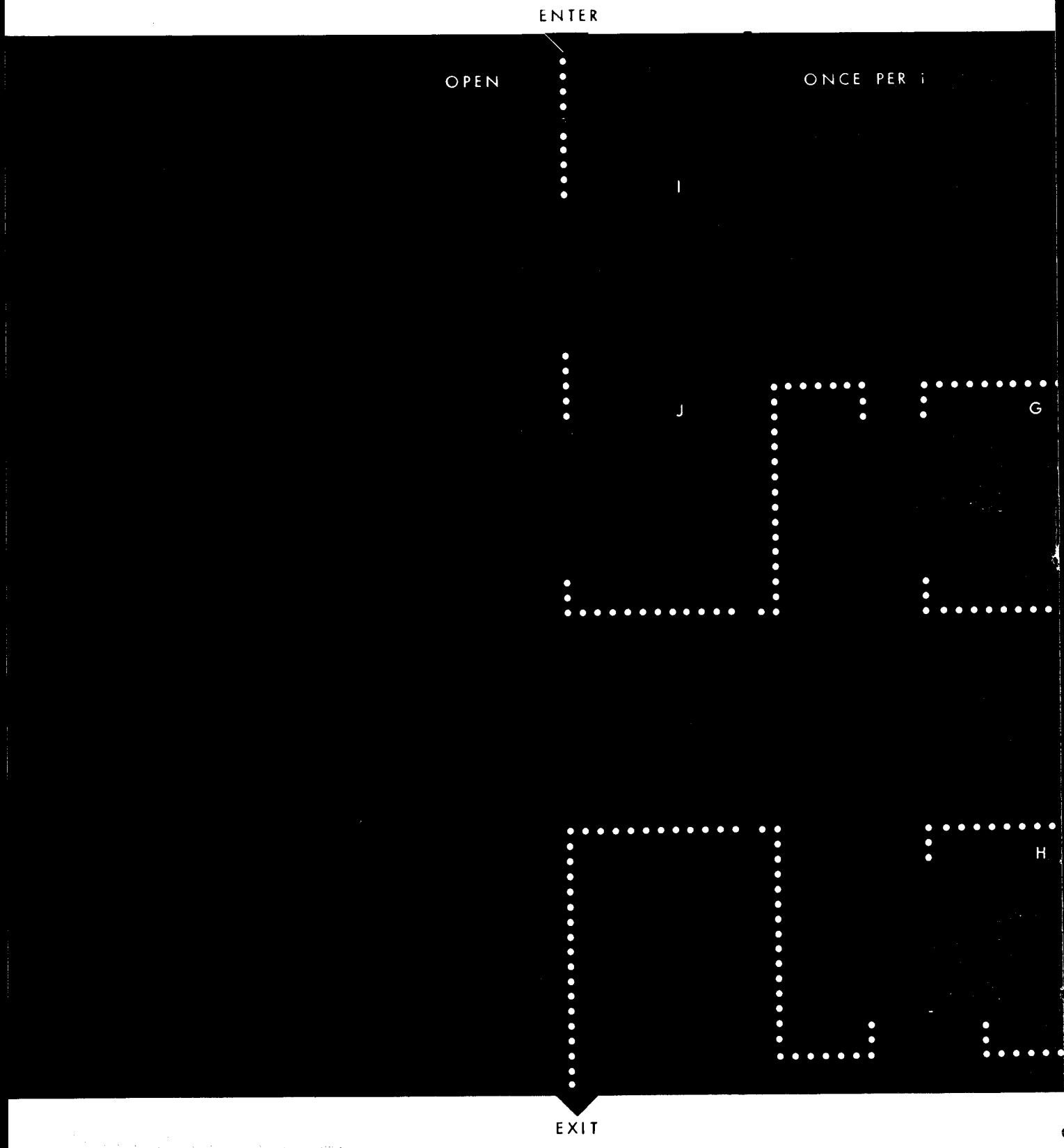
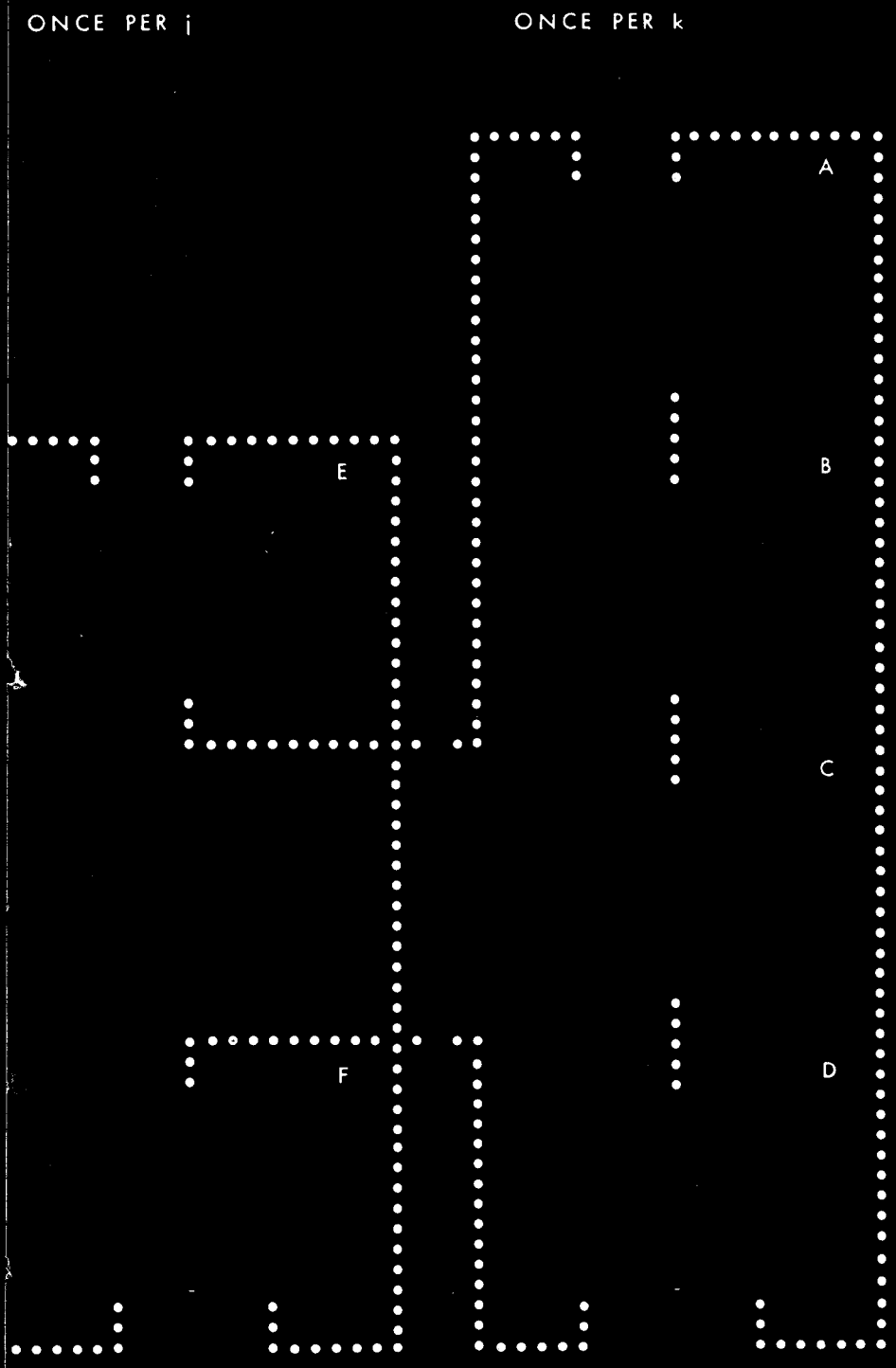
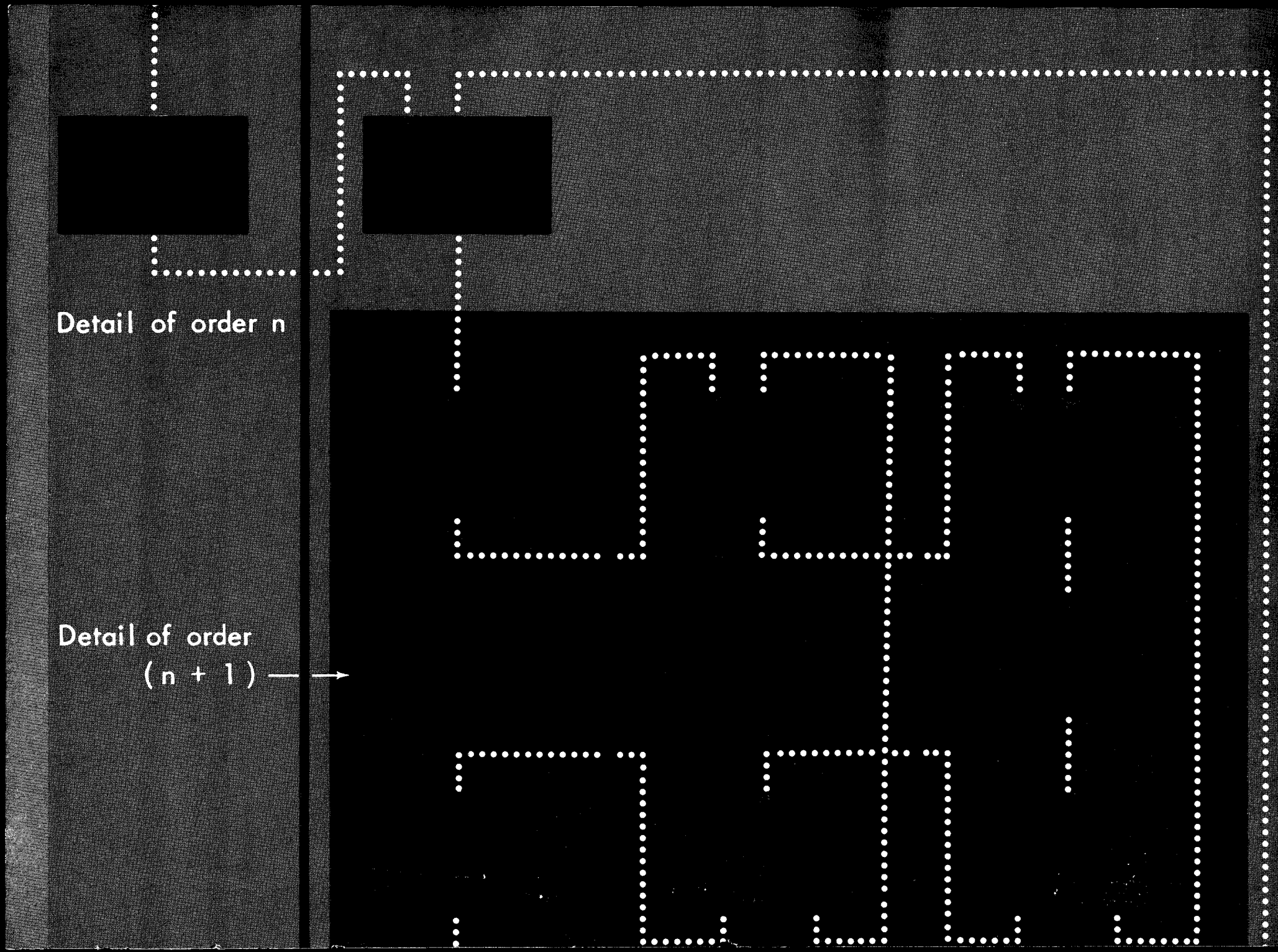


Figure 2 **A program as a basic form of four columns representing the four logical levels.**


ONCE PER j
ONCE PER k
A
B
C
D
E
F


Detail of order n
Detail of order
(n + 1)

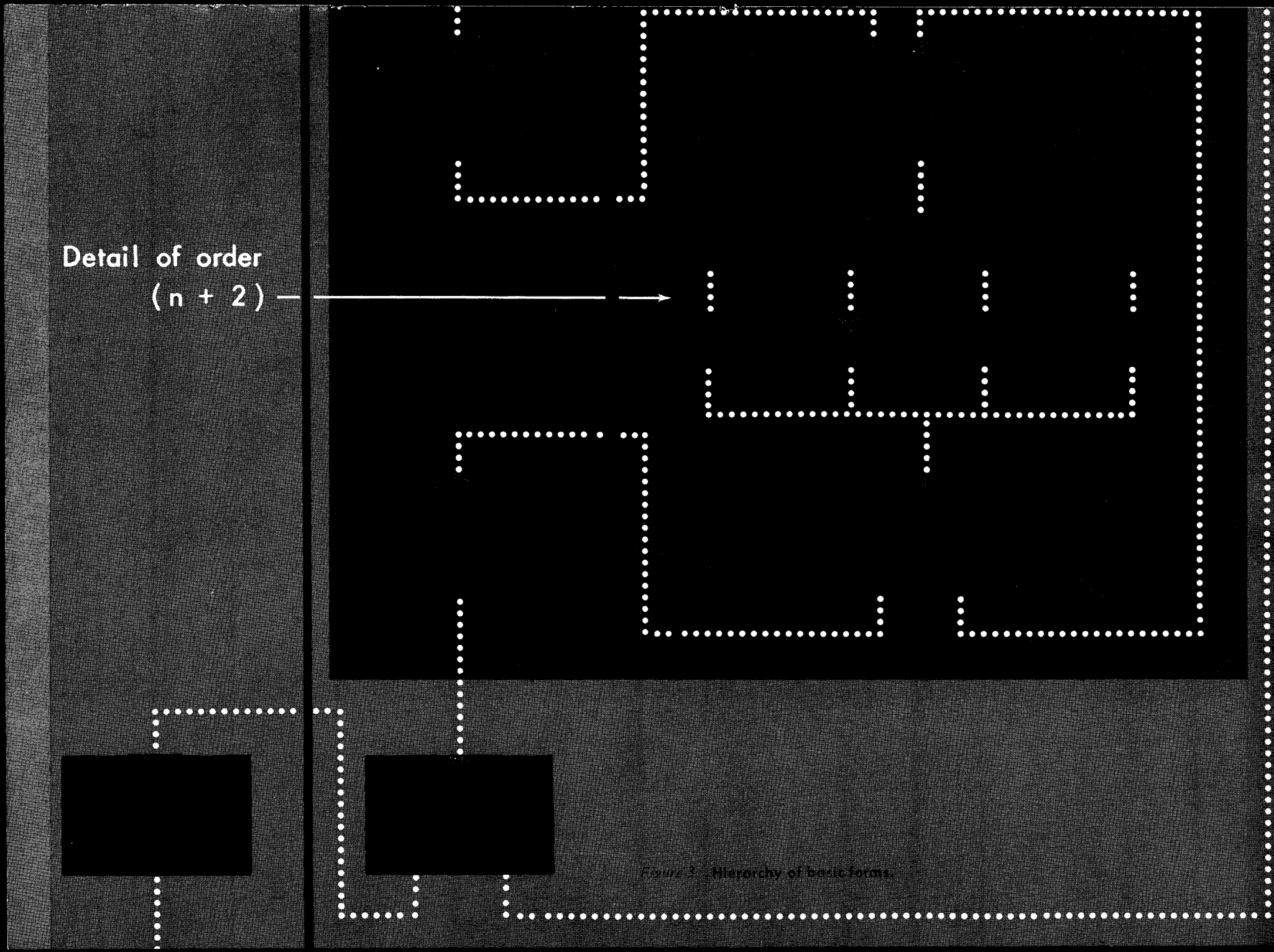


Figure 3 Hierarchy of basic forms.

single loop; more intricate looping would be shown on lower-order details which at this stage are considered "black boxes." Conversely, a given detail may contain any number of cascaded loops with trivial branching represented by the terminating branches associated with the loops; more intricate branching arrangements would in this case be enclosed in "black boxes" to be shown as separate lower-order details.

For a second example of a program, consider the problem previously discussed with the following modification. The second input array, $g_{i,j,k}$, was assumed to be available in the opposite order from the other two arrays $f_{i,j,k}$ and $r_{i,j,k}$. It will now be assumed that $g_{i,j,k}$ is also stored by i, j, and k so that all three arrays associated with the basic form are stored in the same order.

- *Statement of the problem*

$$r_{i,j,k}=f_{i,j,k}+g_{i,j,k} \quad \text{for } \left.\begin{matrix} i \\ j \\ k \end{matrix}\right\} = 0, 1, \ldots, 9.$$

- *Address assignments*

$$L(f_{i,j,k})=1000+100i+10j+k,$$

$$L(g_{i,j,k})=2000+100i+10j+k,$$

$$L(r_{i,j,k})=3000+100i+10j+k.$$

- *Flow chart*

See Figure 4.

The program, which performs the same computing as the one in the previous example, is now represented by a basic form of only two, rather than four, levels. The right-hand level of the present basic form is executed once per triplet of indices (i, j, k). The variable address $L(g_{i,j,k})$, which in the previous example necessitated the existence of the center levels of the wider basic form, is now stepped on the right-hand level according to the recursion formulas

$$L(g_{i,j,k+1})=L(g_{i,j,k})+1,$$

$$L(g_{i,j+1,0})=L(g_{i,j,9})+1,$$

$$L(g_{i+1,0,0})=L(g_{i,9,9})+1,$$

with the same stepping constant 1 in their right-hand members.

A group of indices, such as the triplet (i, j, k), assigned to a level of a basic form is termed a "compound index." A compound index narrows the width of a basic form by consolidating levels which would otherwise correspond to the individual component indices. Whether compound indices can be formed depends on the properties of all input and output address patterns associated with the basic form. A necessary condition for combining several indices into a compound index is that the portions of all address patterns in which the indices appear be linearly dependent. For example, the address patterns

$$L_1=a_1+b_1i+c_1j+d_1k+e_1l,$$

$$L_2=a_2+b_2i+c_2j+d_2k+e_2l,$$

would be linearly dependent with respect to the indices i and k if the following linear relation with constant coefficients R and C holds:

$$(b_2i+d_2k)=R(b_1i+d_1k)+C.$$

A matrix-multiplication problem, for instance, according to the formula

$$c_{ik}=\sum_{j=1}^{n} a_{ij}b_{jk}$$

with all three matrices stored by columns would have address patterns of the form

$$L(a_{ij})=K_1+i+nj,$$

$$L(b_{jk})=K_2 \quad + \quad j+nk,$$

$$L(c_{ik})=K_3+i \quad +nk.$$

Since not all three address patterns are linearly dependent with respect to any pair of indices, compound indices could not be formed. The basic form for the matrix-multiplication program would contain four levels, one for each of the indices i, j, k and an open level.

3. Rearranging and sorting

The programs for the two versions of the problem $r_{i,j,k}=f_{i,j,k}+g_{i,j,k}$ given in Section 2 of this paper are represented by a four-level and a two-level basic form, respectively, although both programs perform the same computing. The greater width of the first basic form is necessitated only by the order in which the arrays are stored in the machine. A portion of the complexity of a program is thus not inherent in the problem itself, which requires only a certain amount of computing to be performed, but is due to the specific manner in which the machine is made to retain the data.

The two-level basic form may be considered to represent the computing as required by the problem, whereas the four-level basic form may be considered to perform an implicit rearranging process on the $g_{i,j,k}$ concurrently with the computing. These two aspects of the program, computing and rearranging, and their effects on the width of the basic form could be separated in the following manner. The rearranging of the $g_{i,j,k}$ could be performed explicitly in its entirety prior to computing. To rearrange $g_{i,j,k}$ from the order in which they were given for the first version of the problem to the order in which they were assumed for the second version would require a four-level basic form. This rearranging program could then be followed by a two-level basic form for computing like the one given for the second version of the problem. The four-level basic form for the first version may thus be visualized as a two-level basic form for the computing, upon which a four-level basic form is superimposed for concurrent rearranging.

The width of a basic form reflects the degree to which the different arrays of data connected by the basic form

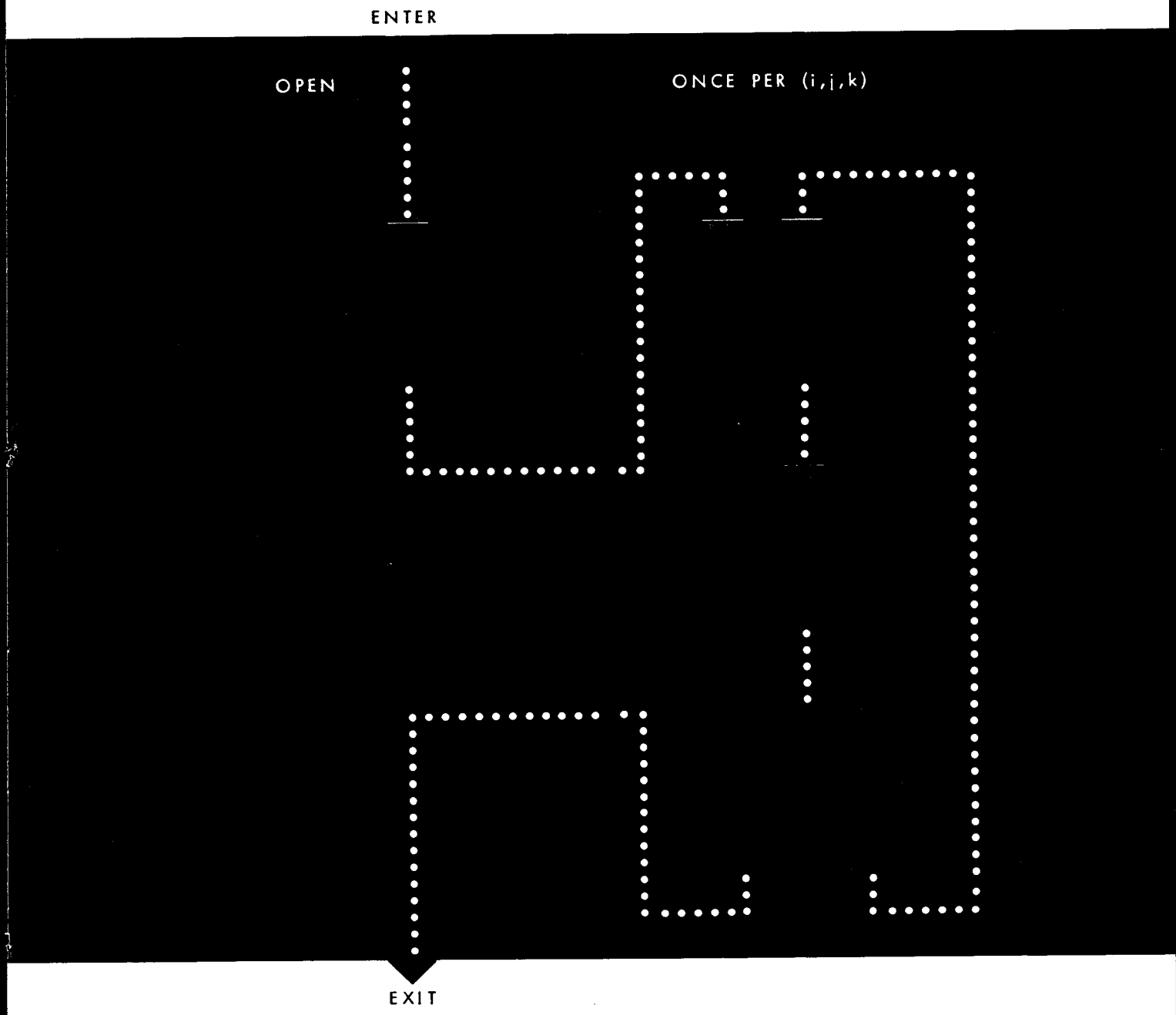


Figure 4 **A two-level basic form for the same computing as represented by Figure 2 but with the arrays of data available in compatible order.**

conform to each other as far as the ordering of the data within the arrays is concerned. If the $g_{i,j,k}$ were given in order by j, k and i, which is only "partially opposite" to the order by i, j and k in which the $f_{i,j,k}$ and $r_{i,j,k}$ are stored, the program for computing and concurrent rearranging would be represented by a three-level (i.e., intermediate) basic form. In this intermediate case, the properties of the address patterns

$$L(f_{i,j,k}) = 1000 + 100i + 10j + k,$$

$$L(g_{i,j,k}) = 2000 + i + 100j + 10k,$$

$$L(r_{i,j,k}) = 3000 + 100i + 10j + k,$$

would permit the compound index (j, k) to be formed and the address patterns to be rewritten as follows:

$$L(f_{i,j,k}) = 1000 + 100i + (j, k),$$

$$L(g_{i,j,k}) = 2000 + i + 10(j, k),$$

$$L(r_{i,j,k}) = 3000 + 100i + (j, k).$$

The order in which indices are assigned to the levels of a basic form determines the order in which the program scans the arrays of data. One may, for instance, visualize the elements of an array $g_{i,j,k}$ corresponding to the nodes of a 3-dimensional mesh where, in the conventional manner, i and j are assigned to the axes in a horizontal plane and where the k-direction is vertical. (Compare Fig. 1.) With the indices assigned in the order k, j and i to the levels of a basic form from left to right, the program would first scan a horizontal row, then the horizontal row behind it in the same plane, and so on until the plane is covered, and then proceed to the next horizontal plane above. With the indices assigned in the opposite order i, j, and k to the levels of a basic form, the program would first scan a column, then the column behind it, and so on until a vertical plane is covered, and then proceed to the adjacent vertical plane.

In most rearranging procedures one may distinguish the two fundamental elements of selection and distribution. With a selection procedure one would select the elements from their old locations in the order in which they are to appear in their new locations. With this method, to rearrange information on tapes, one would shuttle over the input tape selecting the elements in the order in which they are to be written on the output tape. With a distribution procedure, on the other hand, one would take the elements as they come in their old order and distribute them into the desired new locations. A distribution operation on tapes would thus permit the input tape to be read in a single pass but would require shuttling on the output tape. Pure selection and pure distribution permit the output and input array, respectively, to be scanned in a simple linear fashion and may therefore be said to favor the output or input, respectively.

As indicated earlier, a program to completely rearrange a three-dimensional array such as $g_{i,j,k}$ would be represented by a four-level basic form. If the indices i, j, and k are assigned to the levels of the basic form in the order in which the input is available, the program rearranges by pure distribution favoring the input. If the indices are assigned to the levels in the opposite order, that is, in the order in which the output is to be stored, the program performs a pure selection operation favoring the output.

Any other index assignment would represent a combination of selection and distribution in the rearranging process. Assigning, for instance, the "middle index" j to the rightmost level of the basic form would yield a balance between selection and distribution which favors neither the input nor the output. For rearranging on tapes, this index assignment would distribute the shuttling over both tapes and considerably reduce required tape travel. In rearranging a 3-dimensional array of n^3 elements by either pure selection or pure distribution the distance of tape travel required is in the order of n^5 spaces on the tapes. This is almost entirely accounted for by the shuttling over one of the two tapes. With the well balanced index assignment, a combined tape travel in the order of only n^4 spaces is required; the shuttling is here distributed over both tapes and furthermore allows portions of this combined travel to be performed concurrently.

For rearranging in a random-access store, a well balanced index assignment may reduce the time and equipment required for the generation of variable addresses. Consider, for example, a matrix multiplication with all three matrices stored in the same manner, by columns, for instance. Frequently the summation index is associated with the rightmost level of the basic form, that is, with the "innermost loop," and two automatic address modifiers, or "indexing registers," are employed to step the effective addresses of the matrix elements by 1 and by n respectively, where n equals the number of rows and columns in the matrices. A more efficient index assignment would associate the summation index with the second level from the right on the basic form. Such an index assignment would require only one automatic address modifier on the rightmost, high-frequency level of the basic form. In addition, it would tend to yield a faster program than one employing two automatic address modifiers, since only one rather than two modifiers would have to be stepped on the high-frequency level.

The term "rearranging" has been used here to designate processes which transfer data from one state of order into another state of order. The term "sorting," by contrast, will be used to designate processes which transfer information from a state of disorder, with respect to the current purpose, into a state of order. Rearranging and sorting may be distinguished in two ways. In rearranging, on one hand, the new addresses of the data as a function of the old addresses can be represented by a formula, and the data are identified implicitly by their addresses. In sorting, on the other hand, the new addresses as a function of the old addresses can be represented only in the form of a table, and the data are identified explicitly by identification codes, "control numbers," or "keys," carried along with the data.

In rearranging, the formula giving the new addresses as a function of the old addresses is known and is used in designing a rearranging procedure. In sorting, the address table giving the new addresses as a function of the old addresses is usually unknown, and the sorting process is frequently subdivided into two distinct parts. The machine would first extract from the data in their state of "disorder" the necessary information to construct the address table. During the second phase of the sorting process the data would then be moved according to the specifications represented by the address table. In both phases of sorting, some technique of indirect addressing is usually employed.

Sorting, like rearranging, may be performed implicitly and concurrently with computing or else explicitly as a separate operation. When sorting is performed explicitly on tapes one might again reduce the required tape travel by an efficient combination of selection and distribution. If the address table giving the new locations as a function of the old locations is used in the order by the old addresses, the program would perform sorting by pure distribution. If, conversely, the data are moved chronologically in order by their new locations, sorting would be performed by pure selection. For a more efficient sorting operation requiring less tape travel one might reorder the lines of the address table so that when the data are later moved in the chronological order indicated by the table the burden of traveling will be shared by both tapes.

Present tape-storage systems usually permit information to be read in any order desired but require data to be written on tapes into consecutive spaces. Such tape systems permit only pure selection procedures to be implemented and thus preclude the application of potentially more efficient methods for rearranging and sorting.

4. The program as a transceiver

The following example will illustrate how a program may be considered a receiver and transmitter of data. Assume that a vector b_j is to be multiplied by the rows of a matrix a_{ij} to yield the vector c_i according to the formula

$$c_i = \sum_{j=1}^{n} a_{ij} b_j.$$

The input vector b_j is given on a tape. The matrix a_{ij} is given on another tape in order by columns, that is, in the order $a_{11}, a_{21}, \ldots, a_{n1}, a_{12}, a_{22}, \ldots, a_{n2}, \ldots, a_{1n}, a_{2n}, \ldots, a_{nn}$. The output vector c_i is to be written on a third tape. Individual elements, rather than blocks of numbers, are written on and read from the tapes.

An area of n cells, capable of holding one vector, is assumed to be available as work space in memory. Figures 5 and 7 represent alternative programs with different index assignments. Address-computing, setting, and terminating procedures are not shown on the flow charts; procedures for these purposes are implied by the index assignments. In either program the productive computing, represented by the formation of a product $a_{ij}b_j$ and the addition of this product to a progressive sum Σ, is performed in the box in the center of the rightmost level.

In the first program, shown in Figure 5, the summation index j is assigned to the rightmost level of the basic form. The elements c_i of the output vector are produced one at a time, that is, the progressive sum representing one c_i is completed and written on the output tape before computing the next c_i is commenced. The complete input vector b_j is read into the memory area of n cells on the open level at the start of the program. The elements a_{ij} of the matrix are read one at a time on the j-level. Successive elements a_{ij} with consecutive values of j for a given i are located n element spaces apart on the a_{ij} tape. For the transition from each i to the next, the matrix tape is backspaced on the i-level by a distance in the order of a matrix space on tape. The tape travel for reading the elements of the matrix is thus in the order of n passes over the matrix tape.

In the second program, shown in Figure 7, the summation index j is assigned to the second level from the right. With this index assignment, the memory area of n cells is used to hold the output vector c_i. Since the index i is assigned to the rightmost level, the computations proceed "by i's within j"; that is, n progressive sums Σ for the n output elements c_i are built up in parallel in the memory area. During the first n executions of the i-level, for $j=1$, the first contribution to each of the n progressive sums is made; during the second n executions of the i-level, for $j=2$, the second contributions are made, and so on. All elements c_i of the output vector are written on the output tape at the end of the program on the open level. The elements b_j of the input vector are read one at a time on the j-level. The elements a_{ij} of the matrix are read one by one on the i-level. Successive elements a_{ij} with consecutive values of i for a given j are located in adjacent element spaces on the a_{ij} tape. The tape travel for reading the elements of the matrix a_{ij} required with this index assignment is therefore a single pass over the matrix tape.

The two programs for the same problem use the same memory and storage capacities. One vector space is used in memory. Two vector spaces and one matrix space are used on three tapes. The tape travel required for reading the matrix, which may be considered a measure of program execution time, however, amounts to approximately n passes over the matrix tape in the first program but only to a single pass with the second program. This substantial difference in the efficiency of the alternative programs for the same purpose may be interpreted in terms of selection and distribution employed for the implicit rearranging performed by the program. The index assignment in the first case represents a selection operation which favors the output in the sense that the elements c_i of the output vector are produced one by one in the order in which they are to be written on the output tape. The second index assignment represents a distribution operation in the sense that the output elements c_i are built up in parallel by distributing the contributions over the n progressive sums in memory. This second index

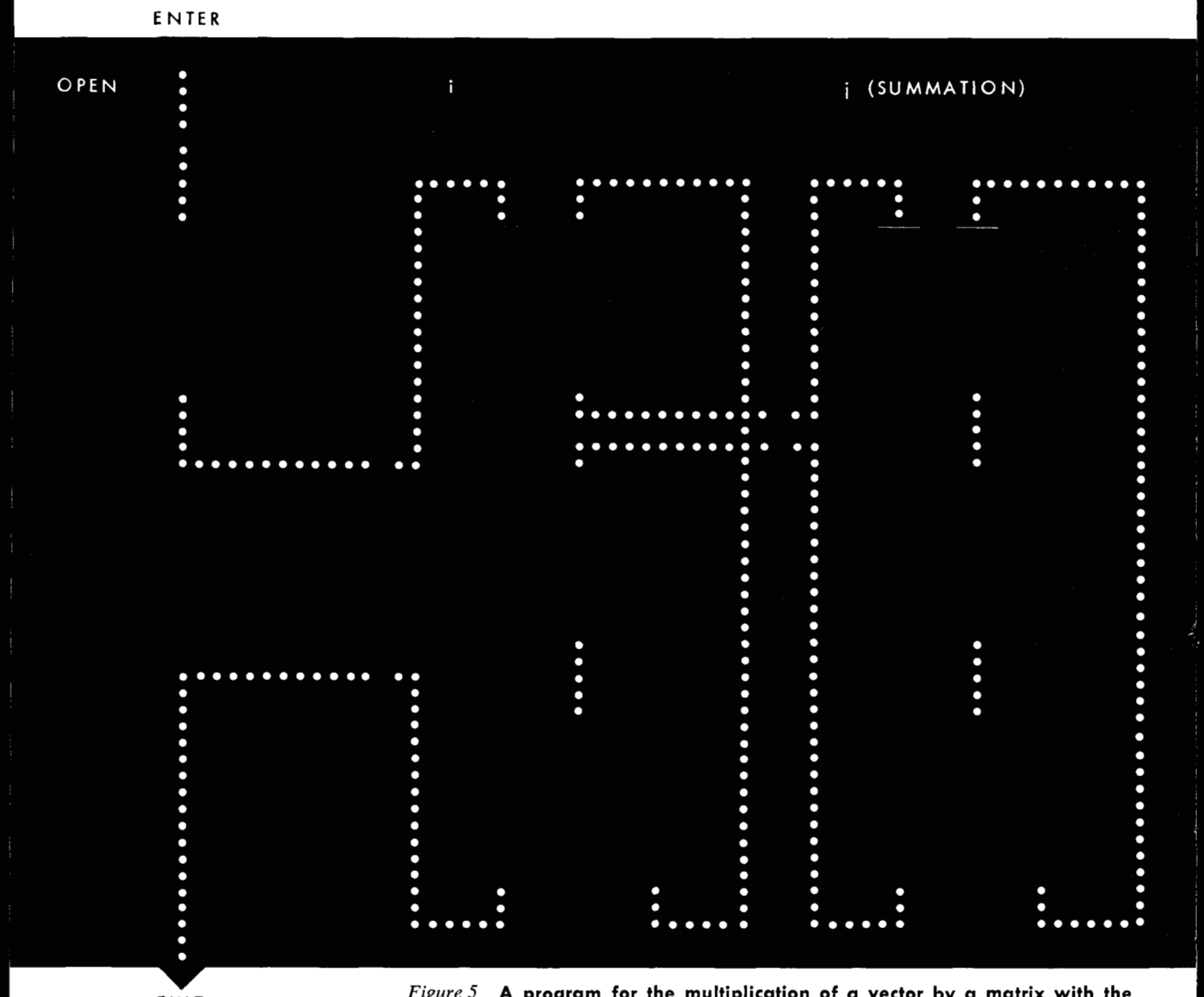
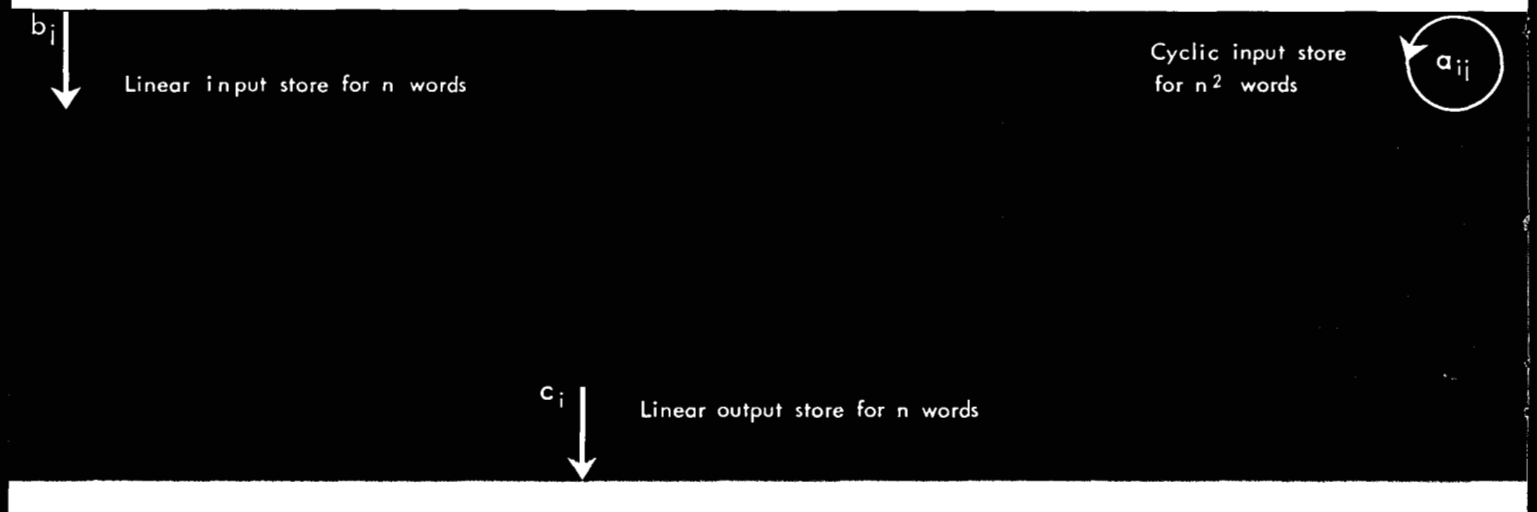


Figure 5 **A program for the multiplication of a vector by a matrix with the summation index assigned to the rightmost level of the basic form.**

Figure 6 **The program above shown as a receiver and transmitter of data.**



assignment, representing a distribution operation, favors the input. The volume of input data, comprising a matrix, is substantially larger than the volume of output data, and it is seen that the index assignment which favors the larger volume of input yields a considerably more efficient program than the index assignment which favors the smaller volume of output.

In either program, progressive sums are completed and become available to be written as output on a tape at the time program execution returns from the level associated with the summation index to the level on its left. The accumulation operations are in either case performed on the rightmost level. When, as in Figure 5, the summation index is assigned to this level, one completed sum becomes available as output during each execution of the center level. When, as in Figure 7, the summation index is assigned to the center level, which is in this case "backed up" by the rightmost level with a relative frequency of n, the entire output consisting of n completed sums becomes available simultaneously on the open level. Visualizing a basic form as a "black box" producing output data, one may consider output to be transmitted from a "slot" on a certain level of the basic form. Similarly, one may visualize "slots" on the levels through which the basic form receives input data.

The basic form in Figure 5 is pictured as a receiver and transmitter of information in Figure 6. The elements b_j of the input vector, which are read on the open level in a linear fashion, are received by the basic form from a linear input store whose capacity is n words. Similarly, the c_i emerging on the center level in a linear fashion may be considered to be transmitted from this level of the basic form into a linear output store. The matrix elements a_{ij} are read on the rightmost level of the basic form. After n elements have been read, the matrix tape is backspaced, and another n elements are read. This repetitive cycle of reading and backspacing may be considered to simulate the function of a cyclic, or "recirculating," store. If the matrix were given, for instance, on a closed-loop tape, which represents a cyclic store, the cycles of reading and backspacing could be replaced by reading repeatedly around the loop in the forward direction. The matrix a_{ij} may thus be considered to be received by the basic form from a cyclic input store whose capacity is n^2 words. The memory area of n cells is pictured as a cyclic work store inside the basic form. This store holds the input vector b_j whose elements are scanned repeatedly in the cyclic order $b_1, b_2, \ldots, b_n, b_1, b_2, \ldots, b_n, \ldots, b_1, b_2, \ldots, b_n$.

The basic form with the alternative index assignment represented by Figure 7 is pictured as a receiver and transmitter of information in Figure 8. A linear input store and a linear output store for the vectors b_j and c_i are again shown for the corresponding levels of the basic form. The cyclic work store inside the basic form is in this case used for the n progressive sums Σ which are referred to cyclically in the distribution operation. The matrix a_{ij} is read in a single pass over the matrix tape and therefore shown as being received from a linear input store with a capacity of n^2 words.

A comparison of Figures 6 and 8 shows equivalent situations with respect to the input and output stores for the vectors as well as the cyclic work store. The stores required for these purposes are in either case two linear stores and one cyclic store with a capacity of n words each. The input store holding the matrix a_{ij}, however, is in the first case a cyclic store with a capacity of n^2 words and a rate of flow of n^3 words per execution of the program. In the second case, the matrix is received from a linear store with again a capacity of n^2 words but the lower rate of flow of only n^2 words per execution of the program.

As pointed out earlier, the second index assignment would yield a more efficient, that is, faster, program for an existing machine. From the point of view of machine organization, the alternative index assignments provide a choice between two pieces of equipment for the same purpose. If a linear store with a certain capacity and rate of flow is assumed to be less expensive than a cyclic store with the same capacity and n times this rate of flow, the second index assignment would also suggest the more efficient machine organization.

5. Stores

Three types of stores may be distinguished according to the technical characteristics of the equipment: random-access, cyclic, and linear stores. In random-access stores, such as magnetic cores, the time required for an access is completely independent of the particular location referred to. In cyclic stores, such as drums or disks, each location becomes available periodically. In linear stores, such as tapes or decks of cards, the information is arranged along an essentially one-dimensional carrier medium.

Linear stores are usually operated on a start-stop basis in order to synchronize their effective rate of flow with the speed of data processing. In a typical application of tapes, for instance, one would read a block of data, let the tape come to rest, process these data, and, at the time processing of the data is completed, start the tape again for an access to the next block. In this manner tape operation is synchronized with processing by means of the program which provides the tape-handling instructions at time intervals whose lengths reflect the rate of data processing. With cyclic stores, on the other hand, the cycle of the store is usually not synchronized with processing. Instead, a cyclic store is usually employed asynchronously and considered a pseudo-random-access store. The pseudo-random-access time of a cyclic store used in this manner might be defined generously as the period of the store which constitutes an upper bound for the access time to any location. More commonly, however, it is defined on a random basis as half the period of the cyclic store or as an even shorter period taking into account some limited "optimization" of address assignments. In certain applications, address assignments for

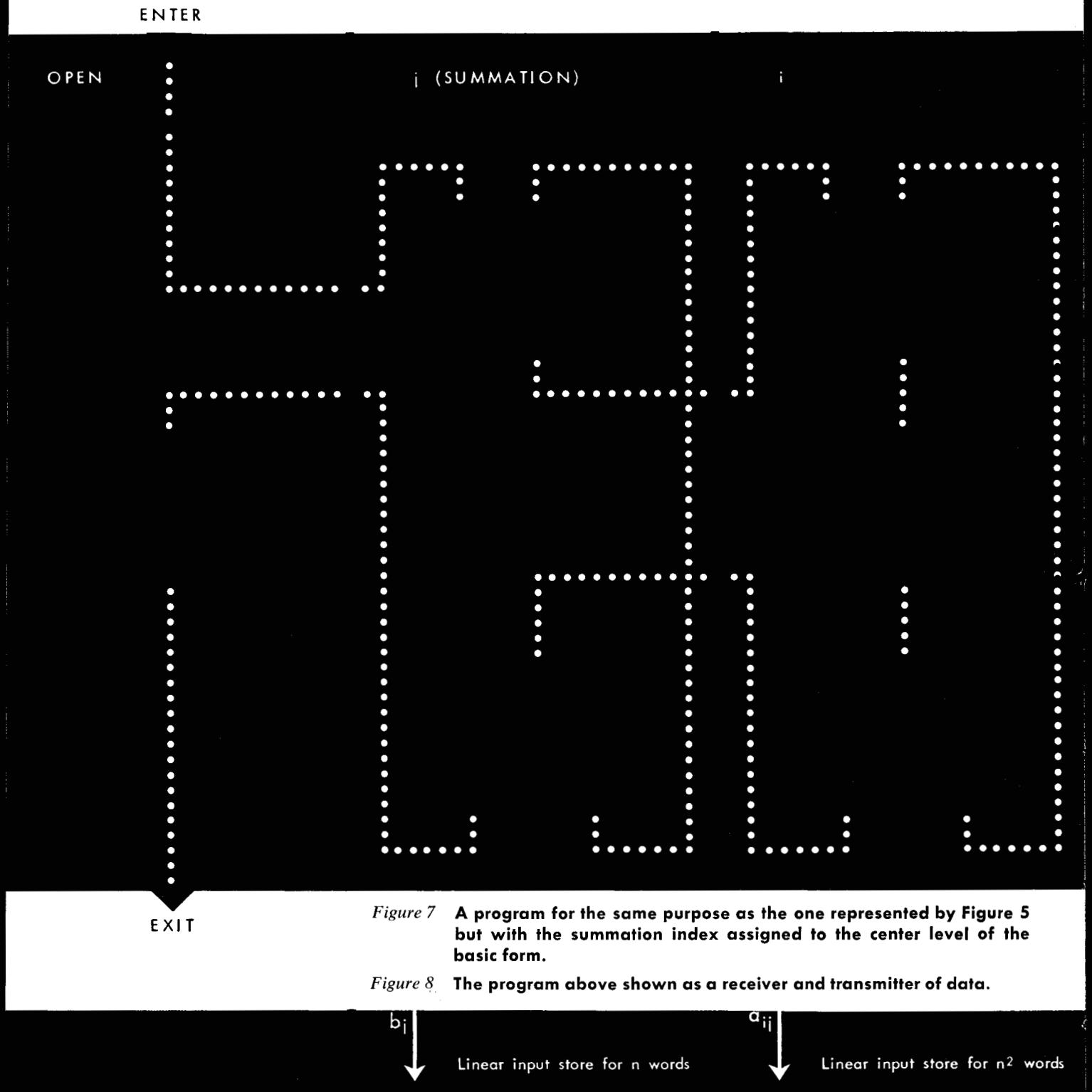


Figure 7 **A program for the same purpose as the one represented by Figure 5 but with the summation index assigned to the center level of the basic form.**

Figure 8 **The program above shown as a receiver and transmitter of data.**

pseudo-random-access stores are not "optimized" with respect to the access times required for data in the cyclic store but rather to reduce the time required for generating the addresses to be inserted into the instructions of the program as in the case of "randomized" address assignments.

In many problems, data are handled in units referred to as "packets of words," and the packets, in turn, are commonly combined into "blocks" for storage. A packet may correspond to an individual business record in a commercial application or to all the data pertaining to a certain point of a space-mesh in a scientific or technical problem. A block, comprising a number of packets, might then correspond to a group of business records or a set of points along a row of a mesh.

In typical applications of data-processing systems one may consider the random-access memory as a buffer between the processing unit of the system and storage devices, such as tapes, drums, and disks. As long as the problem requires only rearranging in the broad sense, including the frequently present implicit rearranging discussed in Section 3, random-access memory is used to simulate cyclic and linear buffers with respect to packets within blocks. Random accesses are in this case required only within the single packet or the small set of adjacent packets currently being processed. Only when sorting, including implicit sorting, is required for packets, the random-access property of memory is exploited also with respect to packets within blocks, i.e., over substantial areas in memory rather than over trivially small portions of memory corresponding to individual packets.

For a concrete example of the use of memory as a cyclic buffer one may refer to the cyclic work store discussed in Section 4. The efficiency of the alternative schemes for the same purpose may be interpreted in terms of the exploitation of random-access memory for the simulation of a cyclic work store. If one considers this problem of multiplying a vector by a matrix a tape-limited operation, the total machine time for the problem will be approximately proportional to the travel required of the matrix tape. In this sense, the first program will be executed during n units of time whereas the second program for the same problem will be executed within only one unit of time. The simulated cyclic buffer, in the first case for the b_j and in the second case for the Σ_i, will in either case complete n cycles of n words each. Since the total duration of program execution is n times larger in the first case, the rate of flow of the cyclic work store is n times higher in the second case. With the second, more efficient scheme, the random-access memory is thus more efficiently exploited in the sense that it is programmed to simulate a cyclic store which is n times as fast as the buffer simulated with the first approach.

This example also indicates that in simulating a cyclic or linear store in random-access memory, one does not utilize the random-access quality of the memory since the spatial distribution of the accesses to the simulated store is perfectly regular and not at all random. For this purpose one utilizes only the fact that memory will retain a piece of information for an indefinite period of time and yet make it available instantaneously when required. It is this quality of memory, rather than its random-access property, which permits the simulated buffer store to be synchronized by the program with the rate of processing.

The preceding observations may be summarized as follows. In many data-processing systems an expensive random-access memory, such as a magnetic-core store, is provided. In many typical applications, this random-access memory is "converted" by programming into linear and cyclic stores which are synchronized with the processing unit of the system. This simulation of work stores does not utilize the random-access quality of the memory. The programmed implementation of this simulation is frequently facilitated by incorporating additional equipment for automatic address-modification in the system. When a magnetic drum is used for memory, one often observes the following twofold transition between types of stores as far as the utilization of the drum is concerned. The drum is a fast cyclic store operated asynchronously with respect to the rate of processing and is considered a pseudo-random-access store. This pseudo-random-access store, in turn, provides the basis for programmed implementation of simulated cyclic stores which are synchronized with the processing unit of the system.

For an example of how expendable capacities affect the speed of rearranging, consider the case of rearranging a two-dimensional array on tapes. The elements of the array are assumed to be given in blocks on an input tape where each block contains a row of the array; the elements are to be written on an output tape by blocks corresponding to the columns of the array. The rearranging process is assumed to be performed by a conventional selection method so that the tape travel on the input tape may be taken as the primary measure of the required machine time. For different programs using different amounts of random-access memory for buffer areas as well as additional true linear stores, that is, work tapes, one then finds the following approximate relations. The machine time required for the rearranging process is a linear function of $1/M$, where M is the total memory capacity available for the simulation of linear buffer stores. The required machine time is also a linear function of $1/T+1$, where T is the number of available true linear stores of virtually unlimited capacity, that is, the number of tapes available for work tapes in addition to the input tape and the output tape.

6. Arithmetic units

Most data-processing systems at present have only one arithmetic unit but several stores. There appear to be two main reasons accounting for this fact. Many problems, especially computational ones, are believed to be of a sequential nature in the sense that usually one quantity must be computed before one can proceed to computing the next quantity. From a technological and economical point of view, it is widely believed that fast arithmetic units are more readily obtainable than fast stores, and

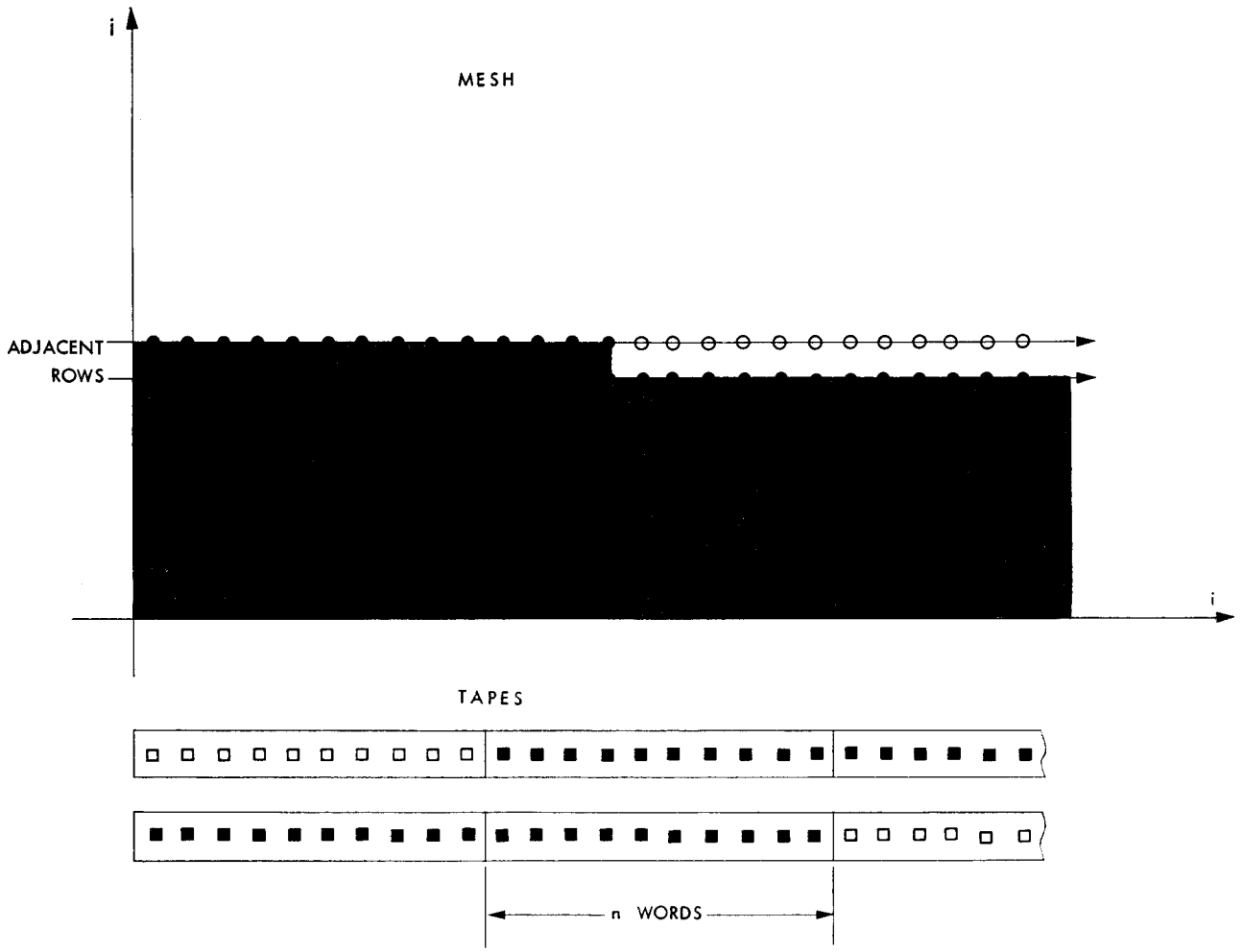


Figure 9 **Mesh of a network problem covered row by row from left to right, and the two tapes holding the data for two adjacent rows.**

that therefore one of the most serious problems in machine organization is to provide enough adequate stores to keep even a single arithmetic unit busy. However, it can be seen from the following example of alternative programs for a typical network problem that these two motivations for having only one arithmetic unit in a system may not be valid for large and important classes of data-processing applications.

Assume that the values of a function $\phi_{i,j}$ are to be computed for the nodes of a two-dimensional mesh according to the formula

$$\phi_{i,j}=f(\phi_{i-1,j}, \phi_{i,j-1}).$$

With the i- and j-directions being horizontal and vertical, respectively, the formula requires for the computations at a point (i, j) the data at the neighboring point $(i-1, j)$ to the left and at the neighbor $(i, j-1)$ below. The values of ϕ are given along the bottom row $j=0$ and the leftmost column $i=0$. The purpose of the computations is to produce the values of ϕ along the top row $j=j_{\max}$. The values of ϕ at internal points of the mesh, which are to be computed as intermediate results, may be discarded as soon as they have served their purpose of advancing the computations toward the top row. Two alternative programs will be considered for this problem. The first program causes the computations to proceed along the horizontal rows of the mesh, the second program traverses the mesh by segments of diagonals.

Figure 9 illustrates the first program. The values along a row are given on a tape in blocks of n numbers each. After having read such a block into a memory area of n cells, the corresponding n points in the adjacent row above are computed. They replace the "old" values in the memory area from which they are then written on the other tape. In this manner the program proceeds along a row, block by block. At the end of each row both tapes are rewound. The previously "old" tape then becomes the "new" tape to be read, and the other tape is available for writing the new current row. The tape travel for reading and writing required by this program would amount to two tape passes per row.

The alternative program is pictured by Figure 10. In

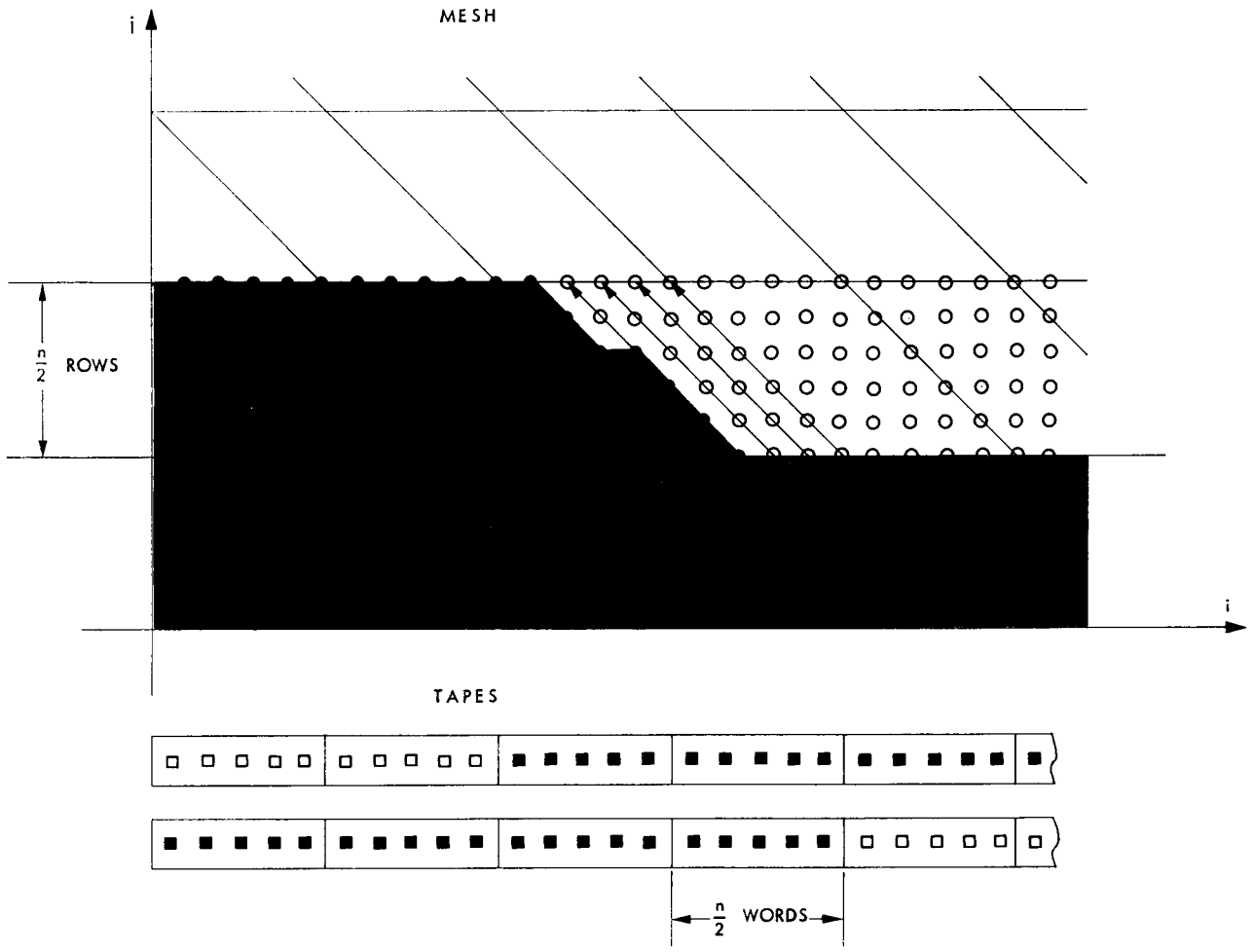


Figure 10 **Mesh of a network problem covered by horizontal layers of parallelograms, and the two tapes holding the data for the bottom rows of two adjacent layers.**

this case the mesh is covered by diagonals within segments of the plane in the shape of parallelograms. Assume the data along the bottom of such a segment read as a block from a tape into a memory area of $n/2$ cells. Also assume the data along the leftmost diagonal of the segment given in another memory area of $n/2$ cells. The computations can then proceed along the first internal diagonal of the segment from the bottom to the top since the lefthand and lower neighbors of all these points are available. The new diagonal will replace the old one to its left in the memory area of $n/2$ cells. The value on top of the diagonal replaces its counterpart at the bottom of the diagonal in a cell of the other memory area which is associated with the tapes. The program then traverses the next diagonal to the right. In this manner the entire segment is covered. The top row of the completed segment is then written on the current output tape and the bottom row of the adjacent segment to the right is read into its place in memory. After an entire layer of segments has been covered, the tapes are rewound and their functions as input and output tapes are interchanged in preparation for the next layer of segments above. The tape travel required for reading and writing with this program amounts to approximately $4/n$ tape passes per row of the mesh.

Both programs require the same number of tapes and the same memory space. In tape-limited operation the second program would require less machine time by a factor in the order of $2/n$, where n is the usually large number of memory cells available for the problem. This substantial difference in the efficiency of the alternative programs, which is a function of the expendable memory space, may be interpreted in terms of the exploitation of memory for simulated buffers (see Section 5). In the first case the input-output area in memory may be considered a buffer between the arithmetic section and the tapes with a certain rate of flow. In the second case, the half of this memory space which holds the successive diagonals may be considered a buffer store whose rate of flow is $n/2$ times greater than in the first case.

The second program suggests the following observations pertaining to machine organization. At the start of

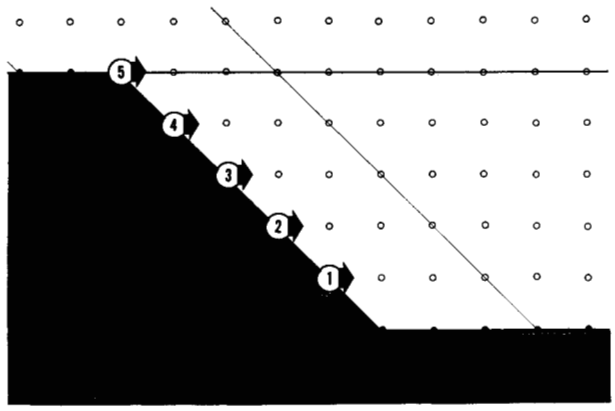


Figure 11 **Five arithmetic units working concurrently along five adjacent rows of a mesh with a horizontal delay of one point between adjacent arithmetic units.**

the computations for a diagonal, the lefthand and lower neighbors of all the points to be computed are available. If as many as $n/2$ separate arithmetic units were available in the system, one could therefore compute all points of a diagonal concurrently. This is illustrated for five adjacent rows of the mesh by Figure 11. The arithmetic units would proceed in parallel along horizontal rows as indicated by arrows. At a given time, the different arithmetic units would simultaneously compute the different points of a diagonal marked with circles. In transition to the next diagonal each arithmetic unit would make its result available to the arithmetic unit above where it will serve as the lower neighbor, and it would retain this result temporarily in order to use it for the lefthand neighbor. In this manner an entire layer of rows could be covered across the whole width of the mesh, requiring memory accesses only for the linear arrays of data along the bottom and top rows of the layer but not for the two-dimensional array of internal points in the layer.

This example illustrates the following points. The problem, although it suggests that the array of $\phi_{i,j}$ be covered systematically, is not sequential in nature since it permits computing to be performed concurrently for any number of adjacent rows. Furthermore, a large number of separate arithmetic units operating concurrently may reduce rather than increase the number of memory accesses required per arithmetic operation. This implies that each arithmetic unit is supplemented with a very small random-access store, possibly in the form of individual registers, whose capacity is in the order of a packet of data in the sense of Section 5. The main memory, which in present systems is in the order of thousands of words and would be used only for linear and cyclic buffers with respect to packets, could be designed for a substantially lower number of accesses per arithmetic operation.

7. Conclusion

The preceding examples indicate how systems organization may be studied in terms of programs. A data-processing system would be viewed as a collection of processing units and stores of different types and characteristics. The program would represent the problem in terms of such system components and thus provide a link between the problem and the machine in terms of which both the applications and the system may be studied.

With a sufficiently powerful representation of programs one might be able to make the problem itself "say" in these terms what kind of a system it wants to be solved on. This might permit the process of systems design to be eventually systematized and ultimately mechanized.

This approach might also lead to more efficient methods of programming for existing systems which would lend themselves better to systematization and automation. The process of programming for existing machines might be subdivided into two distinct phases. The first would be identical with the process of systems design in the sense indicated above; it would produce the specifications for a hypothetical system which is optimum with respect to the particular problem under consideration. The second distinct phase of such a programming process would then consist of realizing the hypothetical optimum system on an existing data-processing system with the primary objective of preserving the optimum qualities of the hypothetical system as far as possible.

Future research might be conducted primarily in the following areas: One would wish to know how a problem can be subdivided into separate parts which can be operated upon independently and concurrently. For a description of a problem in terms of such parts one would wish to know in which chronological order the parts should be processed so as to minimize systems requirements. These requirements might be minimized with respect to the volume of data to be retained during the course of problem solution, with respect to the amount of rearranging and sorting required by the problem solution, and with respect to the cost of all processing units and stores of the system other than the one which limits the performance of the over-all system.

Acknowledgment

The material presented in this paper has been developed at the Watson Research Laboratory by the author in collaboration with present and past members of the laboratory staff. Particularly significant contributions to the subject matter have been made by Dr. D. H. Tycko and Miss A. T. Flanagan.

Received October 14, 1957