FOREWORD

This report, performed under U. S. Air Force Contract No. AF 33(657)-7391, describes results of a study conducted for the Air Force Avionics Laboratory, Research and Technology Division, Wright-Patterson Air Force Base, Ohio. Mr. D. J. Boaz was the project engineer.

The contractor participants were drawn from the Information Systems Laboratory, Department of Electrical Engineering, The University of Michigan. Those engaged in the research were: Dr. Harvey L. Garner, project director, Rodolfo Gonzalez, Sandra Palais, Thomas F. Piatkowski, and Jon S. Squire, with frequent and useful collaboration from the rest of the members of the Information Systems Laboratory.

The work reported was performed during the period October, 1961, to December, 1963.

This is the final report on the contract.

ABSTRACT

The main objective of this research has been to investigate the problems and possibilities generated by the idea of a computer built as an iterative array of elementary self-contained processors.

Since the problem was a new one, a large number of areas of study were available, and the following were treated in detail:

1. Programming aspects:
   a. Decomposition of a program in order to obtain maximum concurrency.
   b. A translation algorithm to facilitate programming.
   c. Algorithms for path-building.
2. Proposed organizations:
   a. A multi-layer computer.
   b. An iterative circuit computer with n-dimensional geometry.
3. Statistical evaluation of accessibility for different geometrical structures.
4. Reliability problems and new possibilities.
5. A theoretical model, linking the iterative circuit computer structure to that of an n-head automaton.

Due to the variety of topics covered, each section has been provided with its own introduction, leading the reader to the problem and providing the necessary relationship with previous work. In this way, each section provides a comprehensive treatment of the title subject in a self-contained form, allowing for independent reading.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Concluded)

LIST OF TABLES

1. INTRODUCTION

It appears that the ever present need for faster and more efficient computers will continue. In the past, computer advances have been obtained by improvements in physical devices which have allowed faster operation and greater logical complexity. While component advances will continue there is no projected development, with the possible exception of coherent optics, which will obtain a speed increase of many orders of magnitude within the framework of conventional machine organization.

The subject of parallel computation is not new. Examples of parallelism and concurrency are found in early computer designs. The ENIAC obtained increases in basic speed over electro-mechanical computers by electronic circuitry and also employed a parallel arithmetic structure consisting of 20 accumulators. Difficulties in programming this machine lead to the stored program concept in conjunction with a single high-speed arithmetic unit as seen in the Princeton class of machines, or the EDVAC and its descendants. This organization has been universally used until recently when again computational requirements are beginning to exceed the capability of present machines. The most recent generation of computers, LARC, STRETCH, GAMMA 60, Bendix G20, and the RW 400, employ either concurrency of parallelism to a limited extent. At least one computer being constructed, but not yet delivered, has a processing unit capable of simultaneously executing many arithmetic operations. It would appear that the question in the area of computer organization is not whether

Manuscript released by authors April, 1964, for publication as an ASD Technical Documentary Report.

parallel machine organization is needed, but rather what should the parallel organization be to obtain effective and economical computation.

The Iterative Circuit Computer (I.C.C.) concept was developed abstractly by John Holland of The University of Michigan. The mathematical existence of this class of machines was required for Holland's[37] studies in the theory of adaptive systems. The original abstract specification is sufficiently broad so that all previous machines, both theoretical and real, can be represented in the I.C.C. framework. In the theory of adaptive systems, the I.C.C. provides a homogenous, isotropic medium for imbedding programs which simulate some physical systems. The interaction and modification of programs in the I.C.C. media correspond to the interaction and changes in the physical system.

In our current research we are evaluating the I.C.C. concept with respect to the organization and the analysis of organization of practical computing machines. In this respect, the I.C.C. appears to represent a reasonable abstract structure for the study of problems related to the organization of parallel computers. We also seek to determine whether some form of the I.C.C. concept provides a basis for the design of a practical parallel computing machine. In this paper, some of the preliminary results obtained from studies considering the I.C.C. as a real computer are presented. These studies have been undertaken to determine the effectiveness and the nature of the limitations of the I.C.C. concept. It should be emphasized that our research is being conducted to learn more about the I.C.C. concept. At this time, we are not in a position to either encourage or discourage the concept of a practical I.C.C.

Ultimately, the I.C.C. as a practical computer concept must rise or fall on the basis of application. The apparent problems of the I.C.C. concept can be summarized by four pertinent questions: (1) What are the programming difficulties? (2) What are the path building limitations? (3) Can large numbers of components be used reliably? (4) Can the components in the I.C.C. organization be used efficiently, and how many components are required for an effective computer for a given problem or class of problems? Aspects of the concept which seem extremely favorable are the possibility of economical manufacturing of iterated structures and the degree of local control inherent in the Holland I.C.C. structure.

It is the degree of local control which distinguishes the Holland I.C.C.[8,30] from other I.C.C.'s such as the Ungar machine,[9] McCormick's machine,[38] or the SOLOMON.[16] In the Holland I.C.C., control must be established by the program. Thus, the degree of logical or global control is variable and controlled by the user. There is no doubt that parallel computation requires some local control. The pertinent questions which must be answered in detail are whether effective use can be made of the high degree of local control which is obtainable, and whether the price which must be paid to obtain variability in the degree of control is reasonable. A high degree of local control was required for the abstract studies of adaptive systems. It is expected that the development of a highly parallel computer and the development of techniques for applying such machines to problems not effectively solved on existing machines will necessitate variable local control. The need for local control in conventional problems is being studied. The question of the need for local control is critical to the evaluation of the Holland concept.

Path building and iterative structure are also fundamental to the Holland I.C.C. concept. Detailed considerations such as the number and types of instructions and the path building mode are pertinent to a detailed design study, and they are treated in Sections 2.1 and also when two proposed organizations are presented in Sections 3.1 and 3.2.

The actual number of instructions that can be executed simultaneously is limited severely by the geometry of the machine. Maximum accessibility is gained when the modules are interconnected as if they were located at the vertices of a n-dimensional cube. Comparative studies for typical problems are presented in Section 4.

The large amount of hardware implicit in any iterative structure brings an old problem into a completely new environment; the large number of elements in itself seems to increase tremendously the probability of failure, while at the same time, the very redundant structure allows for several new schemes, totally impractical in machines with a standard organization.

While the redundancy can still be introduced internally at the module level, it is much more significant that the availability of "extra" modules not busy with the actual calculations permits the introduction of a checking program, which "circulates" over the array, testing each module in turn.

The problem is thus changed radically; now it is a matter only of guaranteeing the simultaneous availability in a working state of a number of modules for the short interval between two scans by the test program.

Section 5 treats this problem and presents calculations on the expected scan time between failure for a practical size machine using microelectronic elements available at this date.

In a new field like this, where there exist no practical guidelines or previous experience, it is perfectly admissible to make assemptions about number of components, mode of interconnection, etc. But if we expect to solve the problems of how to program effectively such a machine, what types of problems are more suitable, what advantages can be expected, etc., a formal mathematical model becomes a necessity.

One such model is presented in Section 6, along with comments on the practical consequences and limitations of the theory. The material presented in this report has appeared or has been the base for the following publications:

"Iterative Circuit Computers," by H. L. Garner and J. S. Squire, Proc. of the Workshop on Computer Organization, Spartan Books (1963).

"A Multi-Layer Circuit Computer," by R. Gonzalez. Presented at the 1963 ACM National Conference. Also to appear in the IEEE Trans. on Elect. Computers, Special Issue on Machine Organization, December, 1963.

"Programming and Design Considerations of a Highly Parallel Computer," by J. S. Squire and S. M. Palais. Presented at the Spring Joint Computer Conference, May, 1963.

"A Translation Algorithm for a Multi-Processor Computer," by J. S. Squire. Presented at the ACM National Conference, August, 1963.

"n-Head Finite State Machines," by T. F. Piatkowski, Ph.D. Thesis, The University of Michigan, December, 1963.

2. PROGRAMMING ASPECTS OF MULTI-PROCESSOR COMPUTERS

2.1 PATH BUILDING PROCEDURES

2.1.1 Introduction

The need for increased computational capabilities, brought into evidence when dealing with problems in the fields of pattern recognition, game playing or simulation of physical models, has suggested the use of highly parallel iterative circuit computers.[8-10] Furthermore, the recent advances in the technology of micro-miniature, integrated and "grown" electronic devices show promise that in the near future the availability of low cost modules essential for the practical realization of this type of machine will make even more pressing the need for detailed studies of this new concept.

A study of the pertinent literature[8-10] reveals that increase in versatility is accompanied by the introduction of several new problems inherent in this novel machine organization. Some of these problems are:

(a) Data allocation difficulties due to the "floating" address, as used in Holland's paper.[8]

(b) Programming difficulties due to the unlimited interaction possible between concurrently running programs.

(c) Necessity of some allocation protection method due to the presence of several program running simultaneously.

(d) Problems in the flow of control due either to the lack of a centralized organ of command or to its impractically enormous complexity.

(e) Problems in programming and in interconnections generated by the lack of continuity of the geometrical properties of the space over which the machine is spread.

A number of these problems appear only in some of the proposed machine organizations, but others are germane to the very essential features of the

general class of I.C.C.'s. Therefore, these latter problems deserve especial consideration and detailed study. The flow of information and control, as determined by the procedures used to connect operators and operands, is one of the factors that lies at the crux of the successful operation of I.C.C.'s. One approach to this problem is given in Holland's paper.[8] In Holland's theory, access to new operands is gained by adding or deleting modules from the termination of a fixed path. This method is essentially a counting procedure along a direction specified by the current instruction, and requires a small amount of hardware to implement it. This advantage is offset by the long time needed for path building since the procedure is a sequential one, and by the fact that the time needed for completion of the path-building phase is a function of the relative position of the modules to be connected.

At the other extreme, one could suggest a method using coincidence of addresses detection, resulting in a very fast but expensive procedure. As in many physical situations, speed is traded for complexity since these are the only two factors that can be rearranged in this case.

A different technique would be to generate a fresh path from each successively active module, and to leave the already used paths connected for possible future use. The potential gain in speed could, however, be offset by the difficulty in finding a new connection through all the pre-existent paths. An algorithm for path tracing is presented for the case of paths of some restricted shapes. In many cases, a situation will be reached in which paths belonging to one or more programs will form an obstacle such that some region of the network becomes isolated.

In this report, attention is given to the problem of path interference as related to the question of generation of barriers. An algorithm is also given for the detection of barriers formed by paths of any shape.

2.1.2 General Outline of the Path-Building Problem

A network of modules arranged in an n x n grid is given. The modules are all alike and contain a switching network (Fig. 1) and four registers that can be connected to any of the four outputs. Figure 2 indicates these for one of the registers.

The problem consists of successively connecting pairs of modules leaving intact a fixed number of the previously established connections for possible future use. This is referred to as path building. The end modules, i.e., the pair connected, are the originators and receptors of information and are called the terminal modules. The intermediate modules in the path serve only as connections and these are called the connecting modules. See Fig. 3.

The connecting modules can be in any of the nine states shown in Fig. 4 (a through i). A terminal module of one path can be at the same time a connecting module for another path, as shown in Fig. 5. Here the shaded squares indicate different states of the connecting and terminal modules. A pre-existent path may be crossed by a new one, and in general, all the possible connections which the internal structure of the modules will allow can be made as long as the independence of the paths is maintained.

Therefore, any connecting module can be part of as many as two different paths, and any terminal module can be associated with as many as four different paths, as the connection diagrams of Fig. 5 show. In this and subsequent diagrams, the modules are shown contiguous to each other, but this is simply an illustrational convenience not implying any other connections between the individual modules.

The paths are composed of segments, these being defined as the connecting lines between adjacent modules. However, since the internal configuration of the modules is not shown on the diagrams the connecting segments are con-

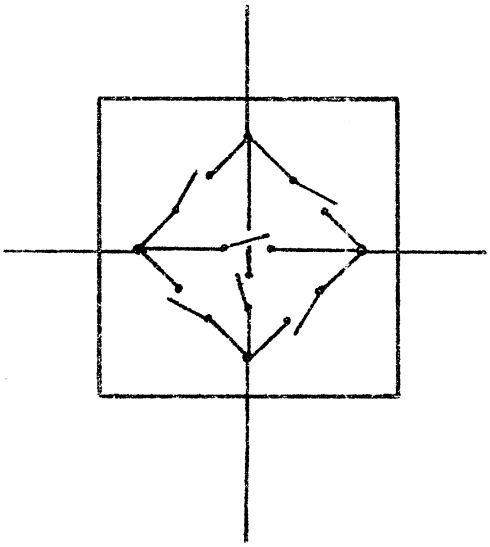


Fig. 1. Internal switching network.

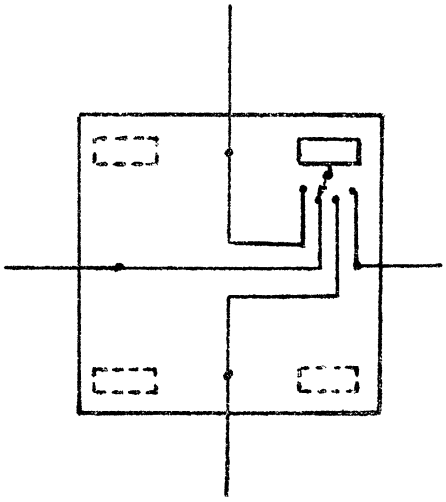


Fig. 2. Internal registers.

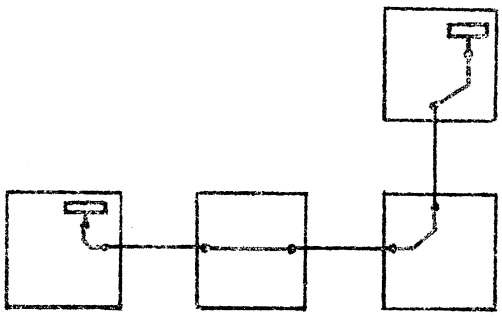


Fig. 3. Terminal and connecting modules.

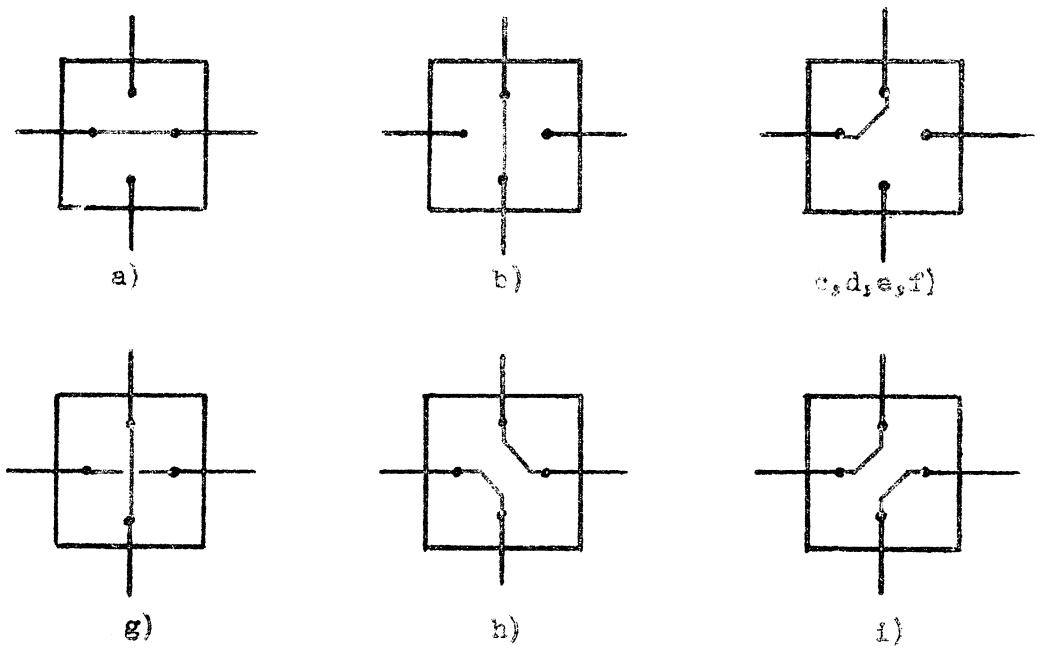


Fig. 4. States of connecting modules.

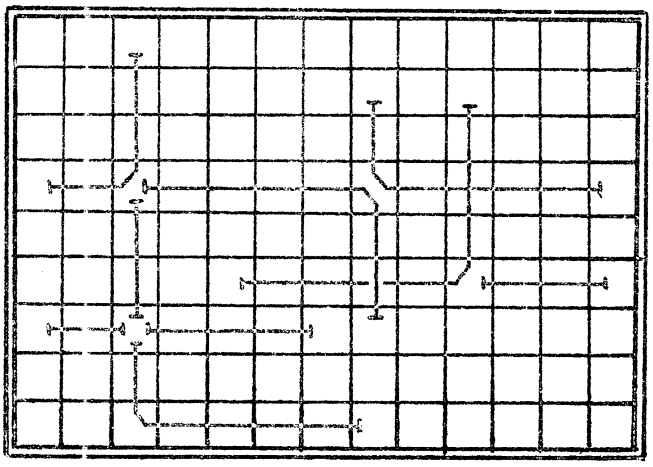


Fig. 5. Modules belonging to several paths.

sidered to extend from center to center of the modules.

As the number of pre-existing paths increases, it becomes more and more difficult to find a path connecting two given modules. The difficulty is caused by the accumulation of "obstacles" of various types. In some cases, these obstacles (previous paths) combine in such a way that a region of the network becomes isolated from the rest of the modules. In other cases, a single path of special shape suffices to isolate a region. Whenever this happens, we call the isolating line a barrier. Thus, a barrier is defined as a path or combination of path segments such that because of its particular shape and neighborhood relations it divides the whole network into two disconnected regions. In other words, it becomes impossible to connect two modules lying on opposite sides of the barrier. However, it is not easy to show the barrier as a physical entity because it entails geometrical shape as well as a positional relationship of the path or paths.

In general, if a path or a combination of path segments divide the network into two disconnected regions, one can visualize the barrier as the set of modules through which the segments constituting the path run.

A barrier may be generated by:

(a) A single path: See Fig. 6 a, b, c, d.

(b) Several paths: Any number of paths may contribute segments to form a barrier, as in Fig. 7. Paths a and b form a barrier between B and C; paths c, d, and e form a barrier between A and C.

Since a barrier is a function of the shape of the paths and of some neighborhood relation, it becomes imperative to find a mapping such that when both conditions defining a barrier are met, then some easily detected geometrical property is simultaneously satisfied. This leads to the definition of a "dual" of a path, as explained in Section 2.1.4.

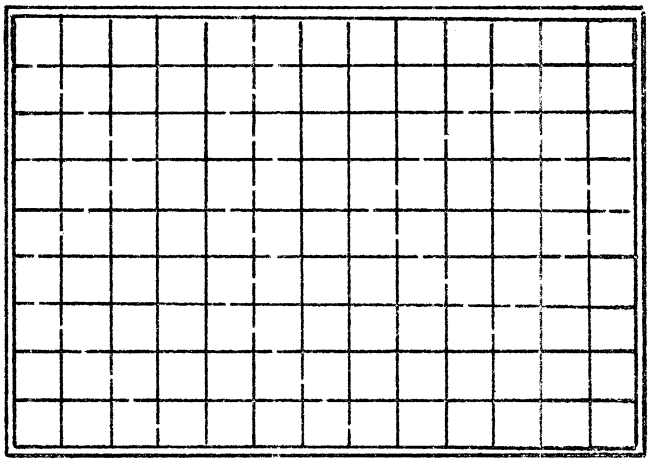


Fig. 6. Barriers formed by single paths.

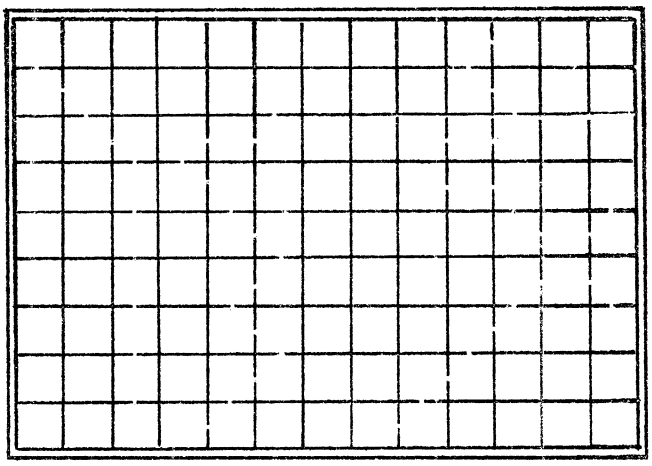


Fig. 7. Multiple path barriers.

Once this easy identification of barriers is obtained, another procedure is needed to ascertain whether two given modules lie in the same region or in disconnected regions with respect to every barrier detected up to then. If the two modules belong to disconnected regions there exists no path connecting them, and one has to resort to erasing some of the existent paths. This is done even if the allowable fixed number of paths has not yet been reached.

It must be remembered that here we are concerned with a path-finding procedure that must be repeated in its entirety for every path that is to be created. Consequently, some restriction on the types of paths to be considered is almost mandatory in order to avoid resorting to maze-solving techniques. Maze-solving techniques are essentially sequential algorithms and, as such, are much too slow for this application.

2.1.3 Definitions and Nomenclature

The purpose of this section is to present definitions and to establish uniquely the meaning of labels and names used in this report. The need for this rigorous defining of commonly used words is evident when, for example, several neighborhood relations with subtle differences have to be distinguished. Although the language is rich enough to permit this fine gradation of shades in meaning, the everyday use of these words has assigned to them almost synonymous implications.

In the next sections, the words defined below will be used exclusively within the meaning here indicated and, in many cases, a word of attention will be included to prevent misinterpretations.

2.1.3.1 Definitions: Neighborhood Relations

Contiguous: Meeting or touching on one side.

Adjoining: Meeting or touching at least at some point. This concept includes that of contiguity.

Neighbor: Satisfying some empirical rule established as the "neighborhood relation," not necessarily implying contiguity.

As an example, the shaded modules of Fig. 8a and b are adjoining, but only the ones in Fig. 8b are simultaneously contiguous. This apparently excessive detailing is useful in cases like that of Fig. 8c: If we refer to the modules adjoining A, we refer to B, C, D, and E. But if we refer to the modules contiguous to A, only B and C qualify as such.

The neighborhood relation can assume any form and is not restricted to the common meaning of "immediate" neighbors. For example, the shaded modules of Fig. 9a and b represent the neighbors of module A under the conditions of the following rules: For 9a: Those modules having one side in common with A. For 9b: Those modules within a "Manhattan" distance of 3 from A.

2.1.3.2 Nomenclature

Non-regressive path: Any path traced in such a way that no two consecutive turns are in the same direction. Figure 10c and 10d.

Regressive path: Any path with two or more consecutive turns in the same direction. Figure 11.

Normal path: A distinguished set of the class of non-regressive paths characterized by having only one turn. Figure 10d.

Barrier: A collection of paths dividing the network of modules into two disconnected regions.

2.1.3.3 Labels

Labels of modules: Modules are labeled with a pair of coordinates, as indicated in Fig. 12. The first term of the pair indicates the row, and the second the column to which the module belongs.

Labels of vertices: The labels of vertices are derived from the ones of the corresponding modules according to the rule indicated in Fig. 13. The resulting coordinates are shown in Fig. 14.

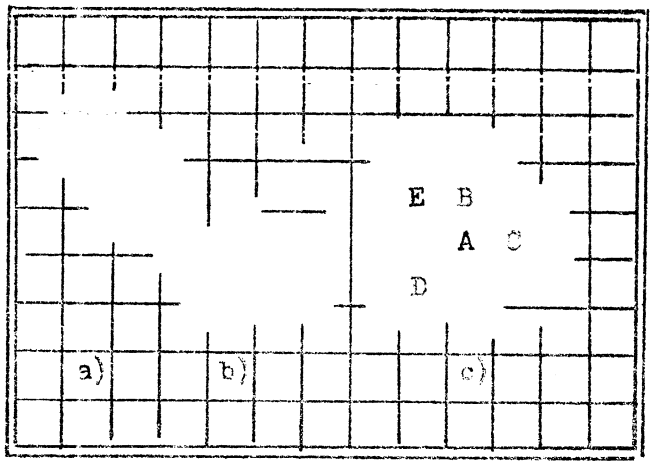


Fig. 8. Adjoining and contiguous modules.

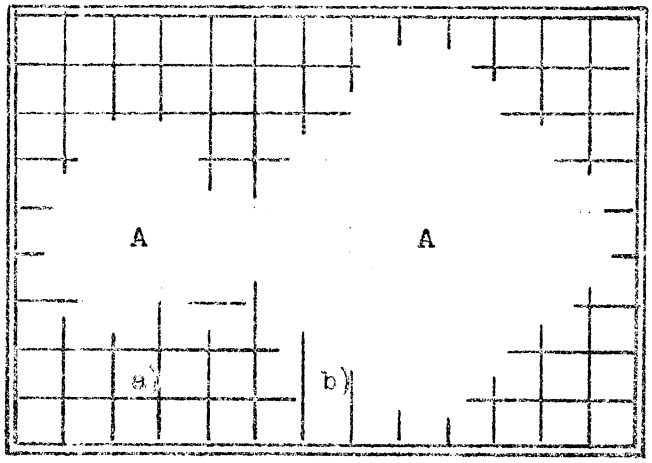


Fig. 9. Neighboring modules.

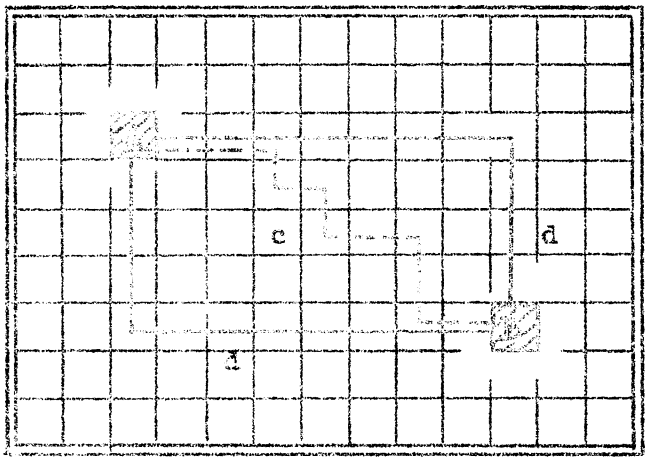


Fig. 10. Normal and non-regressive paths.

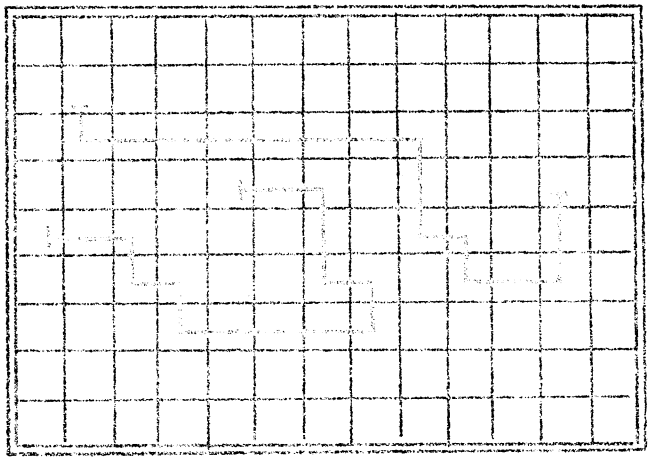


Fig. 11. Regressive paths.

(01;01)	(01;02)	- - - -	(01; n)
(02;01)	(02;02)	- - - -	(02; n)
(m;01)	(m;02)	- - - -	(m; n)

Fig. 12. Labeling of modules.

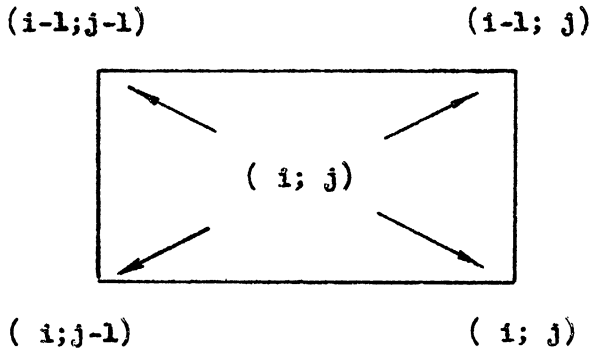


Fig. 13. Generating method for vertex coordinates.

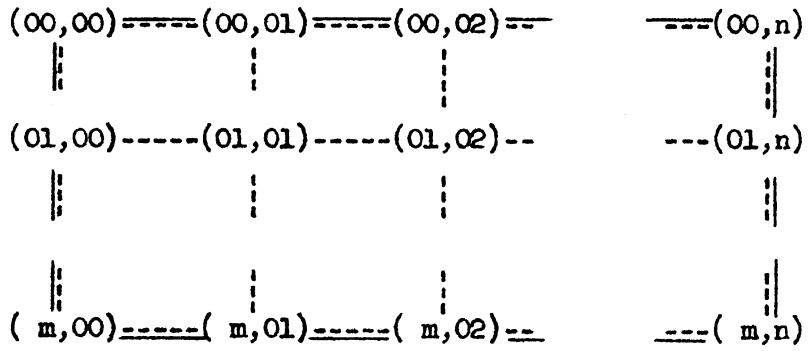


Fig. 14. Labeling of vertices.

2.1.4 Detection of Barriers and Isolated Regions

As explained in Section 2.1.2 a physical barrier is difficult to define because it entails the shape as well as a neighborhood relation between the paths contributing to the formation of the barrier. This difficulty is also present in the identification of barriers by means of an algorithm, because several tests have to be applied in sequence to ascertain whether a particular combination of path shapes and geometrical disposition constitutes a barrier.

Therefore, it seems logical to resort to some kind of mapping technique, such that when applied to the original configuration of paths it produces an image in which the desired property is easily identified.

At this point, we shall introduce some necessary definitions: Let us define the dual segment as the common side of a pair of contiguous modules connected by a segment of a path. Note that there is a one-to-one correspondence between dual segments and path segments associated with a particular path. Let the set of dual segments corresponding to a path be known as the dual of that path.

The path lists contain the labels of the modules through which the path is traced, and when the path is extended the list is updated by adding the label of the new module or modules. Simultaneously with the building of the path and the path list, the labels of the dual segments corresponding to the paths are ordered in lists. If the dual remains as a connected line then a single list is maintained, but when a disconnected dual segment is generated, it starts a new list. It can happen that while generating a new path, all the corresponding dual segments generate independent new lists. Further, if a dual segment S_1 belonging to a dual list D_1 is connected to some dual segment S_2 belonging to a dual list D_2 originally created by some other path, then the contents of the dual list containing S_1 are added to the list con-

taining S_2. At any stage, a dual list contains only a connected sequence of dual segments, perhaps contributed by several paths.

Therefore, there is no correspondence between the path lists. The paths simply generate the dual segments and these, depending on their resulting connections, can add to the dual of their own path, generate other dual lists, or join some pre-existent dual lists. Consequently, each dual list contains the coordinates of dual segments such that they have one coordinate in common, identifying them as belonging to a connected tree.

Lemma 1: A list of dual segments containing either a closed sequence of coordinates or the coordinates of two points belonging to the border of the network defines a line which divides the grid into two disconnected regions. This line is called a barrier.

Proof: Since each dual segment arises from a corresponding path segment connecting two contiguous modules, clearly no new path can connect them. Hence, no new path can cross the dual segment. Therefore any continuous sequence of dual segments cannot be crossed anywhere by a new path. So if both ends belong to some border or close on themselves, a disconnected region is generated. Hence, by definition, a barrier exists.

Corollary 1: Discontinuity of the set of dual segments arising from a single path indicates the possibility of crossing the path by a new path at every point of discontinuity.

Example: It is desired to determine if points A, B and C can be connected two at a time, given that the four paths indicated in Fig. 15 are pre-existent.

First, the duals of the paths can be generated according to the definition of dual. In Fig. 16, the four duals corresponding to the paths are shown distinctly to emphasize their origin. However, we are really interested in

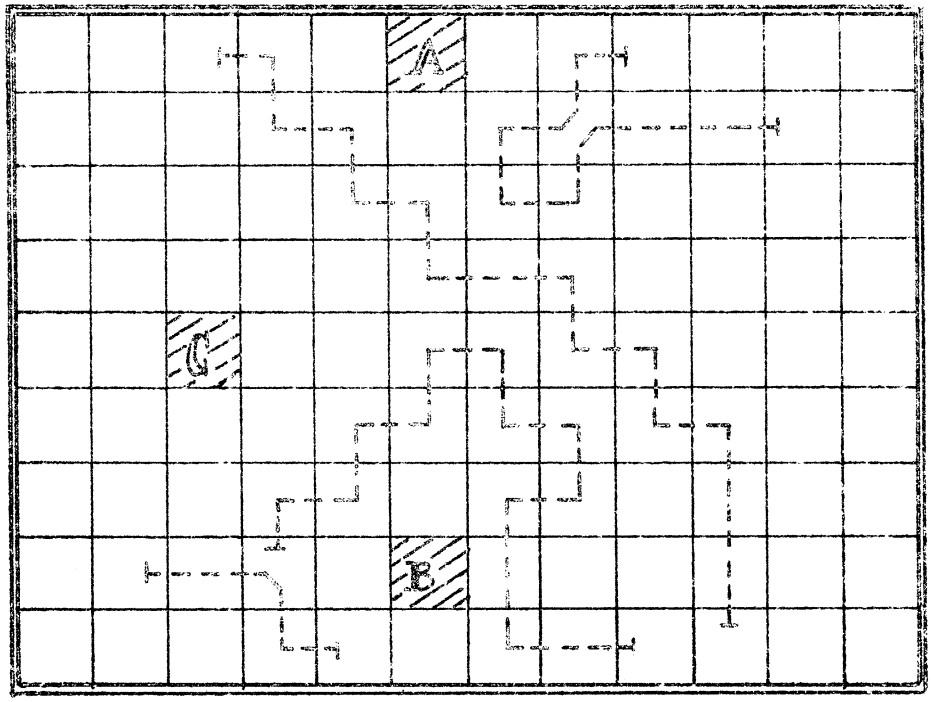


Fig. 15. Pre-existent paths.

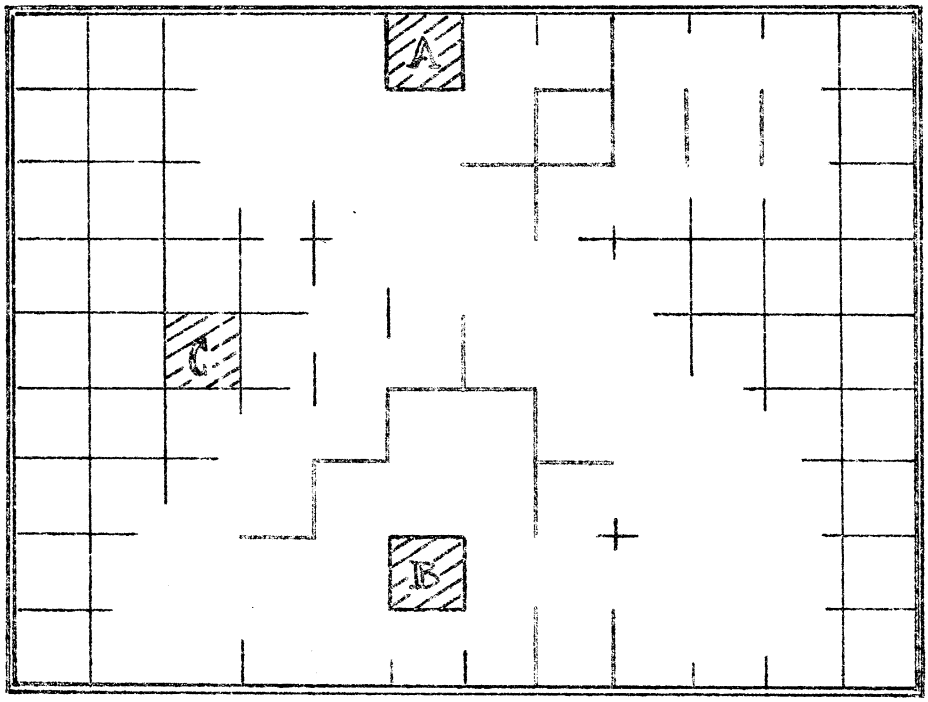


Fig. 16. Duals of the pre-existent paths.

connected sequences of dual segments independently of whether or not they belong to the same path. Therefore, at this stage, there should be nine dual lists as can be verified from Fig. 16 by counting the number of connected dual lines. Of all these nine lines, only one satisfies the requirements of Lemma 1; and this line is shown in Fig. 17. Because it is a continuous line formed by dual segments and with the ends on the border of the network, it is defined as a barrier and thus divides the network into two disconnected regions. As a result, it can be stated that module C is isolated from A and B, but A and B can be connected. If now all the duals are shown again, as in Fig. 18, it is easy to see how to trace the connection AB.

This suggests the possibility of solving the problem of connecting two points through the existent paths by treating it as a maze problem in which the walls are represented by the duals instead of by the actual paths. While this is perfectly possible when only a few of these connections are needed, it is completely impractical in this case when the procedure is to be repetitive, and one has to create a new path as soon as a connection has been made and some information has been transmitted between the terminal modules. Thus, a much simpler and faster, if less general, procedure is needed.

2.1.4.1 Programming the Detection of Barriers

While the number of paths is increasing, the lists of paths and of dual segments are constantly being kept up to date, and thus they increase both in length and number. A new kind of list denoted as a "barrier list" is generated from the list of duals whenever either of the following cases occur:

(a) When one entry happens to have each of the two coordinates common to one coordinate of two previous entries, identifying it as a link closing a loop; and

(b) When the list contains two coordinates belonging to the borders of the network.

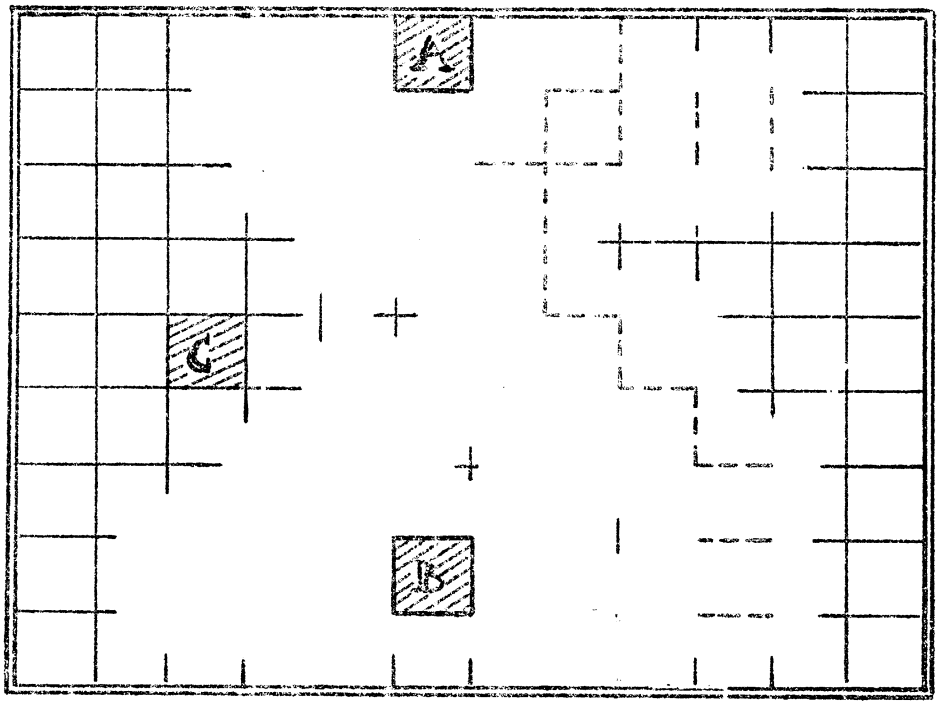


Fig. 17. Continuous barrier.

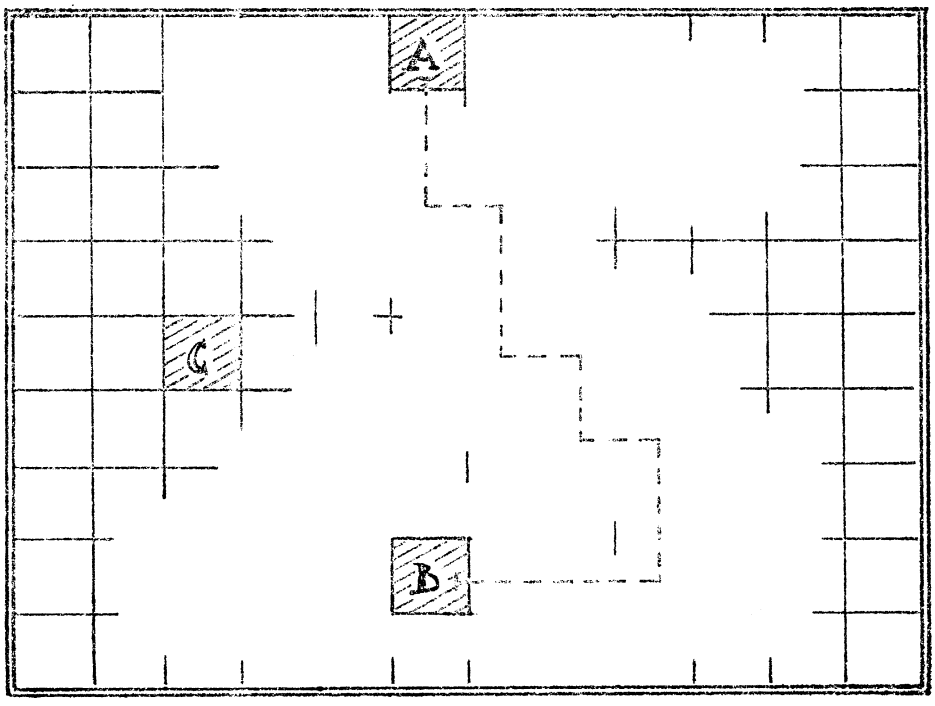


Fig. 18. Tracing path A-B.

The continuous updating of the three kinds of lists suffices to tell if a path is possible between any two given modules at any moment.

It is evident that if no barriers have as yet been completed and if no restrictions are placed upon the path as to the number of corners or its permissible length, then the connection is always possible. But even if one or more barriers are already present, it is still possible that the two modules to be connected lie in the same region with respect to the barrier or barriers. Two solutions for this problem are suggested:

(a) For each new barrier, two tables would be generated containing the labels of all modules belonging to each of the two disconnected regions. Any two new modules to be connected would be checked to see if their labels are contained in a single table, in which case both lie in the same region with respect to the barrier inducing the partition under consideration.

Obviously, this method implies the shifting of enormous amounts of data (the labels of all modules), and the whole procedure has to be repeated for every new barrier. Furthermore, the amount of data to be treated remains appreciably constant and does not diminish as could be expected, since the modules belonging to other previous barriers can now act again as originators of new paths.

(b) A normal path connecting the two modules is built on an assumed blank network, and its dual list is generated. Then Lemma 2 is applied.

Lemma 2: Given two modules, if the dual of any of the two normal paths connecting them contains an odd number of segments in common with the barrier proper, then the modules lie in disconnected regions and a path is not possible.

In Fig. 19, the black segments indicate the dual of the normal path connecting A with B. The green line is the barrier created by the red path. In Fig. 20a, the dual of the normal path (black) and the barrier (green) have one common segment, but Lemma 2 specifies with the barrier proper, so 20b is the appropriate representation of this case. Here, C, A and B are found to be in the same region because an even number, namely two, of segments are common to both the dual of the normal path and the barrier proper.

If the other normal path is traced it is found that it too has an even, namely zero, number of segments in common with the barrier proper.

2.1.5 Final Considerations for Path Building

2.1.5.1 Restriction on the Class of Paths Admissible

It has been mentioned in Section 2.1.2 that the path-building procedure has to be repeated for every instruction to be executed, and therefore it is imperative to employ a very simple and fast algorithm for path tracing. Furthermore, since some of the paths may be of considerable length, it will be helpful if the algorithm is amenable to parallel processing.

If one tries to resort to maze-solving techniques, it is found that these use a method of cell classification, assigning relative weights according to some neighborhood relation and in a monotonically varying sequence. Therefore, these methods are intrinsically sequential since the weight of a cell cannot be determined until the weight of its immediately preceding cell is specified.

In order to simplify the problem, we restrict the types of admissible paths to what is referred to as "non-regressive paths." A non-regressive path is defined as one traced following a set of priorities on the vertical and horizontal directions. The set of priorities establishes which of the

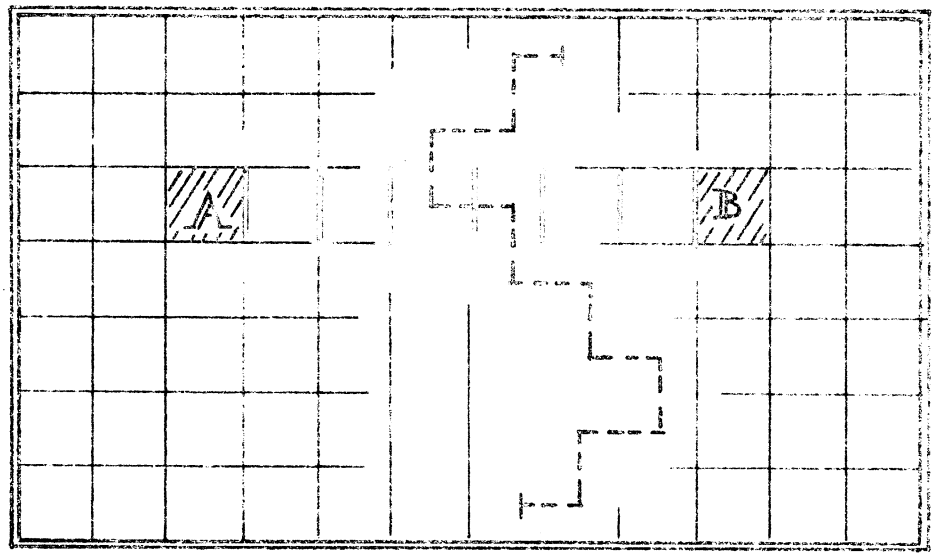


Fig. 19. Dual and barrier.

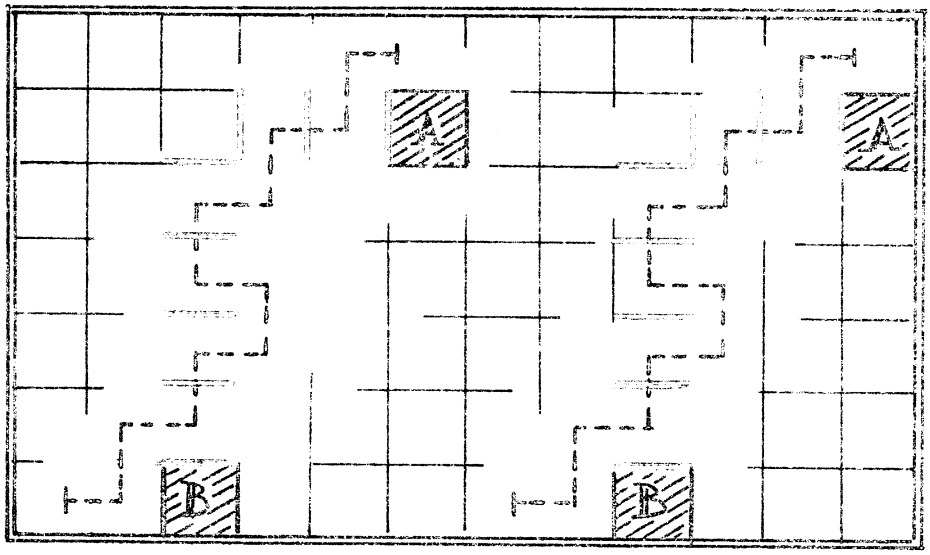


Fig. 20. Dual and barrier proper.

two senses are to be followed when tracing the segments in each of the horizontal and vertical directions.

This means that once the vertical and horizontal priorities are established, for example vertical down and horizontal to the right, all the path segments have to be traced in either of the two specified senses exclusively.

It is to be noted that this restriction on the allowable class of paths has the advantage of eliminating the need for an algorithm capable of tracing a minimum length path, since all non-regressive paths have the same length when measured in terms of the number of segments needed to connect the modules. This method of measuring distance has been called "Manhattan distance." And if each module is assigned a pair of coordinates, then this distance can be assimilated to the "Hamming distance."

It could be thought that this restriction on the types of paths would severely limit the possibility of connecting two modules through a set of obstacles, but it is easy to show that this is not the case even for networks of small size.

For an m x n network, the number of paths (non-regressive) connecting two opposite corners is:

$$\text{no. of n-r paths} \qquad \binom{n+n-2}{n-1} = \frac{(m+n-2)!}{(m-1)!\,(n-1)!}$$

For a minimal iterative circuit computer, with a 40 x 40 network, the number of non-regressive paths is

$$\binom{m+n-2}{m-1} = \binom{78}{39} = \frac{1.13 \times 10^{115}}{(2.03 \times 10^{46})^2} = 280 \times 10^{20}$$

Even for the impractical case of a 10 x 10 network, the number of paths is still very high:

$$N = \binom{20-2}{9} = \frac{18!}{9!\,9!} = 48{,}620$$

Within the class of non-regressive paths we distinguish the "normal paths" and the "broken paths," illustrated as a, b, and c respectively in Fig. 10.

2.1.5.2 Use of Redundant Paths

Since the number of available paths is so large, it is necessary to define some priorities as to the type of path most convenient to try to build first. Evidently the normal paths employ a very simple algorithm and therefore seem to be the natural choice for a first try. Furthermore, the length of all non-regressive paths, including the normal ones, is the same, and therefore there is no particular advantage with respect to propagation time through the path. But there is an advantage in time during the path-building phase, since only one "turn" instruction is needed in the case of the normal paths.

Since it is very probable that it may be possible to trace more than one path, one could very well use the extra paths to add redundancy to the transmission of information between the modules. If only one path is found, then it is used for the transmission with no checks. If two paths are available, the information received from them at the receiving module is checked for agreement and it is then used or discarded, stopping the program. If three or more paths happen to be available, a "majority vote" can be taken at the receiving end to determine the correct information.

The "majority vote" technique when used with three paths affords a high degree of reliability since the condition for acceptance of incorrect information is the occurrence of the same type of error at the same time in two of the three channels.

All the modules in the path except the end ones act only as transmission elements and therefore perform no logical function. It is safe then to assume that the only type of error that can be introduced is the failure of trans-

mission, as opposed to the dropping of bits or the generation of erroneous "ones" filling the spaces of some original "zeros."

Under these conditions, the bounds for cumulative errors as treated in Reference 4 do not apply. The simpler rules that follow give an estimate of the increase in reliability that can be expected from a transmission line composed of m parallel paths.

Let's denote by R_i the reliability of the individual module, and by R_T the reliability of the total path. Similarly, $F_i = 1-R_i$ will indicate the individual "failability" of the modules.

It is necessary here to remark that reliability is understood as the probability that the element will perform correctly during a certain time interval; in this case, during the time it takes to execute the transmission phase. That is, a reliability of 0.95 indicates that the element functions correctly 95 out of 100 times that it is pressed into service. It does not indicate that the element performs correctly during 95% of the transmission phase in any one attempt since in this case only a very few special patterns would be transmitted with no errors.

In order to calculate the reliability of systems with elements having individual reliabilities R_i, one proceeds in the following way: The total reliability R_p of a series of n elements with individual R_i's is the product of these R's.

$$R_p = \prod_{1}^{n} R_i$$

In this case, all the modules are physically alike, so that barring different environmental conditions all are supposed to possess the same $R_i = R$. Also, in this case, the total reliability of a path consisting of n elements in series is simply: $R_p = R^n$ See Fig. 21.

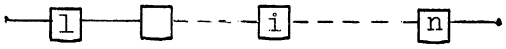


Fig. 21. Series connection of n modules.

When m such paths of individual reliabilities R_p each consisting of the same number n of elements are connected in parallel with no intermediate interconnections, as in Fig. 22, then the total "failability" is $F_T = F_p^m$, and therefore:

$$R_T = 1 - F_T = 1 - F_p^m = 1 - (1 - R_p)^m = 1 - (1 - R^n)^m.$$

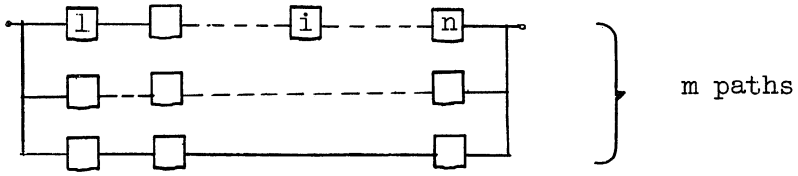


Fig. 22. m parallel paths of n modules each.

It is evident from the previous equality that the number m of parallel paths has a greater influence than the number n of elements in series in a path. This is especially true for values of R approaching unity and means that even for a long path there exists a possibility of compensating the effect of the large number of elements in series with a few parallel paths. For example:

For $R = 0.95$, $n = 10$, $m = 3$:

For a single path: $R_p = (0.95)^{10} = 0.5987$

For 3 paths in parallel: $R_T = 1 - (1 - R^n)^m = 0.9345$

It can be seen that the total reliability has not yet reached the original reliability of the individual module.

For R = 0.99, n = 10, m = 3:

For a single path: $R_p = (0.99)^{10} = 0.9045$

For 3 paths in parallel: $R_T = 1 - (1 - 0.9045)^3$

$$R_T = 1 - (0.0955)^3$$

$$R_T = 1 - 0.00087 = 0.99913$$

In this case, the total reliability of the network is greater than the reliability of the individual module.

2.1.5.3 Extra Requirements Introduced by the Redundant Mode of Operation

It is to be noted that this feature of built-in redundancy is obtained with almost no penalty in time or equipment. Time is not lost since all m paths can be traced at the same time and all are of the same length, and thus the procedures terminate simultaneously.

An extra requirement is the necessity of having extra hardware in the modules to implement the majority vote. This simply means that for a fair-sized machine there is a slight reduction in the number of modules available for the rest of the program.

2.1.5.4 Assignment of Priorities

Given two modules to be connected we will consider the one with the lowest row coordinate as the starting point. Informally, we can say that the "uppermost" module is the starting point. The priority is established as a sequence of two directions, and is indicated as the pair consisting of V (for vertical) and H (for horizontal) in one of the two orders.

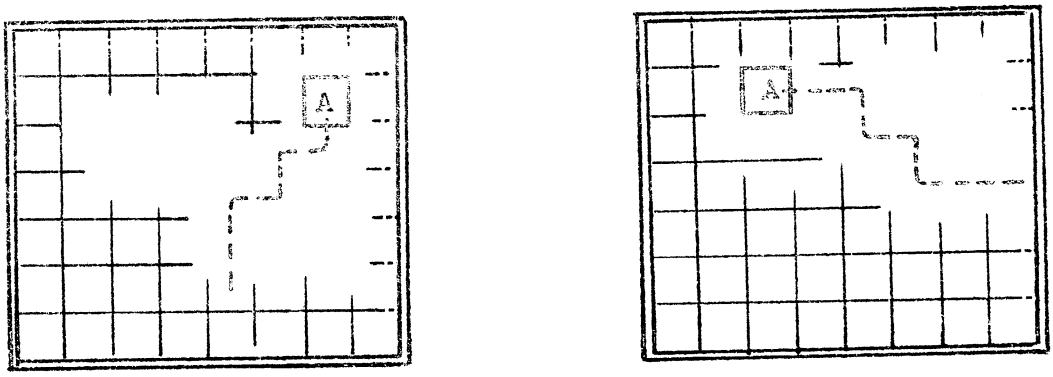
Thus, the pair (V,H) indicates that the vertical direction is followed as long as it is possible, whether or not a change to horizontal direction is possible. Only when an obstacle is met does the change to horizontal direction take place. But still the vertical direction has latent priority,

and consequently even if the horizontal direction is clear the vertical one is resumed as soon as it becomes possible to do so.

As the starting point has been defined as the uppermost module, the vertical direction needs no further qualification since it can only be in the "down" sense. The qualification for the horizontal direction is automatically given by the relative position column-wise of the "lower" module relative to the "upper" module.

Given two modules, the two combinations of (V,H) and (H,V) can, and generally do, give rise to completely different paths. In Fig. 23 the two different combinations of priorities are illustrated. In 23a the priority is (V,H) and the starting direction is vertical. Notice that when the path is proceeding in the horizontal direction, it resumes the vertical direction as soon as this becomes possible. This is notwithstanding that there are two or more modules available in the horizontal direction.

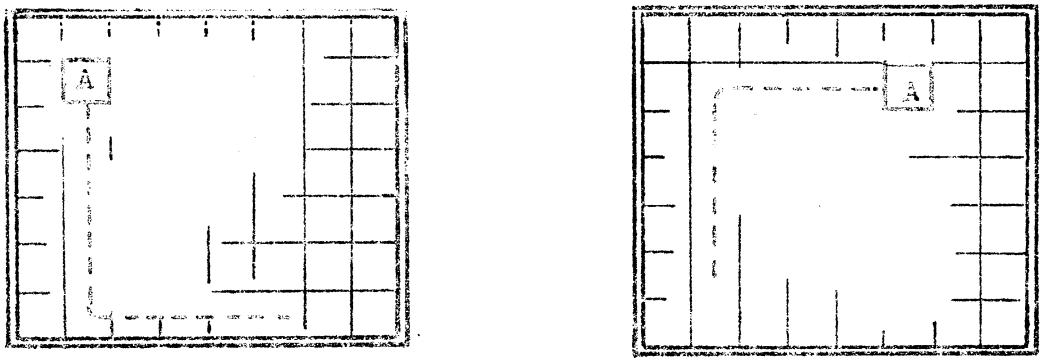
In some cases, the path starts from the uppermost module to be connected, following the low priority direction and thereby apparently violating the rules of direction precedence. What actually happens, as in Fig. 24, is that the immediate neighbor of the starting module in the direction specified by the priority is either an obstacle or is non-available because it lies in the shadow of some other obstacle. Therefore, as the high priority direction finds no available modules through which to trace the path, the secondary priority is followed, but only as long as is necessary to find an available module in the high priority direction. In Fig. 24a the obstacles are such that even with a (H,V) priority the resultant path has all the features of one traced according to a (V,H) priority. The same happens for a (V,H) priority, as seen in Fig. 24b.



a) Priority: (V H)

b) Priority: (H V)

Fig. 23. The two cases of priorities.



a) (H V) priority

b) (V H) priority

Fig. 24. Two cases starting with the low priority direction.

2.1.5.5 Path-Building Procedure

The path-building procedure involves two different steps: (i) Eliminating zones of modules non-acceptable as path components, and (ii) Tracing procedure.

(i) The elimination of module zones non-acceptable as possible path components is carried on by a "shadowing" technique. This procedure takes into account the priority pair and indicates which zones are forbidden for path penetration. This implies that if the path is allowed to enter one of these zones, then it becomes necessary to trace back part of the path in a direction opposite to one of the directions specified in the priority, thereby producing a regressive path which is not admissible.

The shadowing method operates in the following way:

(a) Determine the starting point and choose a priority pair.

(b) From the priority pair and the relative position of the two modules, determine the sequence of directions in which the path has to progress.

(c) From this sequence of directions, determine the two sides of the network which will serve to determine the starting obstacles for the shadowing procedure. If the directions are vertical down and horizontal left, then they can be indicated by a pair of arrows, like ↓← . From here we deduce that the right-hand border and the lower border are the starting places for the shadowing procedure.

(d) For every obstacle contiguous to the lower side, determine its highest point and project a horizontal line to the right until it intersects the right-hand side of the network, even if this means running over some obstacle. The set of modules limited by this line, plus the obstacle and the two sides of the network constitute

the "shadow" of the obstacle, which is considered a zone forbidden to the path-tracing procedure.

(e) Repeat the same procedure of (d) for obstacles attached to the right-hand side, projecting the shadow vertically from the left-most point until it intersects the lower side of the network.

(f) Repeat the horizontal and vertical shadowing procedures as described in (d) and (e) for every obstacle now adjoining any of the shadowed zones. Adjoining means having at least one vertex in common.

(ii) When no more shadowing is possible because the rest of the obstacles are detached from the shadowed zones, the path is traced following the assigned priorities and considering both the obstacles and shadowed zones as obstacles.

In Fig. 25 the preceding method has been applied to the problem of connecting modules A and B through the set of obstacles indicated by C through S. The elimination and tracing procedures are explained below, with reference to steps (a) through (f):

(a) The starting module is determined by the lowest row coordinate. A priority pair is set arbitrarily: (V,H). Complete specification: (V_SH).

(b) Module B is to the left of A, therefore the sequence of directions is vertical down and horizontal left.

(c) The sequence of directions can be represented as: ↙. Therefore the right-hand side and the lower borders are the starting places for the shadowing procedure.

(d) In Fig. 25, obstacles R and S are contiguous to the lower side. Thus their high points project shadows to the right indicated by (1) in Fig. 26, extending to the right side of the network.

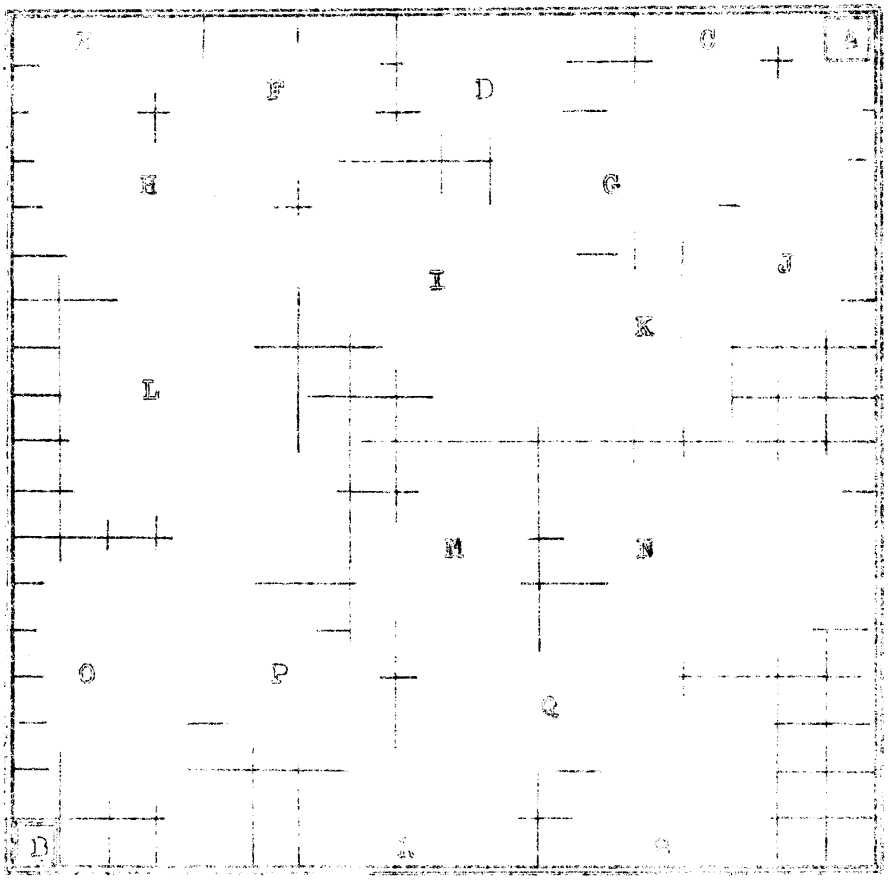


Fig. 25. Maze problem.

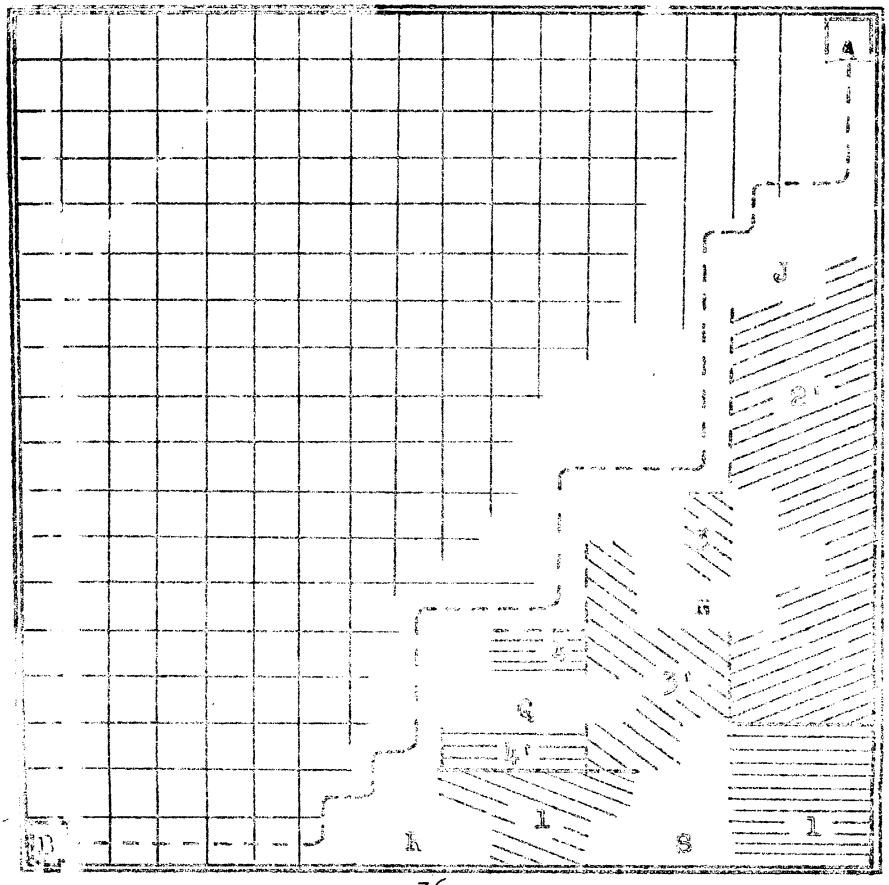


Fig. 26. First solution with priority $V_S H$.

(e) Obstacle J is contiguous to the right border, and therefore it projects shadow 2' vertically extending to the lower border. Here it is not necessary to cover up region 1 in the shadow of S because it is already eliminated as a possible region for path tracing.

(f) Obstacle N is now contiguous to a shaded zone, and so it projects a horizontal shadow 3 and a vertical shadow (3'). Q is now contiguous to a shaded zone. It projects shadow 4 and 4'. The shadowing possibilities are now exhausted.

The limitation to non-regressive paths makes the route pattern very sensitive to small changes in the shape of obstacles.

If all types of paths were admissible, a small change in the shape of one of the obstacles would probably imply only a small detour of the original path, but with non-regressive paths not only the shape but the adjacency of other obstacles to the modified shadow intervenes to modify radically the shape of the path. This means that a small variation in the configuration of one obstacle has a local influence plus a "long distance" effect according to whether the new shadow profile becomes contiguous or ceases to be contiguous to some other obstacle.

In order to illustrate the wide variation induced in the path route by minor changes in the shape of the obstacles, the same basic pattern of Fig. 25 has been used in Fig. 27 with the addition of a one-module obstacle G! The priority is still (V,H), but the exit from A is impossible in the vertical direction, and consequently the path is started in the direction of secondary priority, H.

If now a second one-module obstacle is added, like I' in Fig. 28, the path again is modified substantially; this time by the inclusion of obstacles L and O and their shadows in the forbidden zone. Figure 28. Here, the new

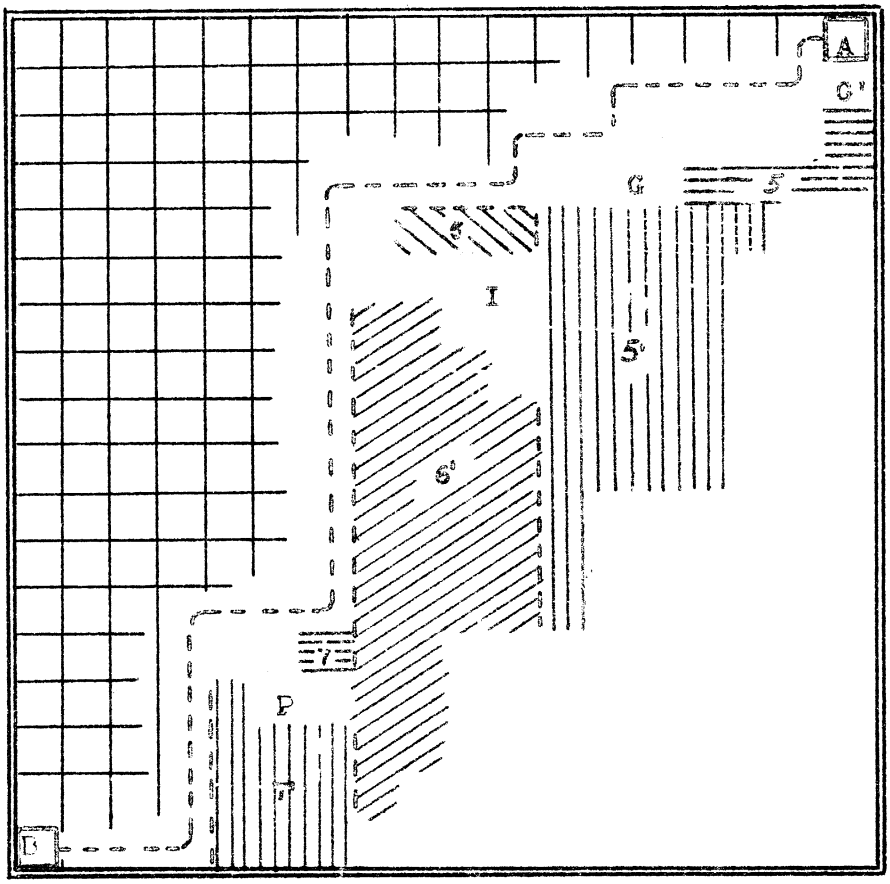


Fig. 27. Change in path route induced by the inclusion of G'.

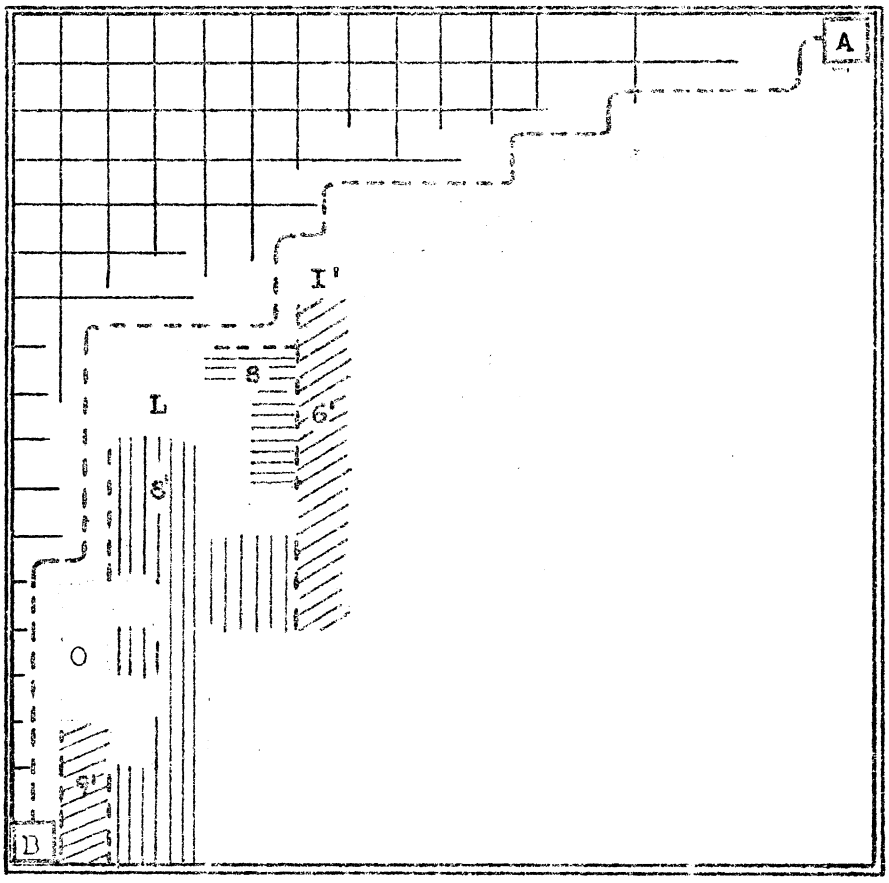


Fig. 28. Change in path route induced by the inclusion of I'.

additional shadow projected by I' is enough to cause obstacle L to become contiguous, which in turn brings O into the forbidden zone. The new path, as a result, is very different from the one in the original problem (Fig. 26).

2.1.6 Flow Diagrams for the Implementation of the Barrier Detection and Path-Tracing Procedures

2.1.6.1 Adapting the Procedures for Computer Solutions

The implementation of the barrier detection procedure does not present any difficulty. On the contrary, the implementation of the path-tracing procedure, as explained in Section 2.1.5, requires the use of pattern recognition techniques since it depends on the determination of such features as the leftmost and uppermost corner of an irregular pattern of modules consituting an obstacle. In order to circumvent this difficulty, the obstacles are not treated as patterns of modules, but each individual module in the pattern is treated as a "unit obstacle." Therefore the obstacles referred to in previous sections are now conglomerates of "unit obstacles." These unit obstacles are also now treated individually as independent contiguous obstacles. The procedure is now applicable in the same way as before, since the contiguity of the unit obstacles assures the final treatment of the whole pattern by the shadowing method.

2.1.6.2 Flow Diagram for the Detection of Barriers

Figure 29 shows the flow diagram for the detection of barriers. It starts with the given coordinates of the two modules to be connected. Then the existence of any barrier is checked by reference to the barrier list. If no barriers are present, the whole algorithm is skipped and the path-tracing algorithm is entered directly. If there is a barrier, then the normal path connecting the two modules is traced, and the number of common segments be-

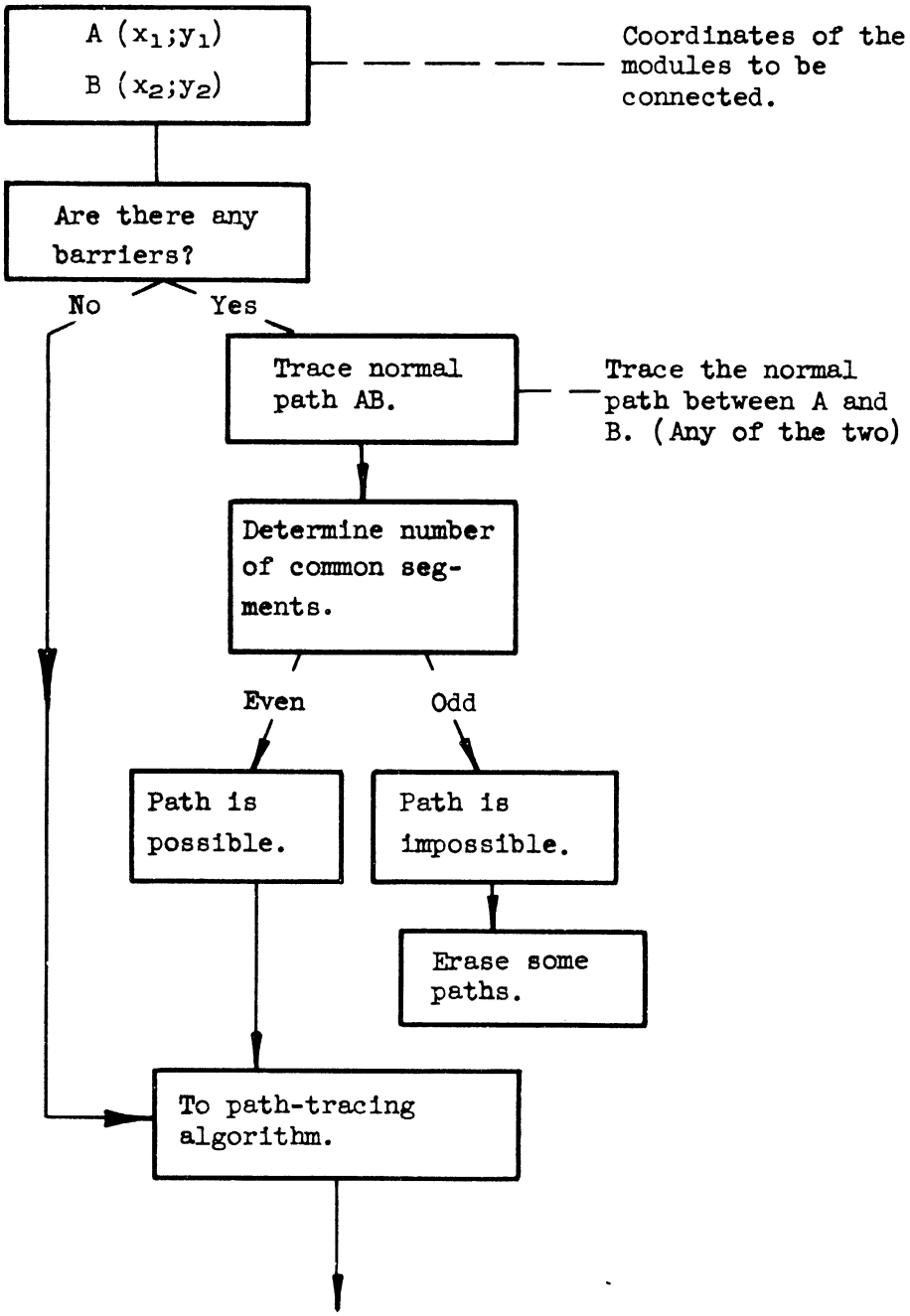


Fig. 29. Flow diagram for the detection of barriers.

tween the normal path and the barrier proper is determined. If there is an even number of common segments, then the two modules are in separate regions and there is no possibility of a path. If there is an even number including zero, of common segments then the two modules lie in the same region with respect to the barrier and a path is still possible. Therefore, the path-tracing algorithm is entered.

2.1.6.3 Flow Diagram for the Path-Tracing Algorithm

Figure 30 shows the flow diagram for the path-tracing algorithm. It starts with the determination of the sequence of directions. This is derived from the arbitrarily assigned priority and a comparison of the relative addresses of the two modules to be connected. Next, the baseline is determined. In the initial step, the baseline will be constituted by two sides of the original network, but later in the iteration it will be formed by the horizontal and vertical projection of the uppermost and leftmost (or rightmost) corner in the obstacle being considered. Any module adjoining (having at least one corner in common) the baseline is considered the new neighbor, and its coordinates are labeled (x_o, y_o). Then all modules whose $\underline{x}, \underline{y}$ coordinates are greater than x_o and y_o respectively, are eliminated. After the elimination or "shadowing" of these modules, a new baseline is determined and the procedure is repeated. If there are no new neighboring modules, the other half of the baseline is considered and a similar procedure is followed. If there are no neighbors to any of the two halves of the baseline, then the shadow procedure is terminated and the path can be traced.

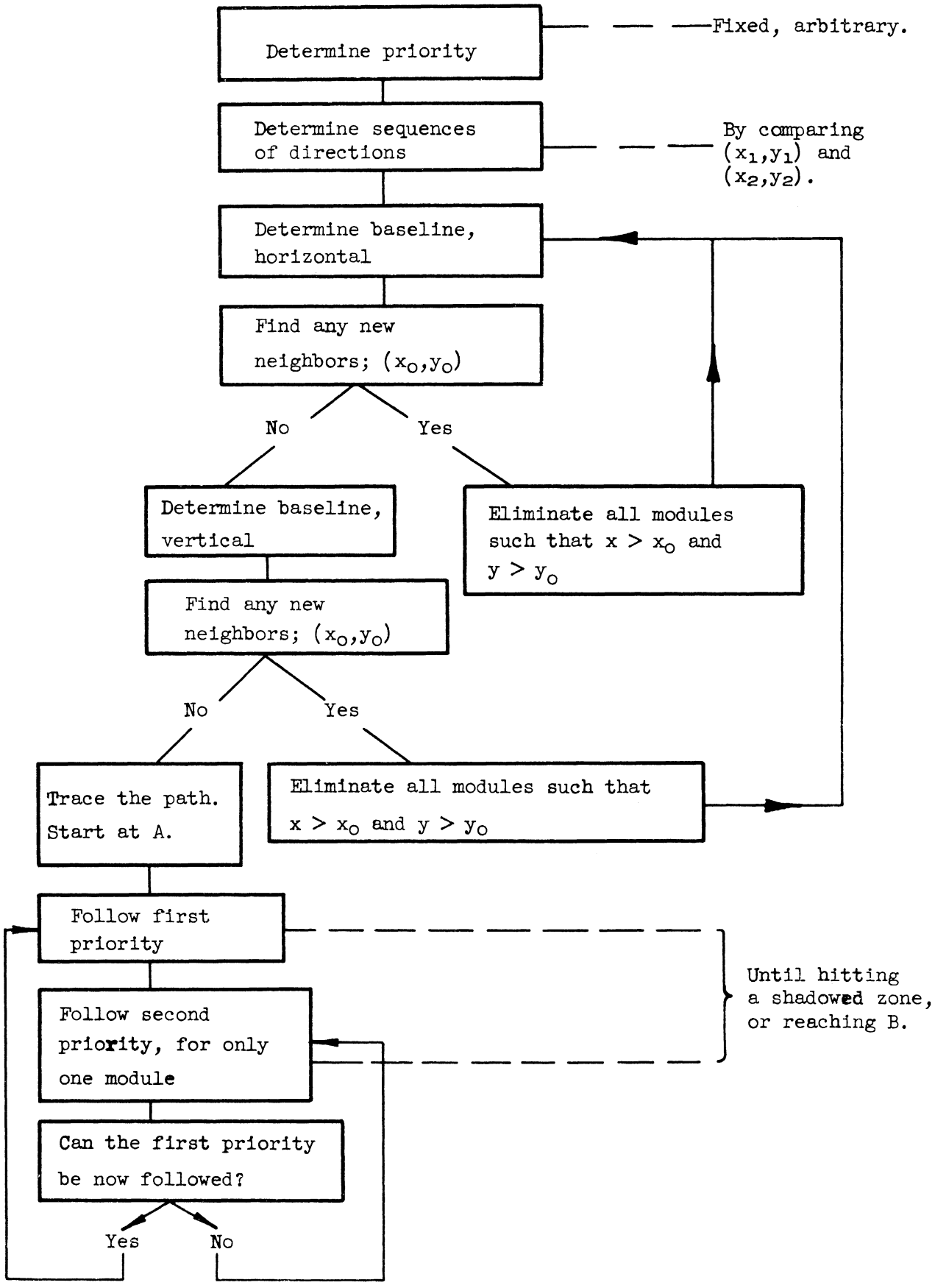


Fig. 30. Flow diagram for path tracing.

2.2 TRANSLATION ALGORITHMS

2.2.1 Maximal Decomposition of Algorithms

The fundamentals which govern the following discussion are from the general area of mathematical logic and, in particular, from metamathematics. Markov[24] defines a formal mathematical entity as the "normal algorithm." The metamathematical properties of the class of normal algorithms have been shown to be equivalent to other computational structures by Detlovs.[22] It is interesting to note there exists a universal algorithm with properties similar to a universal Turing machine. Also, any number whose representation is computable by a normal algorithm is computable by a Turing machine and vice versa. Further, the unsolvability of the halting problem for normal algorithms can be proven using the same basic techniques as used for Turing machines.

The results of Markov add to the intuitive belief of Church's thesis[19] but do not seem directly applicable to the analysis of programming algorithms. The normal algorithm is a finite sequence of symbols which define a unique mapping from an input string of symbols into an output string of symbols. The number of distinct symbols in all three classes must be finite. Few programmers consider computation, input and output in this limited sense. For example, variables in algorithms are considered as representing real numbers rather than truncated approximations.

The significance is that a formal procedure for the analysis of the most general forms of algorithms is desired. Yet, it has been proven that no procedure can be developed to analyze all possible normal algorithms.[20,21,23,25] It is worthwhile to emphasize the difference between these classes of algorithms. There

are a countable number of normal algorithms. This follows from the fact that each normal algorithm can be represented by a finite number of symbols from a finite alphabet. The set of algorithms in its most general form has cardinality aleph null, the order of the real numbers. This follows from consideration of the set of algorithms: Input value for the real variable, X, add X to a real constant, output the real varialbe, Y, which is the sum. By letting the constant be each real number, a set with cardinality at least aleph null is generated. Since there can be at most a countable number of variations in the computation of such an algorithm, the set of all algorithms of the most general form has cardinality at most aleph null.

By observing the methods for proving undecidability results about algorithms,[21,25] two key factors become apparent. First, any formal procedure for analysis of algorithms must prevent self-reference in order to guarantee termination in a finite number of steps, i.e., eliminate the possibility of the halting problem.

Second, the presentation of the algorithm must be interpreted as a sequence of subalgorithms where the lowest level of subalgorithm analyzed is a primitive operation on variables or constante, i.e., addition of two real numbers is considered as a primitive operation.

2.2.1.1 Step 1. Choice of Primitives

Choose a set of primitive operators. This set must be sufficient to express the algorithm of interest. The algorithm must be explicitly stated in terms of the primitive operators plus connectives and control information which designate flow of computation in the algorithm.

Remark 1

Implicit in the word algorithm, as used here, is a specific beginning and termination as well as a unique sequence of operations for any given data set. It is assumed that at least one set of data allows the algorithm to terminate after a finite number of computations. The decomposition proceeds by replacing the conditional branch instructions which are a function of data by definite functions which represent a reasonable data set. Although no specific set of data on which the algorithm might operate is required, a reasonable knowledge of such data sets is assumed.

2.2.1.2 Step 2. Branch Assignment

To each conditional branch statement in the algorithm, assign a function whose single argument is the number of times the conditional branch has been reached. The range of this function is a set of places in the algorithm from whence the computation may proceed. The operations performed in computing this function are not considered as operations of the algorithm itself. When following a sequence of operations through the algorithm at a branch: (i) the operations to evaluate a conditional branch variable in the algorithm are considered in the sequence, (ii) the next operation of the sequence is determined by the function assigned to that branch, and (iii) the counter which indicates the number of times the branch has been reached is incremented by one. There will be some finite number each possible branch.

Remark 2

The purpose of these functions is to obtain an explicit sequence of computation that is independent of data values required by the algorithm. The

functions need not model any particular data but should represent a first order approximation of a hypothetical average data set.

The type of function that will usually suffice is defined as follows: The conditional branch in the algorithm is replaced by a function which specifies branch every k^{th} time this function is evaluated and otherwise continues to the next step in the algorithm.

2.2.1.3 Step 3. Formation of Tree

Form a tree of primitive operations from the algorithm starting with the terminal branches and proceeding toward the base as follows:

Create a terminal node for each operation that can be performed on constants and/or input data. Replace each such operation and its associated arguments by a number sign in the algorithm. The number sign may appear as an argument for other operations or may be a result produced by the algorithm.

On successive levels of the tree, create nodes for each remaining operation that can be performed on constants and/or data and/or the number sign. Repeat this construction until only number signs remain in the algorithm.

Connect the remaining number signs to a single base node. The instance of the algorithm determined by the branch assignments is thus completely encoded as a tree structure of operators.

Remark 3

For any algorithm Step 2 guarantees that Step 3 may be effectively carried out, i.e. the number of operations stated in the algorithm is finite. For each conditional branch there is a finite number of encounters before all possible branches have been taken. Thus, with a finite number of observations it can be

determined if an operation meets a condition of Step 3.

If the algorithm can be expressed in some formal language, Step 3 can be performed on a computer. The description of such a computer program is presented in 2.2.3.: Machine Implementation via a Translation Algorithm. 2.2.4 extends the capability of 2.2.3 through an augmented language.

2.2.1.4 Step 4. Determine Computation Time and Degree of Concurrency

Begin at the base of the tree resulting from Step 2 and label each node with α_i where α_i is the length of the longest path from the base to the i^{th} node. The maximal length computation sequence, T, is max (α_i). This is the number of execution cycles required on a highly parallel computer. To determine the degree of concurrency, apply the scheduling procedure of T. C. Hu,[18] i.e., define N_j as the number of nodes labelled with T-j. The minimum number of processors required to complete execution in T steps is P.

$$P = \max(r_k)$$

where

$$r_k = \frac{1}{k} \sum_{j=1}^{k} N_j \text{ for } k=1,\ldots,T.$$

Remark 4

The computation of Step 3 is trivial in the sense one need only count nodes, then sum, divide, and pick out the largest of a few numbers.

Example

The given algorithm: (Compute $C_0, C_1, \ldots, C_{2k}$ which are the Fejer approximation

of the Fourier series

$$X = C_0 + C_1 \sin\theta + C_2 \cos\theta + C_3 \sin 2\theta + C_4 \cos 2\theta + \ldots + C_{2k} \cos k\theta$$

where the n equally spaced values which would be supplied as data are $x_1, x_2, \ldots, x_n$. Note k and n would also be supplied as data.)

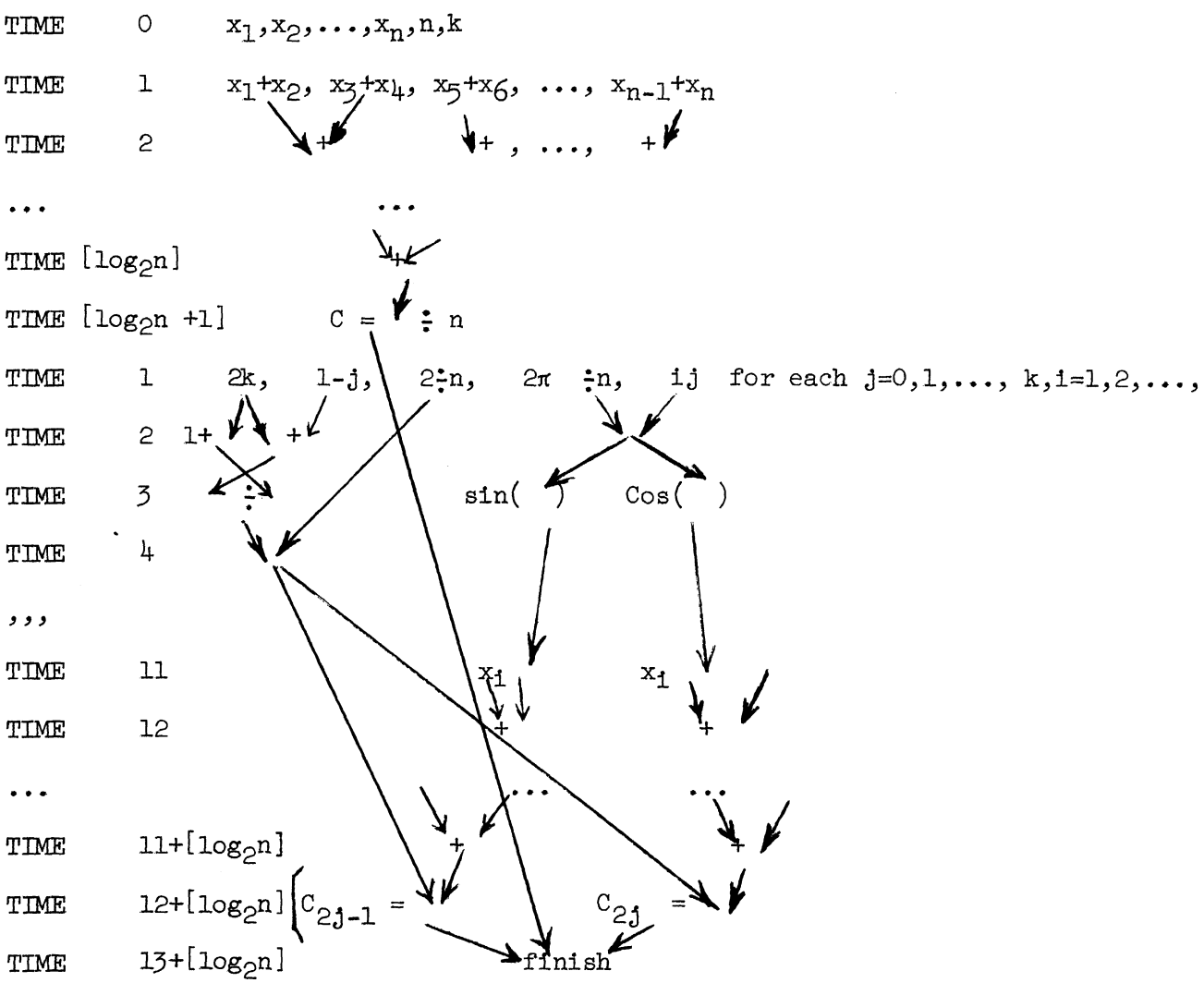
$$C = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\left.\begin{aligned} C_{2j-1} &= \frac{2k+1-j}{2k+1} \frac{2}{n} \sum_{i=1}^{n} x_i \sin\left(\frac{i}{n}\, j\, 2\pi\right) \\ C_{2j} &= \frac{2k+1-j}{2k+1} \frac{2}{n} \sum_{i=1}^{n} x_i \cos\left(\frac{i}{n}\, j\, 2\pi\right) \end{aligned}\right\} j = 1,2,\ldots,k$$

Step 1. Choose the elementary operations: add, subtract, multiply, and divide, assign 1 unit of time for computation, with sin and cos assigned 7 units of time for computation.

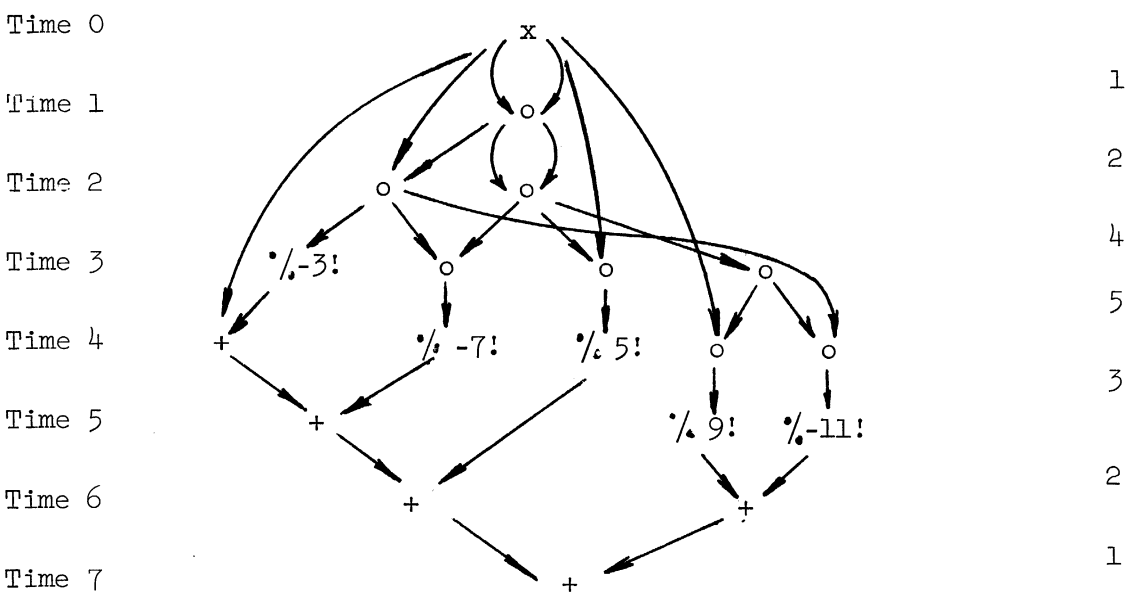
Step 2. Rather than use a specific integer for n and k, let the symbols n and k denote two fixed integers.

Step 3. Form the tree in sufficient detail.



Step 4. The minimal time for computation of all coefficients simultaneously is $13 + [\log_2 n]$.

To determine the minimal number of processors for a specific case, assume n = 200, k = 10. Also assume the sin and cos routines have the following computational graph:



This graph computes

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!}$$

For computation of the number of processors required a table of number of operations for each execution time is prepared:

		r_k			
Time 1	2002	2002	*		
Time 2	2000	2000	*		
Time 3	4000	4000	*		
Time 4	8000	8000	*		
Time 5	16000	16000/1	=	16000	Maximum
Time 6	20000	36000/2	=	18000	
Time 7	12000	48000/3	=	16000	
Time 8	8000	56000/4	=	14000	
Time 9	4000	etc.			
Time 10	4000				
Time 11	2100				
Time 12	1050				
Time 13	525				
Time 14	252				
Time 15	137				
Time 16	74				
Time 17	52				
Time 18	31				
Time 19	21				
Total Sequential Operations Less Indexing	84,244				

*r_k must account for the precedence relations where the computation is defined by a graph and not a tree. Each r_k is the number of operations at the k^{th} time until the computation structure becomes a tree.

Thus, at most, 18000 processors could be used. The number of execution times would be 19 on a computer with this number of processors while 84,244 execution times would be required on a single processor computer. If the requirement is made that only one type operation may be performed during a given execution time,

then at most 6000 processors could be used and about 60 execution times would be required.

2.2.2 Analysis of Sixteen Numerical Computation Algorithms for Concurrency of Arithmetic and Control

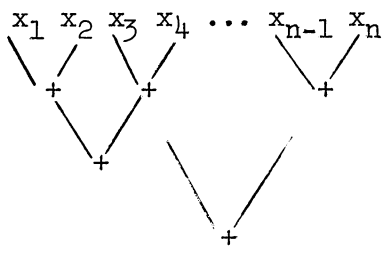
A brief description of each algorithm will be given. Significant degrees of concurrency, methods of obtaining concurrency, and degree of local or global control will be mentioned when applicable. A table with pertinent data summarized appears at the end of this part of the report.

Algorithms are presented beginning with trivial computations so that complicated algorithms may be analyzed in terms of simpler ones. The following computations are examined:

1) Sum or product of N numbers.
2) Evaluation of n^{th} degree polynomial.
3) Multiplication of a vector by a matrix.
4) Multiplication of 2 vectors.
5) Matrix inversion.
6) System of linear equations.
7) Ordinary differential equations.
8) Least square bit.
9) Computation of the n^{th} prime number.
10) Search Procedure.
11) Sorting procedure.
12) Fourier expansion.
13) Evaluation of Fourier series.
14) Neutron diffusion equation.
15) Neutron transport equation.
16) Eigenvalues of a matrix.

2.2.2.1 Sum or Product of N Numbers

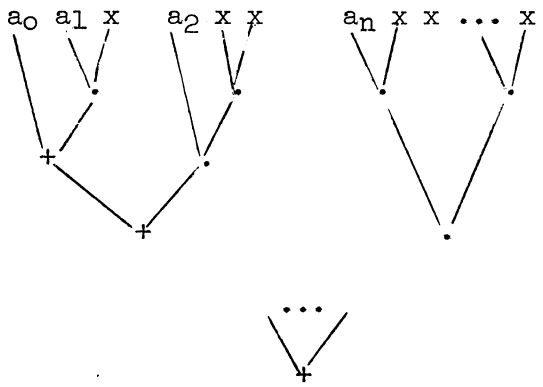
The fundamental limitation is the combination of partial results. During the first step at most N/2 operations may be performed on distinct pairs of numbers. During the second step at most N/4 operations may be performed on the N/2 previous results. The significant form of the computation is a tree



In this form it is easily seen that at most N/2 operations could be performed concurrently and the computation time using maximum concurrency is at most $[\log_2 N]$. The brackets [] denote the least integer equal or greater than the enclosed quantity. The number of sequential computations, N-1, is the same as the number of computations when using concurrency.

2.2.2.2 Evaluate N^{th} Degree Polynomial

This computation has the form of two superimposed trees



The minimum concurrency is formed by eliminating redundant computation from this tree and applying the scheduling algorithm of T. C. Hu.[18] As might be expected, the concurrent time is twice that of a single tree, $2[\log_2 N]$. By removing redundant computation of x^k terms at most N simultaneous operations are needed to compute

$$\sum_{k=0}^{n} a_k \, x^k$$

in the minimum concurrent time. The optimal sequential computation, $a_0 + (a_1 + \ldots (a_{N-2} + (a_{N-1} + a_N x)x)\ldots)x$, requires 2N sequential times.

2.2.2.3 Multiply Vector by Matrix

An N element vector is multiplied by an N x N matrix to produce another N element vector. During the first step, each element of the vector is multiplied by each element of the corresponding column, i.e., N^2 multiplications. During the next $[\log_2 N]$ steps the addition of the row products is completed as per 2.2.2.1 above. The sequential time consists of N^2 multiplications followed by N^2 additions.

2.2.2.4 Multiply Two Matrices

During the first step all possible products are formed between the elements of two N x N matrices. Then N^2 applications of 2.2.2.1 during the next $[\log_2 N]$ steps complete the computation of the product matrix,

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \, B_{kj}.$$

The sequential time is N^3 multiplications plus N^3 additions.

2.2.2.5 Matrix Inversion

The computation of the inverse of an N x N matrix is given in detail in the 1963 SJCC paper by Squire and Palais. This computation is in the local con-

trol category because various operations are performed on different rows of the matrix simultaneously.

2.2.2.6 Solving System of Linear Equations

Given N equations in N unknowns with K different constant vectors, the K sets of solutions can be found in 3N steps by performing the same operations as in 2.2.2.5 on the augmented N x N + K matrix. Additional concurrency needed is 2NK and the additional sequential time is $5N^2K$ over and above that in 2.2.2.5.

2.2.2.7 Solving Ordinary Differential Equations

Given a system of N simultaneous second order ordinary differential equations, solve for K intervals by the fourth order Runge-Kutta method. The sequential time of $12N^2K$ was computed on the assumption that 17 operations would be required by the forcing functions. The concurrent time of $4(4 + [\log_2 N])K$ and maximum concurrency $N^2/2$ were based on the same assumption. The algorithm may be found in most introductory texts. There is no gain by overlapping the computation for each of the K intervals but there is a sizable reduction of computation time within each interval. The majority of the computation is four applications of 2.2.2.3 to obtain the required derivations. The classification as global is marginal up to a system of 20 equations were concurrent time is about 1/100 sequential time.

2.2.2.8 Least Square Bit

Given N data sets with K factors in each, find K coefficients for a linear polynomial that is the least square predictor of the data. The computation consists of constructing a matrix of sums of squares and cross products, i.e., NK^2

multiplications and NK^2 additions to form a K x K matrix. This requires 2N concurrent time steps with K^2 operations each step. Once formed the matrix is inverted, requiring 3K concurrent steps as in 2.2.2.5. Then a final application of 2.2.2.3 yields the desired K coefficients in $[\log_2 K] + 1$ steps. Since $[\log_2 K] + 1$ is small compared to 3K, it does not appear in the summary. For N at least five times K the procedure may be classified as global.

2.2.2.9 Compute the N^{th} Prime

Increasing numbers of divisions are performed each step. During the K^{th} step, the k-1 primes are divided into the integers greater than the K-1 prime. A non-zero remainder test, made simultaneously, exhibits the value of the next prime. Storage requirements grow too rapidly for this to be of practical use but it is of interest that the method yields one prime for each computation step.

2.2.2.10 Unordered Search

A simultaneous comparison is made between a given value and all N values in a table. At the entry where the comparison occurs the corresponding second entry is fetched to a common cell. This requires 4 steps for any length list. Sequentially, the log-two search is optimal if the table is ordered.

2.2.2.11 Sort N Elements

An interchange sort is used which repeats the following N times for a list of N elements. Consider adjacent pairs and interchange those pairs which are not in the proper sequence. Alternate steps consider pairs displaced one position. With 2.2.2.2 instructions for the test and interchange a slight advantage

over a sequential process is realized.

2.2.2.12 Coefficients of Fourier Series

See example on p. 49.

2.2.2.13 Evaluating Fourier Series

Evaluate $A_0 + A_1 \sin X + A_2 \cos X + \ldots + A_{2N+1} \sin NX + A_{2N} \cos NX$. Let CS be the sequential time for sine and cosine computation. During the first step compute KX for K = 1,2,...,N, then compute the sines and cosines during the next CS steps, then compute the products, and finally apply 2.2.2.1. Classification is global if N is large due to simultaneous computation of sines and cosines.

2.2.2.14 Neutron Diffusion Equation

Solution of the parabolic partial differential equation is by the alternating direction method. The computation involves computing a tridiagonal matrix then solving a system of equations using their matrix. Greater concurrency is possible in higher dimensions since the computation involves relaxing one dimension per iteration. For N intervals in the grid in each dimension, there can be an order of N to N^2 reduction in concurrent time over s sequential time. As in 2.2.2.7 the saving is within an iteration.

2.2.2.15 Neutron Transport Equation

This is an integral-differential equation. For the N group case there are N equations. K is the order of integration used at each step. As with 2.2.2.7 and 2.2.2.14, the saving is within an individual iteration and not in overlapping

iterations. No estimate of local or global control can be made without a specific choice of the integral approximation method.

2.2.2.16 Eigenvalues of a Matrix

On real symmetric matrices there are several procedures where modifications of elementary row operations are the major computation. Thus, the figures for matrix inversion are increased to include the extra computation. Control is essentially the same as in matrix inversion.

2.2.2.17 Summary

The notation used in the summary is defined as follows:

Description refers to the previous discussion.

An X in the column headed "local" implies more than 50% of the operations could not be systematized to allow global control or the use of global control would have more than doubled the concurrent time.

An x in the column headed "global" implies more than 50% of the operations could be performed such that only one type of operation was performed in a given execution cycle. The " local" and "global" categories are disjoint but not all encompassing.

Sequential time indicates the number of sequential operations required to complete the computation.

Concurrent time indicates the minimum number of operation times required to complete the computation when the maximum degree of concurrency is used.

Maximum concurrency is a measure of the maximum number of operations that could be performed during a given computation cycle.

TABLE 1

SUMMARY OF OPERATIONS

	DESCRIPTION	LOCAL	GLOBAL	SEQUENTIAL TIME	CONCURRENT TIME	MAXIMUM CONCURRENC
1.	Sum or produce of N numbers.		X	N-1	$[\log_2 N]$	N/2
2.	Evaluate N^{th} degree polu-nomials.			2N	$2[\log_2 N]$	N
3.	Multiply vector by matrix.		X	$2N^2$	$[\log_2 N]+1$	N^2
4.	Multiply two matrices.		X	$2N^3$	$[\log_2 N]+1$	N^3
5.	Matrix inver-sion.	X		$5N^3$	3N	$2N^2$
6.	Solving system of linear equa-tions.	X		$5N^2(N+K)$	3N	$2N(N+K)$
7.	Solving ordinary differential equations.		X	$12N^2K$	$4(4+[\log_2 N])K$	$N^2/2$
8.	Least square bit.		X	$2NK^2+5K^3$	2N+3K	$2K^2$
9.	Compute the N^{th} prime.			$N[\ln N]^2$	N	$N/[\ln N]$
10.	Unordered search.		X	$2[\log_2 N]$	4	2N
11.	Sort N elements.		X	$N[\log_2 N]$	2N	4N
12.	Coefficients of Fourier series.			see example on page 49		
13.	Evaluating Fourier series.		X	CS·N	$[\log_2 N]$+CS+2	2N
14.	Neutron diffusion eqn. 2-dimensional 3-dimensional			$21N^2$ $30N^3$	6N 9N	$4N_2$ $4N^2$
15.	Neutron transport equation.	?	?	$2N^2+KN$	$2[\log_2 N]+[\log_2 K]$	$2N^2$
16.	Eigenvalues of a matrix.	X		$30N^3$	18N	$2N^2$

Table 1 summarizes the sequential time, concurrent time, and maximum concurrency possible in the sixteen different operations studied.

2.2.3 Machine Implementation Via a Translation Algorithm

Many computers with the ability to simultaneously perform a number of arithmetic computations have been proposed.[13,14,16,17] Some have been manufactured and others are in various stages of development. In most cases, these machines have no predecessors with programming compatibility. This creates a need for many programs to be rewritten and new programs developed. Thus, it is desirable to have an ALGOL-type language available so that people other than "the experts" can gain early access to the machine.

The purpose of this paper is to present a method of extending current translation techniques to obtain object programs for multi-processor computers, while keeping the translator straightforward and fast. A specific translation algorithm which has been implemented on an IBM 7090 is used for expository purposes. It will become obvious that there are many variations possible in table structures, scanning methods, and intermediate translations. The purpose here is to present the complete picture of producing a code for multiprocessor computers via height assignment rather than to justify the particular implementation.

The translation algorithm can be divided into three basic phases. Phase 1 reduces the input statement to tabular form, phase 2 generates sequences of operations from the tables of phase 1, and phase 3 fills in addresses to generate the translated program.

Phase 1 processing includes reading statements, determining the statement type, and applying the decomposition routine appropriate to that type. Substi-

tution type statements are decomposed into tabular form, quintuple and symbol tables, by the expression scan routine. Iteration and conditional type statements are reduced to control quintuples, generated directly, plus quintuples from using the expression scan on the appropriate parts of the statement. All statements in a program are processed by phase 1 before phase 2 processing begins. At the end of phase 1 the original statements are no longer needed since the sequence of quintuples contains all of the computational information. Information from declarative type statements is preserved in appropriate tables. Figure 31, which will be explained later, for a specific example, shows the result of the phase 1 translation.

Phase 2 processing converts each quintuple to machine operations. Each quintuple consists of two operands, an operator, and two pieces of height information. By a table lookup procedure each operator which appears in a quintuple is converted to a sequence of machine operations. The operands and created temporary locations may appear as data adresses for the machine operations. The height information us used to create a merge-link table, Fig. 36, which is used by phase 3 to generate sequencing addresses. Phase 2 processes one block of quintuples (statements), then phase 3 does the final translation on the code for that block. A block is defined as the largest sequence of quintuples which does not contain a label or a conditional jump operator. Intuitively, if any statement in a block is executed, all statements in the block must be executed. Figure 34, shows an example of the result of the phase 2 translation.

Phase 3 processing uses the merge-link table to set up sequencing addresses. The most general translation algorithm is described, assuming an unlimited num-

ber of processors are available. By a trivial modification, the results of T. C. Hu[18] can be applied in phase 3 to produce code for a computer with any fixed number of processors. In the example described later and in Figure 37, symbolic addresses and operation codes are used for exposition. Sufficient information is known after phase 2 so that completely numeric code could be produced.

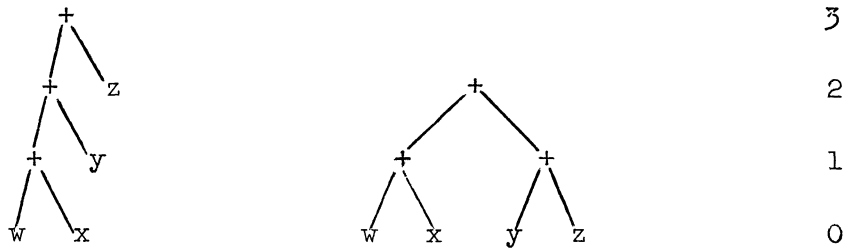
Figure 39 gives a macroscopic flow diagram of the general translation algorithm.

2.2.3.1 Phase 1. Recognition of Concurrency

The previous section could apply to a translation algorithm for a conventional computer. The significant features needed in the translation algorithm for a multiple processor computer are the use of the block and assignment of heights to operands and partial results.

The translation algorithm uses a one pass scan of a statement applying a set of simple rules. Before giving the rules, an intuitive understanding is more readily obtained from the tree representation of a few statements.

Consider the statement w + x + y + z and two of its possible tree representations:



Notice that the tree on the left computes w + x then adds this to y then adds

the sum to z obtaining the result in 3 execution cycles. Thus, the height of this tree is 3 as indicated at the far right. Also, only one arithmetic unit was required during each execution cycle.

Now compare the tree on the right which simultaneously computes w + x and y + z during the first execution cycle, then adds the two sums during the second execution cycle. Thus, the same numerical result is obtained, but by having more than one processor, two arithmetic operations could be performed during the same execution cycle.

In the next section all information will be presented as quintuples which are equivalent ways of representing trees. Each quintuple represents a computation of a partial result as does each operator in the tree. The quintuples corresponding to the trees above are:

R1	w + x	0	1	R1	w + x	0	1
R2	R1 + y	1	2	R2	y + z	0	1
R3	R2 + z	2	3	R3	R1 + R2	1	2

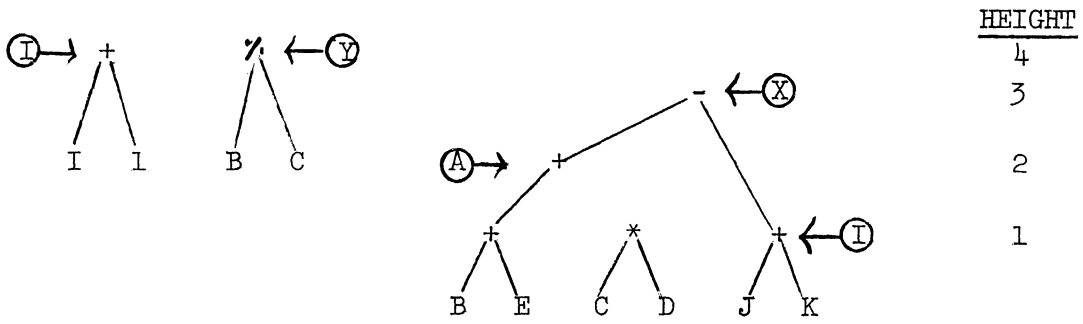
The first column is an implicit numbering of the quintuples (rows of the table). The next five columns are respectively: operand 1, operator, operand 2, starting height, ending height. The starting height corresponds to the execution cycle during which the operation is performed. The ending height corresponds to the execution cycle in which the partial result represented by a quintuple may be used.

In addition to concurrency within a single statement, there can be concurrency between statements in the same block. There cannot be concurrency be-

tween statements of different blocks since the sequencing of blocks can be a function of data and thus is not generally known at translation time.* An example of two blocks and their trees is given below:

Statements

```
In      A = B + C * D + E
        I = J + K
        X = A - I
Out     I = I + 1
        Y = B/C
```



*Using a combined execution and translation mode of operation suggested by A. J. Perlis, this restriction could be eliminated.

2.2.3.2 Expression Scan

Almost every executable statement will require some use of the expression scan. It is in the expression scan that the algebraic notation of a statement is reduced to quintuples. All recognition of concurrency occurs during this scan.

There are a number of cases where the previous discussion must be expanded.

These cases include requiring all arguments to be computed before a function call may be executed, compensating the height values if an operation requires more than one execution cycle, etc. These cases will be defined by a set of rules for performing the expression scan. The rules will be more precise and clear than a prose description of each case.

Because function calls and subscription are allowed in expressions, each sumbol must have heights associated with it. The occurrence height of a symbol is the largest numbered execution cycle in which the symbol was used. The availability height of a symbol is the execution cycle when the symbol may next be used. In the example, Fig. 31 has the symbol table and heights for each set of quintuples. Constants always have occurrence and availability heights of zero. At the beginning of each block all heights in the symbol table are set at zero.

a. Precedence Scan

The statement is scanned from right to left using a well-known precedence scan.[1] The essential features of the precedence scan needed to understand the following discussions are:

1) A simple push down list is established which is denoted hereafter as "LIST." Figure 33 shows the status of the LIST for the example which follows.

2) Precedences are established for operators and punctuation which are not handled as special cases. The relative magnitude of precedences must be such that addition has lower precedence than multiplication. Also addition and subtraction must have the same precedence as must multiplication and division.

3) The purpose of the precedence scan is to reduce the input statement to tree form. As was shown earlier, this tree is not always unique and thus, height considerations can be used to generate the tree with lowest height.

The flow diagram of the precedence scan is given in Fig. 40. The references on the flow diagram to 7.1, 7.2, and 7.3 refer to application of the height rules which follow.

b. Weight Determination Rules

The procedure to be applied at point 7.3 on Fig. 40 is:

1) Scanning the LIST from left to right, sleect the first two operands such that no other operands have lower height. In comparing two operands of equal height, preference is given to row references over variables or constants. Denote the leftmost operand selected as A and denote the other as B. Do not scan past an operator with precedence different from the leftmost operator on the LIST. The form of the LIST during this scan is alternate operands and operators.

2) If there is a subtraction or division operator immediately to the left of the A operand then the operator immediately to the left of the B operand is replaced by its dual.

3) If the B operand has lower height than the A operand and the operator is commutative, then the A and B operands are interchanged on the LIST. In comparing two operands of equal height, preference is given to row references over variables or constants.

4) A quintuple is produced consisting of:

i the A operand
ii the operator immediately to the left of B
iii the B operand
iv the maximum of the height of A and B (START)
v the maximum of the heights of A and B plus 1 (END)

Note that either A or B can be variables or row references, thus the height used would be the availability or ending height, respectively.

5) If neither A nor B is a row reference, item iv is the maximum of the height of A and the height of B minus 1. Item v is the new item iv plus 2.

6) If the operator is "≡" and B is a row reference, and the starting height

of the last quintuple of the chain of first operands which are row references beginning with B is greater than the occurrence height of A, then item v is set equal to item iv.

7) The quintuple is assigned the next row in the quintuple table. This row reference replaces the A operand on the LIST. The B operand and the operator immediately to its left are removed from the list.

8) If the A operand is a variable and its occurrence height is less than item iv, then the occurrence height of A is set equal to item iv. If the B operand is a variable and its occurrence height is less than or equal to item iv, the occurrence height of the B operand is set equal to item iv plus 1.

9) If the operator is "=", the available and occurrence height of the A operand is set equal to item v.

10) Repeat 1) through 9) until the precedence of the first operator on the LIST is different from the precedence of the first operator on the LIST before these rules were applied. The procedure to be applied at point 7.4, unary operator, on Fig. 40 is:

1) The quintuple produced consists of
   - i a blank operand
   - ii the operator being scanned from the statement
   - iii the first item on the LIST (operand)
   - iv the height of the first item on the LIST
   - v the height of the first item on the LIST plus 1

Note that the operand may be a variable or a row reference, thus the availability or ending heights, respectively, are used.

2) If the operand is a variable, item v is increased by one. Further, if the operand is a variable and its occurrence height is less than or equal to item iv, the occurrence height of the operand is set equal to item iv plus 1.

The procedure to be applied at point 7.2, function call, of Fig. 40 is:

1) The arguments for the function call are on the LIST enclosed in parenthesis and separated by commas. An argument may be either a variable or row reference. The arguments are scanned from left to right until the closing parenthesis is encountered. The first quintuple produced consists of

i the name of the function
ii the CALL operation
iii the return height of the function (next merge number)
iv the maximum height of the arguments and function name
v the same as iii

2) The second quintuple generated consists of

i the row reference of the first quintuple
ii the operation PAR (denoting parameter)
iii the first argument
iv blank
v blank

3) The third and successive quintuples consist of

i the next parameter
ii the operation PAR
iii the next parameter after item i if any
iv blank
v blank

4) For those arguments which are variables and have occurrence height less than or equal to the maximum heights of the arguments, their occurrence heights are set to one greater than the maximum heights of the arguments.

5) The availability height of the function is set to v.

The procedure to be applied at point 7.1, subscription, on Fig. 40 is:

1) If there is only one subscript, then a quintuple is generated which consists of:

i the variable to be subscripted
ii the operation ↓
iii the subscript
iv the availability height of the subscript
v the maximum of iv plus 1 and the availability height of the subscripted variable.

2) If there is more than one subscript, the same procedure as the function call quintuple generation is used. The variable to be subscripted is used in place of the function name except for the additional condition on item v above. The number of subscripts juxtaposed to an ↓ is used in place of the CALL. The subscripts are treated as arguments.

A refinement must be made to the definition of height to allow for variable heights. A variable height is necessary for the returned value from a function call. The number of execution cycles required by a function is not always known and is usually dependent on values of the argument. The technique used is to assign merge numbers to each height value. If a maximum availability height is required and the two heights being compared are of different merge numbers, a merge quintuple is generated and the maximum height is given a new merge number. All rules given previously apply directly to sets of heights with the same merge number. The merge number is set to zero at the beginning of each block.

2.2.3.3 Phase 2 of Translation Algorithm

After all statements have been processed by phase 1, the generation of machine code from the quintuples begins. Phase 2 is essentially a table lookup procedure. For each operation that may appear in a quintuple there is a sequence of machine instructions to be inserted. Either/or both symbolic and numeric machine instructions can be generated.

Each quintuple is examined and the corresponding sequence of machine instructions is located. Flags set from previous sequences can be used to recognize instructions which may be deleted or changed to produce more efficient code.

This process is very dependent on the specific machine* for which the object code is being prepared. Insertion of one or both operands from a quintuple are made as designated by flags in the sequence of instructions. In case a row reference is to be inserted, a temporary location is assigned. As each temporary is assigned and used, pertinent height information is retained. In phase 3 of the translation temporaries are reused as much as the logic of the program permits.

As each instruction is generated its position in the execution sequence is computable from the height information in the quintuple. For each instruction produced, an entry in the merge-link table is augmented. The merge-link table consists of entries which count the number of instructions for each (merge number, height) pair. The merge-link table is cleared at the beginning of each block.

2.2.3.4 Phase 3 of Translation Algorithm

When all quintuples in a block have been processed, phase 3 processing begins a scan of the instructions generated by phase 2. Using the information in the merge-link table, a successor address is appended to each instruction. Each instruction is transferred from phase 2 to phase 3 with its execution number in the form of a (merge, height, number) triple. If the merge-link table has a positive count in the entry for the same merge number and one greater height, then a successor address is generated and inserted. If there is a successor for a given instruction, a zero is inserted as the successor address. Each time a

*A specific set of instructions is presented in the example.

non zero successor address is generated, the corresponding count in the merge-link table is decreased by 1.

The condition may arise when the last instruction of a given merge number and height is encountered and the count in the merge-link table for the same merge number and one greater height is greater than 1. In this case, an indirect address is generated and placed in the successor address of the instruction. The required number of indirect address words are generated to form a tree. The tree begins with the indirect successor address and terminates at the remaining locations indicated by the count in the merge-link table.

If the last entry in a row of the merge-link table is a merge designation, an indirect successor address tree is generated. The tree begins with the last instruction for that merge row and terminates at all the instructions of the merge designation with height zero. The beginning of a block is a special case of the above. Each zero merge number, zero height instruction is at the termination of an indirect successor address tree which begins with the final instruction of the previous block.

2.2.3.5 Machine Instructions

Continuing with the philosophy of presenting a specific example, the following instructions were chosen to implement the final phase of the translation.

In all cases γ is the location of an instruction to be executed during the next execution cycle. The next execution cycle does not begin until all instructions being executed have been completed. A zero for a γ address indicates an instruction has no successor.

List of Operations

Operation	Addresses	Description
ADD	α,β,γ	Add contents of α to contents of β and place result in α.
SUBTR	α,β,γ	Subtract contents of β from α and place result in α.
MULT	α,β,γ	Multiply contents of α by contents of β and place result in α.
DIVIDE	α,β,γ	Divide contents of α by contents of β and place result in α.
LOAD	α,β,γ	Replace contents of α by contents of β.
LOADA	α,β,γ	The address, β, replaces the address portion of the contents of α.
ADDRM	α,β,γ	Address modification: the address, α, is added to the contents of β and the result is retained as an address δ until the next execution of this instruction. If this instruction is referred to via an indirect address, Δ is used as the effective address.
COUNT	α,β,γ	One is added to α, if this sum equals β then α is set to zero and γ is interpreted normally. If the sum is unequal to β, then $\alpha + 1$ replaces α and γ is treated as a zero for the current execution cycle.
INDADR		A pseudo operation which generates a tree of indirect addresses with a four to one expansion on each level of the tree.
SYN		A pseudo operation which appears only in symbolic code to indicate several variable names refer to the same machine location.

A = A * (B+C*C+D)/(B+C)/D

Row	Operand	Operator	Operand	Start	End
R1	B	+	C	0	2
R2	C	*	C	0	2
R3	B	+	D	0	2
R4	R3	+	R2	2	3
R5	A	/	D	0	2
R6	R5	/	R1	2	3
R7	R6	*	R4	3	4
R8	A	=	R7	4	4

Symbol	Occur.	Avail.
A	4	4
B	0	0
C	1	0
D	1	0

B = (C*B-D-E)/D*E+A

R9	C	*	B	0	2
R10	D	+	E	0	2
R11	R9	-	R10	2	3
R12	D	/	E	0	2
R13	R11	/	R12	3	4
R14	R13	+	A	4	5
R15	B	=	R14	5	5

A	4		4
B	5		5
C	1		0
D	1		0
E	1		0

A = 3+C

R16	3	+	C	0	2
R17	A	=	R16	4	5

A		5	5

Fig. 31. Example of phase 1 of translation (quintuple generation).

C(I) = F·(D,E(I),I+J)+G·(A(J))*C(J)

Row	Operand	Operator	Operand	Start	End
R18	C	↓	J	0	1
R19	A	↓	J	0	5
R20	G	CALL	1 0	5	1 0
R21	R20	PAR	R19	-	-
R22	1 0	MERGE	1	-	1 1
R23	R20	*	R18	1 1	1 2
R24	I	+	J	0	2
R25	E	↓	I	0	1
R26	F	CALL	2 0	2	2 0
R27	R26	PAR	D	-	-
R28	R25	PAR	R24	-	-
R29	2 0	MERGE	1 2	-	2 1
R30	R26	+	R23	2 1	2 2
R31	C	↓	I	0	1
R32	R31	=	R30	2 2	2 3

Symbol	Occur.	Avail.
A	5	5
B	5	5
C	2 3	2 3
D	3	0
E	3	0
F	2	2 0
G	5	1 0
I	0	0
J	1	0

C(I,A+J) = G·(I)

R33	G	CALL	3 0	1 1	3 0
R34	R33	PAR	I	-	-
R35	J	+	A	0	8
R36	C	↓2	4 0	8	4 0
R37	R36	PAR	I		
R38		PAR	R35		
R39	4 0	MERGE	3 0	-	4 1
R40	R36	=	R33	4 1	4 2

Fig. 32. Example of phase 1 of translation (quintuple generation concluded).

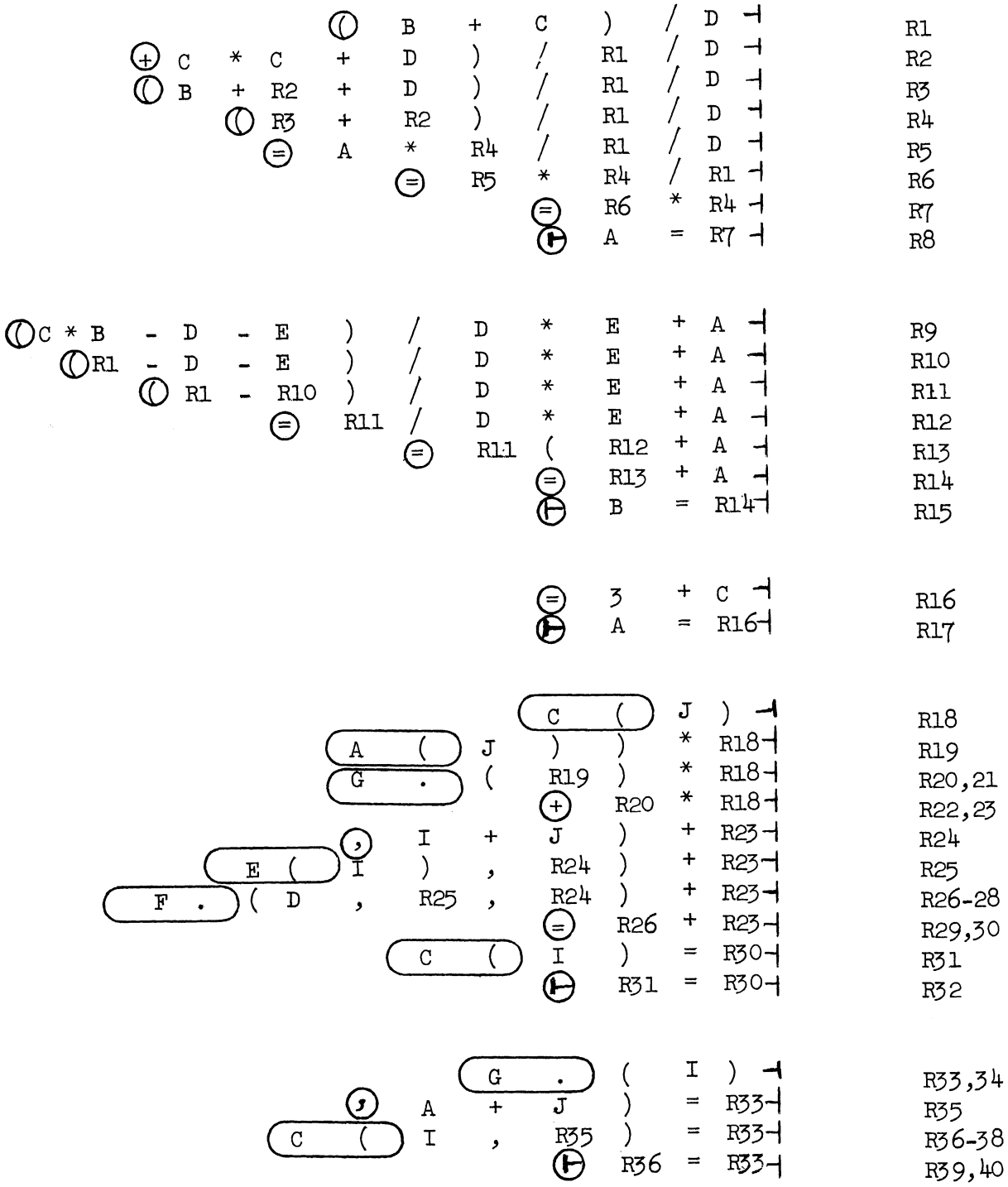


Fig. 33. Status of list before each quintuple is generated. (Circled items are triggers for generation of quintuples.)

Execution Merge #, Cycle #,##	Operation Code	Operand & Result	Operand
	A = A*(B+C*C+D)/(B+C)/D		
0,0,1	LOAD	T1,B	R1
0,1,1	ADD	T1,C	
0,0,2	LOAD	T2,C	R2
0,1,2	MULT	T2,C	
0,0,3	LOAD	T3,B	R3
0,1,3	ADD	T3,D	
0,2,1	ADD	T3,T2	R4
0,0,4	LOAD	T5,A	R5
0,1,4	DIVIDE	T5,D	
0,2,2	DIVIDE	T5,T1	R6
0,3,1	MULT	T5,T3	R7
	CHANGE	T5,A	R8
	B = (C*B-D-E)/D*E+A		
0,0,5	LOAD	T9,C	R9
0,1,5	MULT	T9,B	
0,0,6	LOAD	T10,D	R10
0,1,6	ADD	T10,E	
0,2,3	SUBTR	T9,T10	R11
0,0,7	LOAD	T12,D	R12
0,1,7	DIVIDE	T12,E	
0,3,2	DIVIDE	T9,T12	R13
0,4,1	ADD	T9,A	R14
	CHANGE	T9,B	R15
	A = 3+C		
0,0,8	LOAD	T16,3	R16
0,1,8	ADD	T16,C	
0,4,2	LOAD	A,T16	R17

Fig. 34. Example of phase 2 of translation (preliminary code generation).

Execution Merge #, Cycle #**	Operation Code	Operand & Result	Operand
	C(I) = F·(D,E(I),I+J)+G·(A(J))*C(J)		
0,0,9	ADDRM	C,J	R18
0,0,10	ADDRM	A,J	R19
0,5,1	CALL	G',(1,0,1)	R20
0,5,2	LOADA	G+1',T20	R21
0,5,3	LOADA	G+2',(0,0,10)'	
1,0,1-0,1,9	MERGE	0,2,(1,1,1)	R22
1,1,1	MULT	T20,(0,0,9)'	R23
0,0,11	LOAD	T24,I	R24
0,1,10	ADD	T24,J	
0,0,12	ADDRM	E,I	R25
0,2,4	CALL	F',(2,0,1)	R26
0,2,5	LOADA	F+1',T26	R27
0,2,6	LOADA	F+2',D	
0,2,7	LOADA	F+3',(0,0,12)'	R28
0,2,8	LOADA	F+4',T24	
2,0,1-1,2,1	MERGE	0,2,(2,1,1)	R29
2,1,1	ADD	T26,T20	R30
0,0,13	ADDRM	C,I	R31
2,2,1	LOAD	(0,0,13)',T26	R32
	C(I,A+J) = G·(I)		
1,1,2	CALL	G',(3,0,1)	R33
1,1,3	LOADA	G+1',T33	R34
1,1,4	LOADA	G+2',I	
0,0,14	LOAD	T35,J	R35
0,5,4	ADD	T35,A	
0,6,1	CALL	↓2',(4,0,1)	R36
0,6,2	LOADA	↓2+1',T36	R37
0,6,3	LOADA	↓2+2',I	
0,6,4	LOADA	↓2+3',T35	R38
4,0,1-3,0,1	MERGE	0,2,(4,1,1)	R39
4,1,1	LOAD	T36',T33	R40

Fig. 35. Example of phase 2 of translation (preliminary code generation concluded).

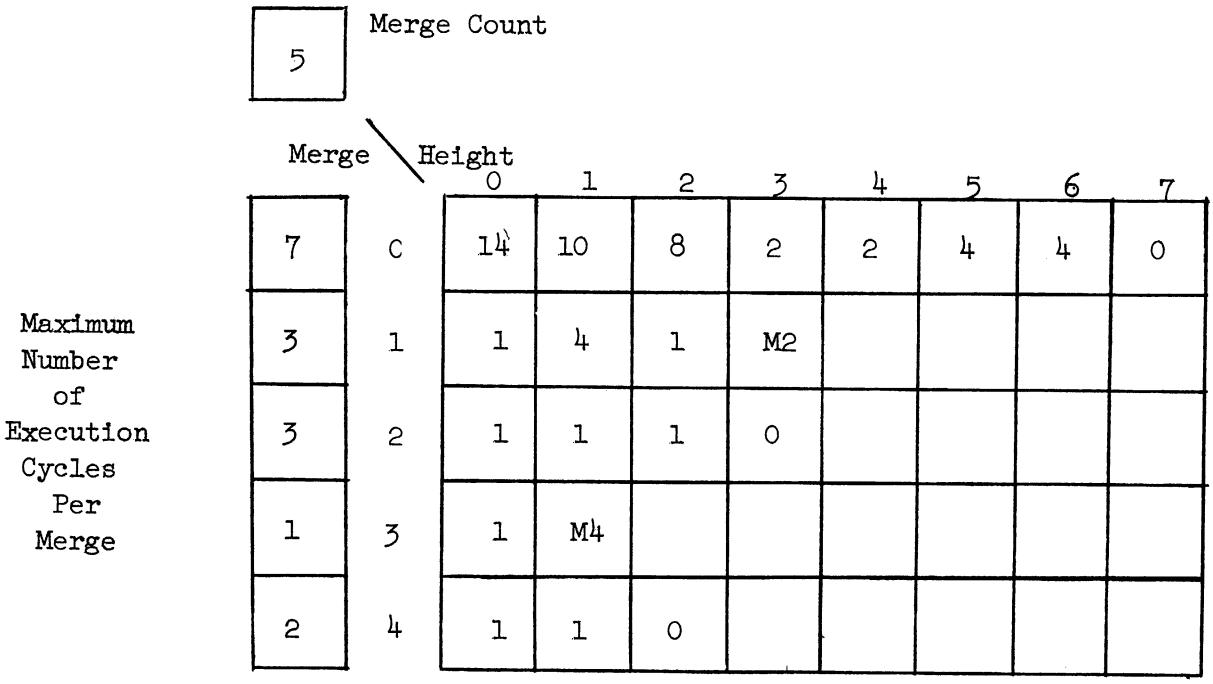


Fig. 36. Merge-link table at end of phase 2.

Loading Address	Operation Code	Operand, Operand, Successor
S0000	INDADR	S0001' ... S0014'
S0001	LOAD	T1,B,S0101
S0101	ADD	T1,C,S0201
S0002	LOAD	T2,C,S0102
S0102	MULT	T2,C,S0202
S0003	LOAD	T3,B,S0103
S0103	ADD	T3,D,S0203
S0201	ADD	T3,T2,S0301
S0004	LOAD	A,A,S0104
S0104	DIVIDE	A,D,S0204
S0202	DIVIDE	A,T1,S0302
S0301	MULT	A,T3,S0401
S0005	LOAD	T9,C,S0105
S0105	MULT	T9,B,S0205
S0006	LOAD	T10,D,S0106
S0106	ADD	T10,E,S0206
S0203	SUBTR	T9,T10,0
S0007	LOAD	T12,D,S0107
S0107	DIVIDE	T12,E,S0207
S0302	DIVIDE	B,T12,S0402
S0401	ADD	B,A,S0501
S0008	LOAD	T16,3,So108
S0108	ADD	T16,C,S0208
S0402	LOAD	A,T16,S0500
S0500	INDADR	S0502' ... S0504'
S0009	ADDRM	C,J,S0109
S0010	ADDRM	A,J,S0110
S0501	LOADA	G',S1001,G
S0502	LOADA	G+1',T20,0
S0503	LOADA	G+2',S0010',0
S0109	SYN	S1001
S1001	COUNT	0,2,S1101
S1101	MULT	T20,S0009',S1201

Fig. 37. Final program after phase 3 of translation (symbolic and/or numeric).

Loading Address	Operation Code	Operand, Operand, Successor
S0011	LOAD	T24,I,0
S0110	ADD	T24,J,0
S0012	ADDRM	E,I,0
S0204	LOADA	F',S2001,F
S0205	LOADA	F+1',T26,0
S0206	LOADA	F+2',D,0
S0207	LOADA	F+3',S0012',0
S0208	LOADA	F+4',T24,0
S1201	SYN	S2001
S2001	COUNT	0,2,S2101'
S2101	ADD	T26,T20,S2201
S0013	ADDRM	C,I,0
S2201	LOAD	S0013',T26,S2301
S1102	LOADA	G',S3001,G
S1103	LOADA	G+1',T33,0
S1104	LOADA	G+2',I,0
S0014	LOAD	T35,J,0
S0504	ADD	T35,A,5604
S0601	LOADA	↓2',S4001,↓2
S0602	LOADA	↓2+1,T36,S0701
S0603	LOADA	↓2+2,I,0
S0604	LOADA	↓2+3,T35,0
S3001	SYN	S4001
S4001	COUNT	0,2,S4101
S4101	LOAD	T36',T33,S0401
S0701	SYN	S9999
S2301	SYN	S9999
S4201	SYN	S9999
S9999	COUNT	0,3,___ Entry to next block

Fig. 38. Final program after phase 3 of translation (Concluded).

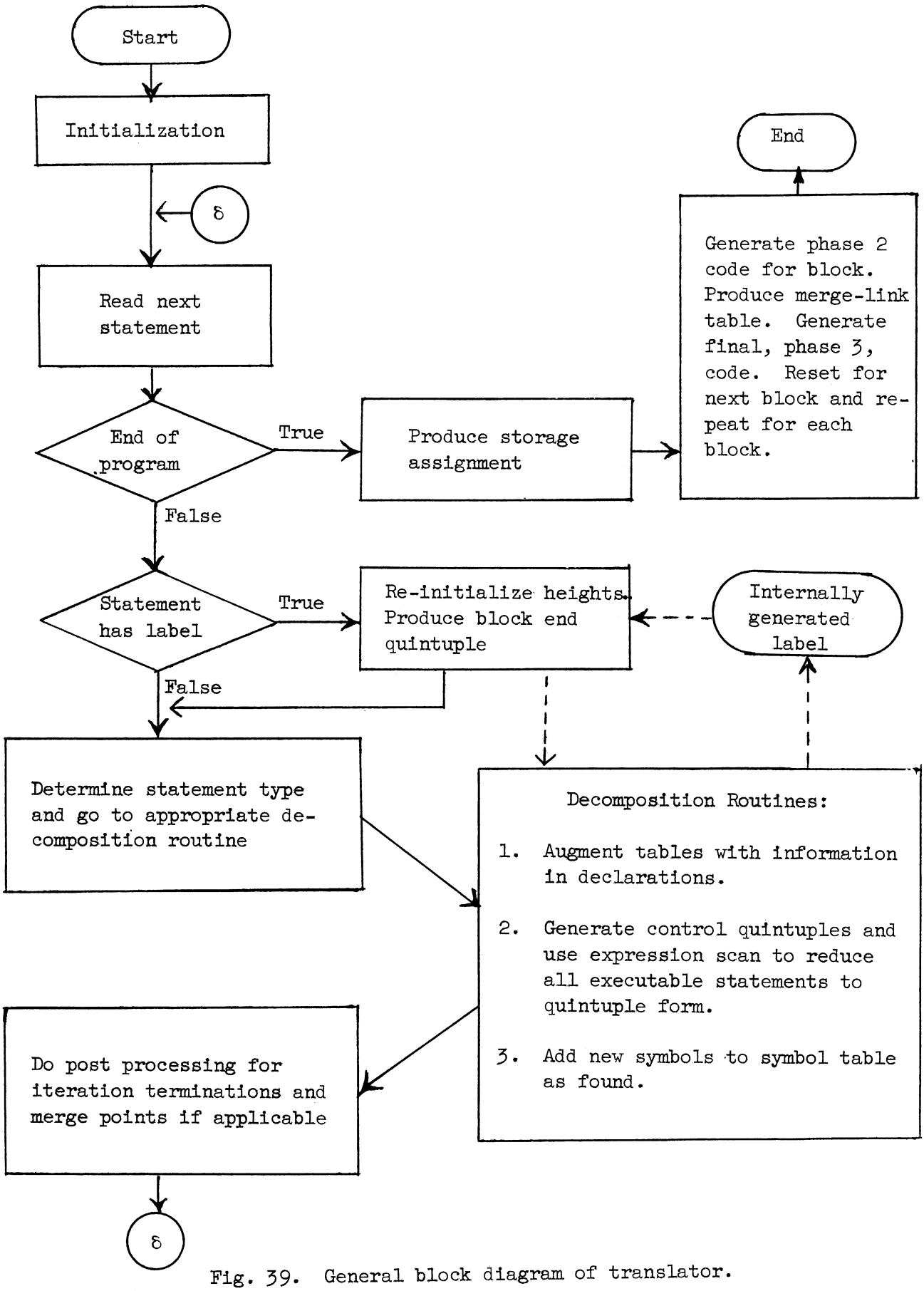


Fig. 39. General block diagram of translator.

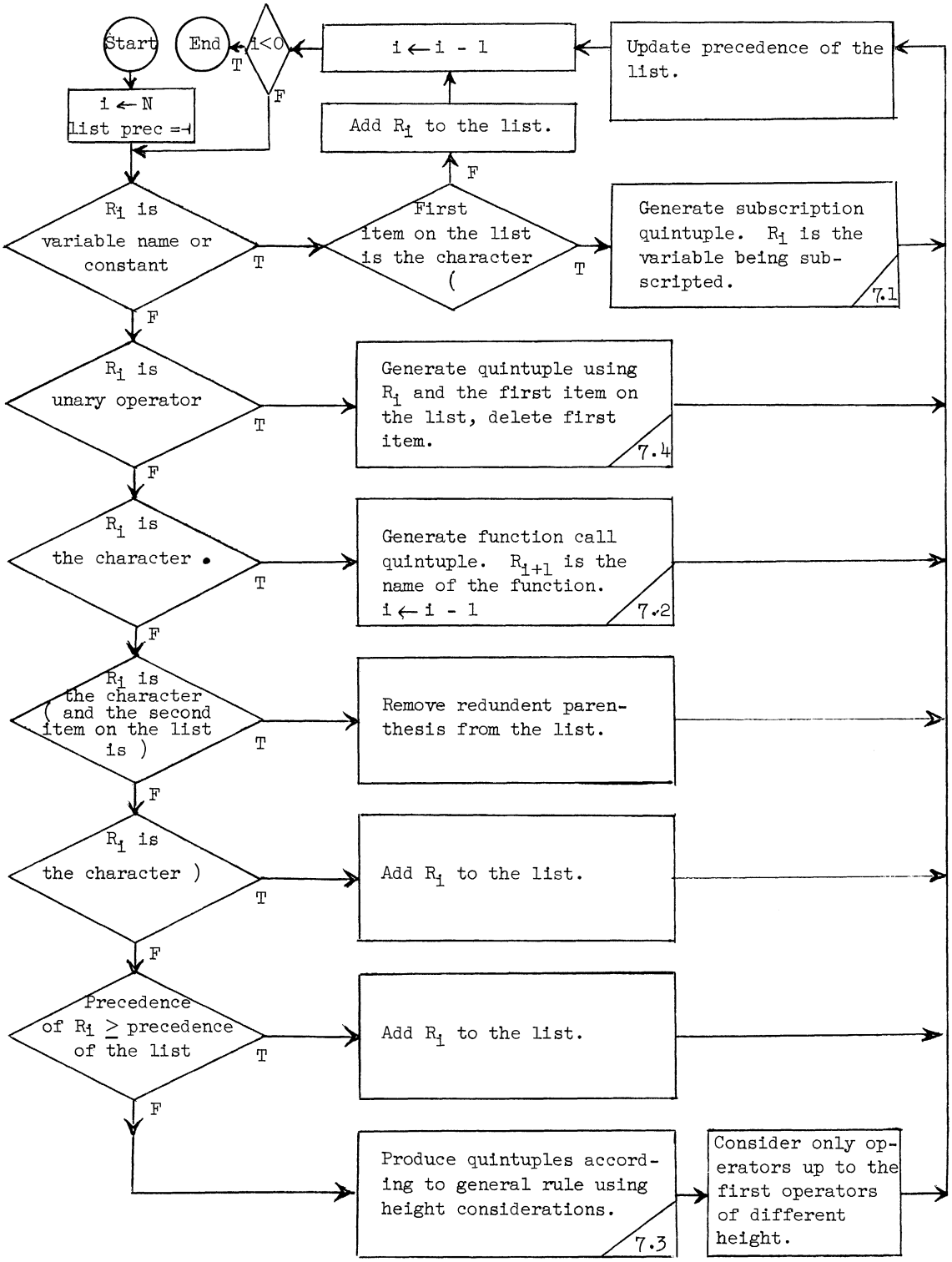


Fig. 40. Expression scan flow diagram.

The approach taken in this section has been considerably different from the sequencing procedures of Schwartz[15] and others.

In evaluating the translation technique presented here, the following points are noteworthy: (1) The translation algorithm does not vary radically from several translators now in use, thus utilizing past experience can keep developmental time to a minimum. (2) The translation is fast since statements and quintuples are each scanned only once. (3) The object programs are optimal in the sense that no faster code could be produced within the limits of the input language.

2.2.4 An Augmented Language to Permit More Concurrence in Processing

It was indicated in 2.2.3 that a number of computers are being developed which have facilities for doing concurrent arithmetic operations. It was also shown that an augmented language is needed if machine translation from an ALGOL-type language is to approach the efficiency of object programs obtainable by programmers. There are a number of ALGOL implementations on conventional computers and ALGOL is a fairly well accepted publication language for algorithms. Since there is no established language of the algebraic type for these computers, it is time to introduce concurrency into ALGOL. Although the following additions are directed at hardware representation, they conform to the seven guidelines for the reference language.

A set of additions are proposed below which apply to the ALGOL GO report.[26] For lack of a better name, these additions are called Concurrent Report on Algorithmic Language CALGOL 60. The additions below appear in the order which applies to the report.[26] The notation ... indicates: repeat same information as in re-

port. The numerical indications in the following refer to the original report.[26]

- -

2.3. DELIMITERS

< operator > :: = ... |< concurrent operator >

< concurrent operator > :: = &

< sequential operator > :: = ... |commence if|halt

4.1.1 Statement Syntax

< statement > :: = ... |< concurrent statement >|< commence statement >

|< halt statement >

4.8 Concurrent Statements

4.8.1 Syntax

< concurrent statement > :: = < statement > < concurrent operator >
< unlabelled statement >|< concurrent
statement > < concurrent operator >
< unlabelled statement >

< unlabelled statement > :: = < unlabelled basic statement >|< for
statement >|< unlabelled block >|< un-
labelled compound >|< if clause > < uncon-
ditional statement >|< if clause > < uncon-
ditional statement > else < statement >

4.8.2 Examples

1:C: = A+B & for q: = 1 step S until n do X[q]: = B[q]

D: = sin (x) & E: = cos (x) & begin z : = 0; f: = sqrt (x);

if z > f then z: = 2 x f + x end

4.8.3 Semantics

The & indicates that all connected statements may begin execution when the first statement begins execution. It is implied that the computational order of the connected statements is irrelevant to the overall computation of the algorithm. The successor statement is not executed until all connected statements have completed execution. See 4.9 for method of commencing execution before connected statements complete execution.

4.9 Commence Statements

4.9.1 Syntax

< commence statement > :: = commence if < Boolean expression >|

< commence statement > : < label >

4.9.2 Examples

commence if p $\vee$ q

commence if x > y : 51: 57: 512

4.9.3 Semantics

If no labels follow the Boolean expression, the next statement in sequence is executed whenever the Boolean expression has value true. If labels are present, all statements so designated are executed whenever the Boolean expression has value true and the next statement in sequence is not executed.

4.10 Halt Statement

4.10.1 Syntax

< halt statement > :: = halt

4.10.2 Example

halt

4.10.3 Semantics

A sequence of computation is terminated by inserting the statement halt. It is assumed some other sequence is still in the process of computing.

5. Declarations

< declaration > :: = ... | < independent declaration >

5.5 Independent Declaration

5.5.1 Syntax

< independent declaration > :: = independent < type list >|

independent < type declaration >| independent < array declaration >

5.5.2 Examples

independent A,B,C

independent own integer array D[5:n]

5.5.3 Semantics

Declaring a variable or array to be independent allows execution of instructions relating to the variable or array in any order. Each occurrence is con-

sidered as a distinct identification. In terms of subscripted variables each subscript is distinct. In terms of arguments in procedure calls, the arguments declared to be independent are not modified by the procedure. The usual conventions about the scope of a declaration within a block apply.

- -

Example

```
procedure sfm (f, n, time); value time;
integer n; independent integer array f[1:2+n];
comment sfm stands for switching function minimization.
```

Two procedures will be initiated simultaneously, one using Quine's method and another using Ashenhurst's decomposition method. There is a maximum time limit for the minimization denoted by the argument "time" in seconds. If either minimization finishes before the allowed time is exceeded, that routine will set its indicator. The corresponding result is then moved into the f region and return is made to the caller. Note that the WAITING indicator is necessary to prevent reactivation of the commence instruction. In case of a tie, the Quine results are used. If time is up before either routine finishes, the result remains the same as the input. Local storage and variables as well as the procedure are given below. The routines of Quine and Ashenhurst are conventional procedures which set Q and A respectively to true before returning to caller. The independence declaration is used for f,g,h, and i to allow simultaneous transfer of results into the f region.

```
begin    independent integer array g:h[1:2↑n]
```

```
Boolean T,Q,A, WAITING;
independent interger i;
WAITING;  =  true; T: = Q: = A: = false;
Quine (f,n,g,Q) & Ashenhurst (f,n,h,A) &
     T: = timeup (time);  halt;
commence if WAITING (T  Q A);
WAITING:  =  false;
if Q then for i:  =  1 step 1 until 2↑n do f[i]:  =  g[i]
else if A then for i:  =  1 step 1 until 2 n do f[i]:  =  h[i];
     end sfm.
```

Example for Algorithm 7

Problem: Given an n loop network of R,L,C fixed with respect to time and voltage sources which vary as functions of time (assuming network quiescent at t = 0), find all loop currents as a function of time. Numerical Solution:

Using Kirchoff's law around each loop, n second order differential equations are obtained. These equations are of the form shown below:

$$Q''_1 = a_{11}Q'_1 + a_{12}Q'_2 + \ldots + a_{1n}Q'_n + b_{11}Q_1 + b_{12}Q_2 + \ldots + b_{1n}Q_n + f_1(t)$$

$$Q''_2 = a_{21}Q'_1 + a_{22}Q'_2 + \ldots + a_{2n}Q'_n + b_{22}Q_1 + b_{22}Q_2 + \ldots + b_{2n}Q_n + f_2(t)$$

$$\vdots$$

$$Q''_n = a_{n1}Q'_1 + a_{n2}Q'_2 + \ldots + a_{nn}Q'_n + b_{n1}Q_1 + b_{n2}Q_2 + \ldots + b_{nn}Q_n + f_n(t)$$

The a's and b's are constants depending only on the values of the RLC com-

ponents of the network. $f_1(t) \ldots f_n(t)$ must be evaluated at each time step.

The initial conditions are $Q'_1 = Q'_2 = \ldots = Q'_n = Q_1 = Q_2 = \ldots = Q_n = 0$

The algorithm for the numerical solution for the loop currents $Q'_1, \ldots, Q'_n$ is given below:

(The notation ${}^{j}Q''_i$ means the value of Q''_i computed during the j^{th} time step, i.e., $t = j \times H$ where H is the basic time step.).

$$\left.\begin{aligned} {}^{j}_{i}K_1 &= \frac{H}{2} \times {}^{j}Q''_i\,(t, {}^{j}Q'_i, {}^{j}Q_i) \\ {}^{j}_{n+i}K_1 &= \frac{H}{2} \times {}^{j}Q'_i \\ t &= t + \frac{H}{2} \end{aligned}\right\} \quad \text{do for } i = 1, n$$

$$\left.\begin{aligned} {}^{j+1/4}Q'_i &= {}^{j}Q'_i + {}^{j}_{i}K_1 \\ {}^{j+1/4}Q_i &= {}^{j}Q_i + {}^{j}_{n+i}K_1 \end{aligned}\right\} \quad \text{do for } i = 1, n$$

$$\left.\begin{aligned} {}^{j}_{i}K_2 &= \frac{H}{2} \times {}^{j+1/4}Q''_i\,(t, {}^{j+1/4}Q'_i, {}^{j+1/4}Q_i) \\ {}_{n+i}K_2 &= \frac{H}{2} \times {}^{j+1/4}Q'_i \end{aligned}\right\} \quad \text{do for } i = 1, n$$

$$\left.\begin{aligned} {}^{j+1/2}Q'_i &= {}^{j}Q'_i + {}^{j}_{i}K_2 \\ {}^{j+1/2}Q_i &= {}^{j}Q_i + {}^{j}_{n+i}K_2 \end{aligned}\right\} \quad \text{do for } i = 1, n$$

$$
\begin{aligned}
{}^{j}_{i}K_3 &= H \times {}^{j+1/2}Q_i''\left(t, {}^{j+1/2}Q_i', {}^{j+1/2}Q_i\right) \\
{}^{j}_{n+i}K_3 &= H \times {}^{j+1/2}Q_i' \\
t &= t + \frac{H}{2}
\end{aligned}
$$

do for i = 1, n

$$
\begin{aligned}
{}^{j+3/4}Q_i' &= {}^{j}Q_i' + {}^{j}_{i}K_3 \\
{}^{j+3/4}Q_i &= {}^{j}Q_i + {}^{j}_{n+i}K_3
\end{aligned}
$$

do for i = 1, n

$$
\begin{aligned}
{}^{j}_{i}K_4 &= H \times {}^{j+3/4}Q_i''\left(t, {}^{j+3/4}Q_i', {}^{j+3/4}Q_i\right) \\
{}^{j}_{n+i}K_4 &= H \times {}^{j+3/4}Q_i'
\end{aligned}
$$

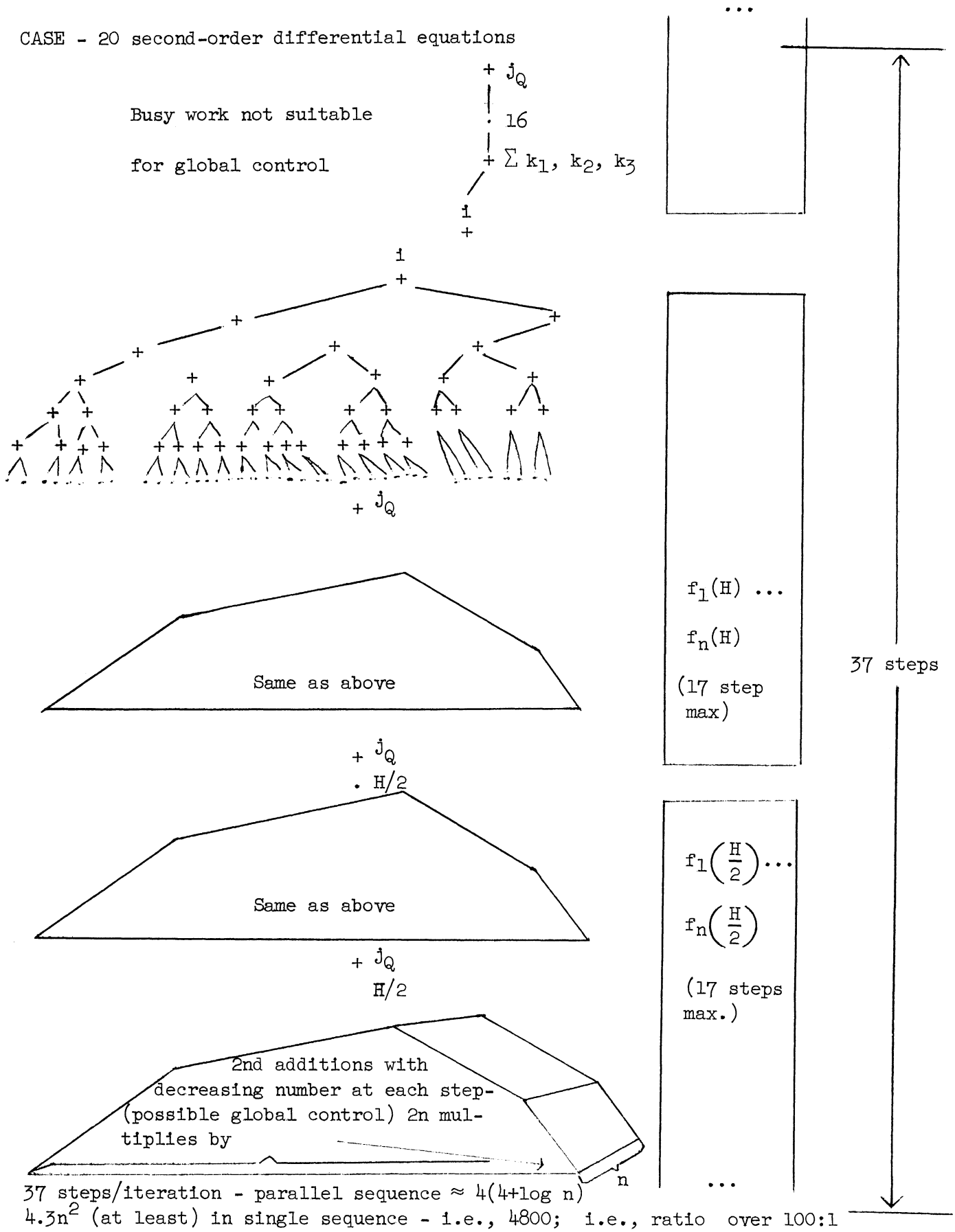
do for i = 1, n

$$
\begin{aligned}
{}^{j+1}Q_i' &= {}^{j}Q_i' + \frac{1}{6}\left({}^{j}_{i}K_1 + 2x\,{}^{j}_{i}K_2 + 2x\,{}^{j}_{i}K_3 + {}^{j}_{i}K_4\right) \\
{}^{j+1}Q_i &= {}^{j}Q_i + \frac{1}{6}\left({}^{j}_{n+i}K_1 + 2x\,{}^{j}_{n+i}K_2 + 2\,{}^{j}_{n+i}K_3 + {}^{j}_{n+i}K_4\right)
\end{aligned}
$$

do for i = 1, n

Go to first step of algorithm until m repetitions have been completed. The solutions are then ${}^{1}Q_i$, ${}^{2}Q_i$, ..., ${}^{n}Q_i$ for i = 1, n.

CASE - 20 second-order differential equations



37 steps/iteration - parallel sequence $\approx 4(4+\log n)$
$4.3n^2$ (at least) in single sequence - i.e., 4800; i.e., ratio over 100:1
20 second-order differential equations

Example for Algorithm 14

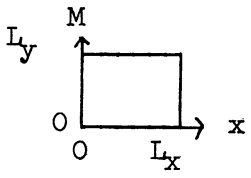
Parabolic Partial Differential Equation (General Diffusion Equation)

$$S(\bar{x},t,\phi)\,\frac{\phi(\bar{x},t)}{t} = \nabla[D(\bar{x},t,\phi)\nabla\phi(\bar{x},t)] + W(\bar{x},t,\phi)$$

1. $$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

$$\begin{aligned} T(0,y,t) &= T_a \\ T(L_x,y,t) &= T_b \\ T(x,0,t) &= T_c \\ T(x,L_y,t) &= T_d \\ T(x,y,0) &= T_e \end{aligned}$$

↖ supplied boundary conditions



2.a. $$\frac{T_{i\,j}^{n+1} - T_{i\,j}^{n}}{\Delta t} = \frac{T_{i-1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i+1,j}^{n+1}}{(\Delta X)^2} + \frac{T_{i,j-1}^{n} - 2T_{i,j}^{n} + T_{i,j+1}^{n}}{(\Delta Y)^2}$$

2.b. $$\frac{T_{i\,j}^{n+2} - T_{i\,j}^{n+1}}{\Delta t} = \frac{T_{i-1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i+1,j}^{n+1}}{(\Delta X)^2} + \frac{T_{i,j-1}^{n+2} - 2T_{i,j}^{n+2} + T_{i,j+1}^{n+2}}{(\Delta Y)^2}$$

$i = 1,2,\ldots,N$ in x direction $\qquad \Delta X \cdot N = L_x$

$j = 1,2,\ldots,M$ in y direction $\qquad \Delta Y \cdot M = L_y$

$n = 1,2,\ldots,\tau$ in temporal direction $\qquad \Delta t \cdot \tau =$ total time of observation

$$\text{Let } S = \frac{\Delta X}{\Delta Y} \qquad \rho = \frac{\Delta X \Delta Y}{\Delta t}$$

2.c. $$T_{i-1,j}^{n+1} - (2+\rho S)\,T_{i,j}^{n+1} + T_{i+1,j}^{n+1} = -D_x$$

2.d. $$T_{i,j-1}^{n+2} - (2+\rho/S)\,T_{i,j}^{n+2} + T_{i,j+1}^{n+2} = -D_y$$

where

$$D_X^i + S^2 T_{i,j-1}^n - (2S^2 - \rho S) T_{i,j}^n + S^2 \cdot T_{i,j+1}^n + \begin{cases} (T_a)_j^{n+1} & \text{if } i = 1 \\ 0 & \text{otherwise} \\ (T_b)_j^{n+1} & \text{if } i = N \end{cases}$$

$$D_Y^j = \frac{1}{S^2} T_{i-1,j}^{n+1} - (2 - \rho S) T_{i,j}^{n+1} + T_{i+1,j}^{n+1} + \begin{cases} (T_c)_i^{n+2} & \text{if } j = 1 \\ 0 & \text{otherwise} \\ (T_d)_i^{n+2} & \text{if } j = M \end{cases}$$

$n+1^{st}$ step

$$\begin{vmatrix} -(2+\rho/S) & 1 & & & \\ 1 & -(2+\rho/S) & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -(2+\rho/S) \end{vmatrix} \times \begin{vmatrix} T_{1,j} \\ T_{2,j} \\ \\ T_{n,j} \end{vmatrix} = \begin{vmatrix} -D_x^1 \\ -D_x^2 \\ \\ -D_x^n \end{vmatrix}$$

N x N system M M

$n+2^{nd}$ step ditto M x M system with y subscripts,

or sequential - Gaussian elimination and back substitution.

Algorithm for Alternating Method of Solving Parabolic Partial Differential Equations

1. Initialize mesh
2. Compute matrix - $|D_X^i|$ N x M elements

$$\xi_1 \cdot \xi_2 - \xi_2 \cdot \xi_3 + \xi_1 \cdot \xi_4 + \begin{cases} \xi_5 \\ \text{Nothing} \end{cases}$$

3. Solve system of equations Tri-Diag 3N steps

4. Compute Matrix - $|D_y^i|$ M x N elements

ditto

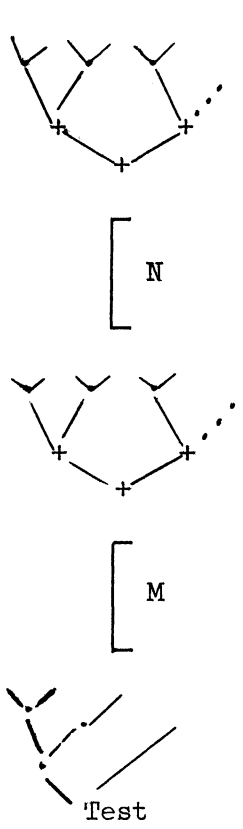
5. Solve system of equations Tri-Diag 3M steps

6. Test $\left|\frac{\eta_1 - \eta_2}{\eta_3}\right| < \epsilon$ M x N elements

7. Go to 2.

- -

Evaluating Degree of Parallelism



	Proc.	
	N x M	5NM
N	N x M	2.2NM
		5NM
M	N x M	2.2MN
	M x N	3MN

parallel 3(M+N+3) execution cycles per double length time step, sequential 21MN

if M = N parallel order N · the number of steps

sequential order N^2 · the number of steps

If the last entry in a row of the merge-link table is a zero, the successor address of the last instruction of that merge number is set to the address of the final instruction of the block. The final instruction of the block is a count instruction which causes execution of the next block when all merge sequences of the current block have executed. The use of a final merge instruction prevents subsequent blocks from beginning execution before the current block is finished.

A secondary naming of temporaries is performed in phase 3. Conventional methods of minimizing temporaries are applicable, but only within sets of instructions which have the same merge number.

2.2.4.1 Limitations and Potential Refinements

It is easy to verify that the object program is as efficient in execution time as the input language allows. The precedence scan of a statement determines the order in which operations are to be performed within a permutation of operators of equal precedence. The height operator determines the order of these operators to obtain the fastest computation. By making the height cumulative within a block, every scan within a block is assigned to execute at the earliest possible instruction time. Arbitrary simultaneity between blocks is impossible with an ALGOL-type input language since the sequence of blocks is, in general, a function of data and, as such, not specified in the input language.

By a limitation of the source language is meant a general rule with a finite number of exceptions has not been found to detect a useful property. For example, in the translation algorithm given here, all occurrences of a subscripted

variable were deemed to refer to the same variable. This method was used since arbitrary expressions can be used as subscripts and there is no general method of determining whether two expressions compute the same value without knowing the values for all the variables. Similar examples are the inability of deciding whether two blocks can run concurrently or not.

A restriction was made for the expression scan as presented. A function in an expression was assumed to leave the values of its arguments unchanged. Further, it was assumed a function would not use or would preserve within itself values of arguments needed more than one execution cycle after its entry. These assumptions allow the availability height of the arguments to be unchanged and occurrence height of the arguments to be one greater than the function call. For subroutine calls which are for the purpose of computing values for arguments, a separate statement using a single call must be used. Heights of arguments and quintuples are computed as though new values for arguments will be available upon return from the subroutine.

The algorithm is complete in the sense that special statements for input-output and tape or disc operations just generate subroutine calls. Storage allocation and/or isolation is essentially the same as on single processor computers. Techniques suitable for current translators should extend with minor modification to this multiprocessor translation scheme. Heuristics which sometimes but not always give advantages are purposely omitted from this text. These can be useful and should be considered in the light of a specific implementation of a translation algorithm. A minor detail is that redundant parenthesis are respected. This is by convention rather than a limitation since, in some numerical computation, the order of performing arithmetic can prevent overflow and

excessive roundoff error conditions.

Object programs from a translator as described here are far less efficient than a coding by a good programmer. This great difference in efficiency indicates a need for research on languages for multiprocessor computers. The problem areas include recognizing when an iteration can be reduced to a number of independent noniterative sequences. Also, the ability to quickly recognize block separations.

It seems reasonable to add additional statements to an ALGOL-type language so that the programmer could supply information about potential concurrency. The difficulty is to determine what information the programmer can supply easily and what information is needed by the translator.

2.2.5 Example of Application of the Translation Algorithm

To clarify and expand on the previous discussion, an example which uses most of the features of the algorithm is presented. In discussing the example, special cases will be mentioned only for the first occurrence. The example is meant to cover many cases in a relatively short space. Rather than give a prolific explanation of the example, tables are given as they would appear during translation.

Figures 31 and 32 show five statements as they would be given to the translator. For each statement the quintuples and part of the symbol table are given. These tables are as they appear at the end of processing the statement. The column entitled ROW in the quintuple table is for the reader only and exists implicitly during computer translation. The operands are denoted symbolically by variable names, constants, or row references. Row references are of the form,

R number. During computer translation operands in the quintuple table would be of the form S number or C number where the numbers would refer to the position in the symbol or constant table. The dotted lines in the columns for heights (START, END, OCCUR, AVAIL) separate the merge number from the height in that merge number. The merge number which is to be left of the dotted line is left blank if it is zero. For quintuples the starting height, START, indicates the machine execution cycle during which computation of an intermediate result begins. The ending height, END, indicates the first machine execution cycle during which the intermediate result may be used. For variables in the symbol table the availability height, AVAIL, indicates the first machine execution cycle during which the symbol may be used. The occurrence height, OCCUR, indicates the highest numbered machine execution cycle in which a symbol has been used to date.

Figure 33 is discussed in conjunction with Fig. 32 because it shows that status of the LIST before each quintuple is generated. The two additional symbols ⊢ and ⊣ appearing in Fig. 33 denote left and right termination operators and are inserted by the translator before processing of an expression. The set of precedences assigned to operators and punctuation for this example are:

Operator	Precedence
⊢ ⊣	0
, ( )	1
=	2
+ -	3
* /	4

Figure 40 is the flow diagram for the precedence scan used in this example.

The first line on Fig. 33 shows operands and operators added to the LIST

by the normal rules of the precedence scan. The "(" is recognized as having lower precedence than the leftmost element "+", on the LIST. There is no choice of ordering operands in the general height scan which is applied to "B + C". The quintuple is generated with designation R1 implying the first quintuple is a block. The starting height of R1 is zero since the AVAIL height of both B and C are zero. The OCCUR height of C is set to 1 as per the height scan rules. The END height of R1 is 2 because the operation requires one execution cycle and neither operand is a row reference thus requiring another execution cycle to load a temporary.

The height selection is illustrated by the third LIST of Fig. 33. The partial result "B +R2 +D" is to be converted to quintuples. "B" and "D" have lower heights than R2, thus the R3 quintuple "B + D Q2" is generated. The row reference, R3, replaces the "B" on the LIST and "+D" is deleated from the LIST. The partial result "R3 + R2" is denoted by R4. The START height is the maximum of the END heights of R2 and R3, i.e. 2, and the END height is one greater.

The last LIST in the first group of Fig. 33 indicates a substitution quintuple is to be generated. Since the chain of row references R7, R6, R5 ends with R5 which has a greater starting height than the occurrence height of A, the value for A will be computed by R7. The START, END height of R8 are thus the same as the END height of R7. The OCCUR and AVAIL height of A are also set to 4.

The second LIST in the second group of Fig. 33 indicates a quintuple is to be generated involving D and E. Since the operator to the left of D is -, the operator in the quintuple is the dual of the operator between D and E on the

LIST. This is necessary to compute the correct result in the form -(D + E) in place of -D-E.

A constant appears in the third statement as a reminder that constants always have zero AVAIL and OCCUR heights.

The fourth statement, Fig. 1b, involves subscription and function calls. Note that the first subscript quintuple, R18, requires one execution cycle. R19 also requires one execution cycle and begins at the same time but a fictitious END height is needed since R19 will appear elsewhere as an operand and the value of A is not available until cycle 5.

R20, which is a function call quintuple, has a START height equal to the maximum of the heights of its arguments, 5. The END height is 1:0 indicating merge number 1 and height 0. The next quintuple, R21, has as first argument the location of the returned value (which later will be a temporary).

The quintuple R22 indicates a computation follows from results with different merge numbers. Since this will ultimately be an instruction one cycle will be required making the END height 1: 1. Note the START height of the next quintuple is where the 1: 1 is used.

In R32 the substitution operation requires one cycle, thus the END height is one greater than the START height.

R36 shows a multiple subscript to be computed by a function call. In this case, the temporary assigned to R36 as an operand will contain the resultant machine address of the subscripted variable.

3. MACHINE ORGANIZATION

3.1 A MULTI-LAYER ITERATIVE CIRCUIT COMPUTER

3.1.1 Introduction

A study of the organization of the latest large scale computers shows a trend to an ever increasing complexity from the system design point of view. This evolution towards complex systems has been dictated by the desire to increase the power of the computers, sometimes in the productivity aspect, and in a few other cases in the computational capability test.

Most machine designs have had as a goal the maximization of the use factor of the computer, or at least of the most expensive units, generally the fast memory. This has been achieved by resorting to input-output buffering, by incorporating multiprogramming facilities reducible in the last analysis to time sharing procedures, by the inclusion of partial multiprocessing capabilities, and in some cases by creating a programmable structure organization as in the polymorphic machine.

It must, however, be recognized that most of the available commercial machines tend to maximize the productivity, that is, they try to minimize the cost per instruction, which is proportional to the ratio of speed to cost per operation.

On the other hand, very little has been achieved with respect to increasing the bounds of practical computability. Thus the problems encountered in the fields of pattern recognition, games, simulation and adaptation still need a computer capable of handling them in an efficient manner.

The iterative circuit computer has been considered as the most suitable solution for these types of problems which have in common the characteristic

that the spatial distribution of the modules is an homomorphic image of the relations governing the interaction of the variables. The undisputed suitability of this class of computers for these problems has relegated to a second place some other properties of the iterative structure that are not exclusive of this organization, but which are much more easily implemented in it than in a system with specialized units. The iterative circuit computer provides the possibility of true simultaneous multiprogramming, plus the powerful resource of infinite interaction between the programs. Neither of these characteristics is present in any of the more sophisticated systems now available.

Furthermore, an individual module can function at various times as an accumulator, register, memory cell or simply as a connecting link, and can be activated in any of these functions at any time during the execution of the program. Therefore, we are in the presence of an organization even more flexible than that of a polymorphic machine.[12] Although it would seem inappropriate to apply this term when there exists a lack of specialized units, it must be remembered that the modules are structurally alike, but their instantaneous functional behavior is different and is defined by the current instruction.

The polymorphic system is a programmable structure machine, but the changes in structure are performed on the interconnections between modules and not on the internal organization of the computer modules. Therefore, the full advantage of a changing structure is not realized, although a great increase in the use factor of the system is obtained. There are two

factors affecting the effectiveness of the system.

The first is due to the specialization and non-convertibility of the units, that is, there is a fixed number of components of each type available resulting in a limited number of combinations that can cover only a limited class of problems. Many problems cannot be handled efficiently because of the lack of more units of a certain type, while at the same time there may be a certain number of idle units that cannot be put into service because they perform different functions.

The second factor is closely related to the first, and is connected with the problem of priority assignment. It has been shown[13] that an attempt to obtain a high utilization factor for the computing modules increases the mean queue length. If there exists a number of programs with low priorities, then a high use factor can be obtained, but usually a compromise must be reached between efficient utilization of equipment and length of waiting lines.

In an I.C.C., however, any number of modules can be performing any one of the possible functions in one step of the program, and entirely different ones in the following operations. Also, the polymorphism of the machine is a function of the current instruction, not of the maximum requirements of the program.

The available literature on I.C.C.'s is surprisingly scant. References 1,2,3 cover the design considerations, both for uni-dimensional and two-dimensional networks. Reference 3 includes also the treatment of the problems of stability and equivalence of iterative networks. References 4,5,6 cover the specific problem of embedding a computer in the logical iterative net-

work. These are practically the only proposals for a computer based on this type of networks. Unfortunately, reference 4 covers only a special-purpose computer intended for pattern recognition and allied spatial problems. The paper by Holland[5] has been the starting point for a number of projects, but its title has mislead many into believing this was a proposal for a practical machine. While most of the ideas are worthwhile, they are by no means unique or optimum, as S. Amarel has clearly pointed out in his review.[8] Holland only describes a mathematical model of a space in which a simulation of the physical laws governing the interaction of a system with the environment can be set up. As such, the model possesses all the uniform properties and generality necessary for its use as a simulator in which the process of adaptation can be studied. While it still retains the power of an ordinary computer, its use as such would imply a wasteful employment of its potential capabilities while its performance would be hindered by an excess of non-essential features for this particular role.

Especially criticizable are the following features which affect characteristics that are fundamental in any I.C.C.:

The scheme used for selecting operands suffers as a consequence of both the method used for addressing and the path-building procedure necessary to reach them. The addressing method employs a "floating" reference, that is, all the addresses are relative to the address of the module active at that moment, and therefore an operand address assumes a different representation in every instruction that refers to it.

The path-building procedure has the disadvantage of being essentially sequential, resulting in a long effective access time, and therefore assigning great importance to the problem of data allocation.

These difficulties can very well be attributed to the lack of organs of command and to the circumstance that both control and information channels flow through the same network.

In Newell's paper,[7] however, we find the first reference to a multi-layer iterative structure, and furthermore, he suggests solutions to the problems of grouping modules to function as single entities and for the simultaneous selection of operands. It seems therefore logical to try to specify the organization of a multi-layer machine having each of the layers fulfilling some specialized function, yet being in itself a complete iterative structure.

The purpose of this paper is to present one possible example of such an organization, in which the following new characteristics are incorporated:

(a) A path-building procedure having the short-time access advantage of the common-bus system, but which also allows simultaneous multiple path building with no mutual interference.

(b) Three-phase operation, with specialized networks operating simultaneously in different phases on three consecutive instructions.

(c) A specialization in the functions performed by the stacked networks.

(d) Inclusion of geometrical operations in addition to the arithmetic and logical ones.

3.1.2 Description of the Computer

The computer is composed of three stacked layers, each consisting of an iterative network of m x n modules. The three layers are exactly alike in size, shape, and type of modules used.

One layer is called the "program plane." This contains at the start the original program or programs, and later the modified programs resulting from the interaction of the original ones. Fig. 41.

The intermediate layer is called the "control plane" and its function is to interpret the instruction following the one being executed at the moment, determining the operand(s) and storing in them the full instruction. These "image operand(s)" in the control plane will in turn generate activation signals which will be transmitted on the wires connecting correlative modules and will determine which modules in the third plane will act as operand(s) II in the next phase.

The third layer is the "computing plane" where the actual arithmetic, logical and geometrical operations are performed.

The number and distribution of the modules active at any time is determined by the signals transmitted from the "control plane" in response to an "operation complete" pulse from the computing plane.

Communication between the three layers is provided by the following sets of connections. Fig. 41:

(a) A set of connections from every module in the program plane to every correlative module in the control plane.

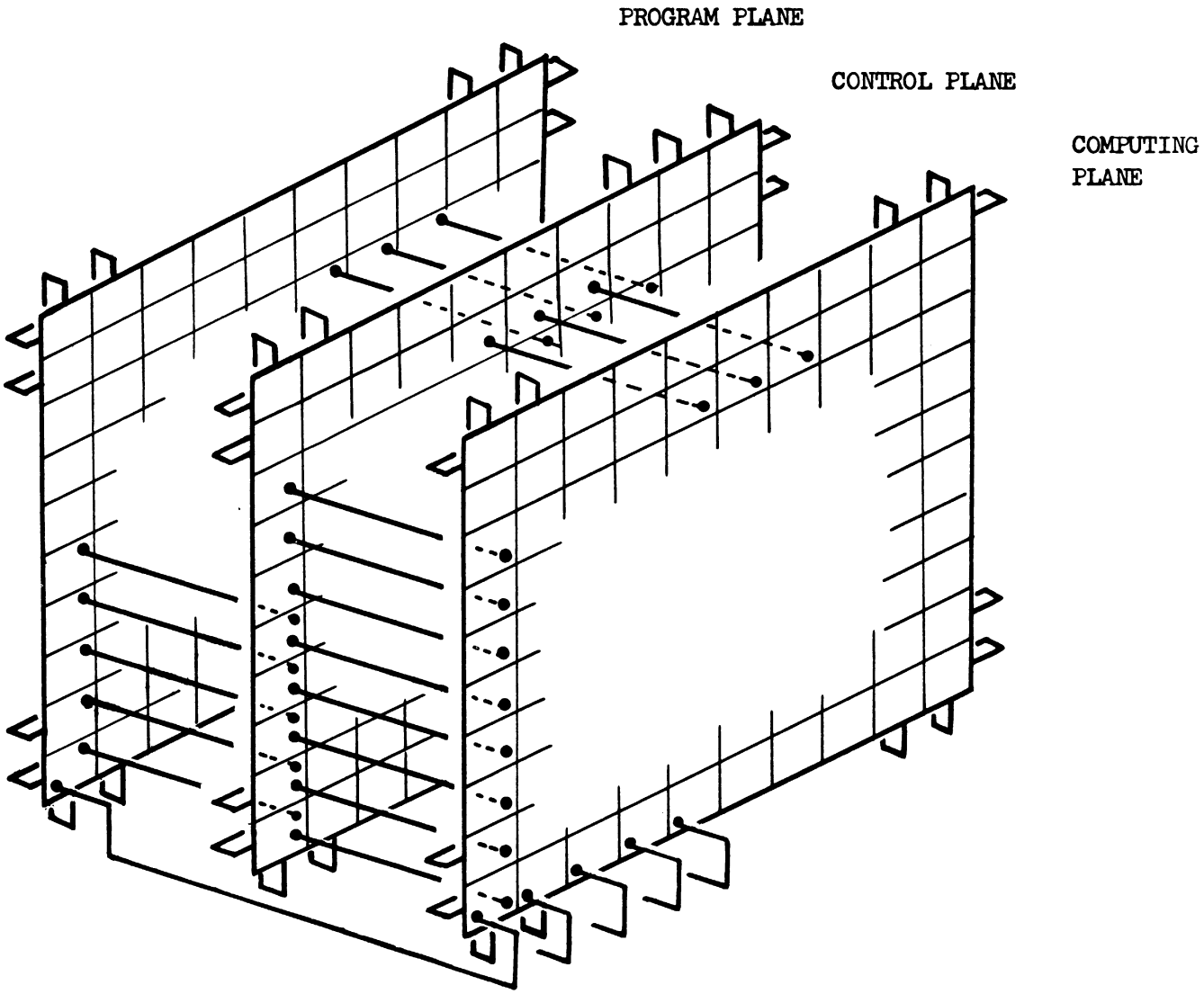


Fig. 41. Inter-layer and wrap-around connections.

(b) A similar network of connections from the modules in the control plane to those in the computing plane.

(c) A similar set of connections from the modules in the program plane to those in the computing plane.

(d) A common bus line connecting all the modules in the computing plane, and transmitting the "operation complete" pulse to all the modules in both the program and control planes. Fig. 42.

The connection lines described in (a), (b), and (c) are called activation lines.

3.1.3 Description of the Planes

Each plane consists of a network of $\underline{m}$ x $\underline{n}$ modules, the modules being connected by a line called the information line running in each row and column through the normally conducting gates in each module. Fig. 44.

Besides these inter-modular connections, there exists an end-around connection between the first and last module of each row and similarly for the terminal modules of the columns. Therefore, there is a separate information line for each column and row, which is closed on itself by the end-around connection. These end-around connections provide the spatial continuity of the structure, transforming the planar distribution into one where a uniform neighborhood relation holds for all the modules, with no constraints due to physical boundaries. The resulting continuity provides the same behavior as that of a network spread over the surface of a torus.

3.1.4 Description of the Modules

All the modules in the three planes are exactly alike. They communicate with each other in the same plane by means of the column and information lines running through them, and with the correlative modules in the other layers by means of connections called activation lines.

The internal structure of the modules includes an accumulator, a register, a decoder and several switching matrices to connect the former to the information lines. These units can be described as follows:

(a) An accumulator capable of performing addition, whose input is supplied from the output of a switching matrix connected to the four possible inputs to the module. The accumulator is connected through parallel gates with a register of the same length, which is described in (b).

(b) A register of the same length as the accumulator, and which is connected with it through parallel gates. This register simply copies the contents of the accumulator every time a load-type operation is completed. At the same time, it supplies the only output lead over which the contents of the accumulator can be read out. This means that every time some instruction requires the transmission of the contents of the accumulator, the information is actually taken from the corresponding register. The output can be directed to any of the module's four terminals by the switching matrix (d).

(c) A switching matrix with inputs from the module's four input terminals, and whose output is connected to the input of the accumulator. The

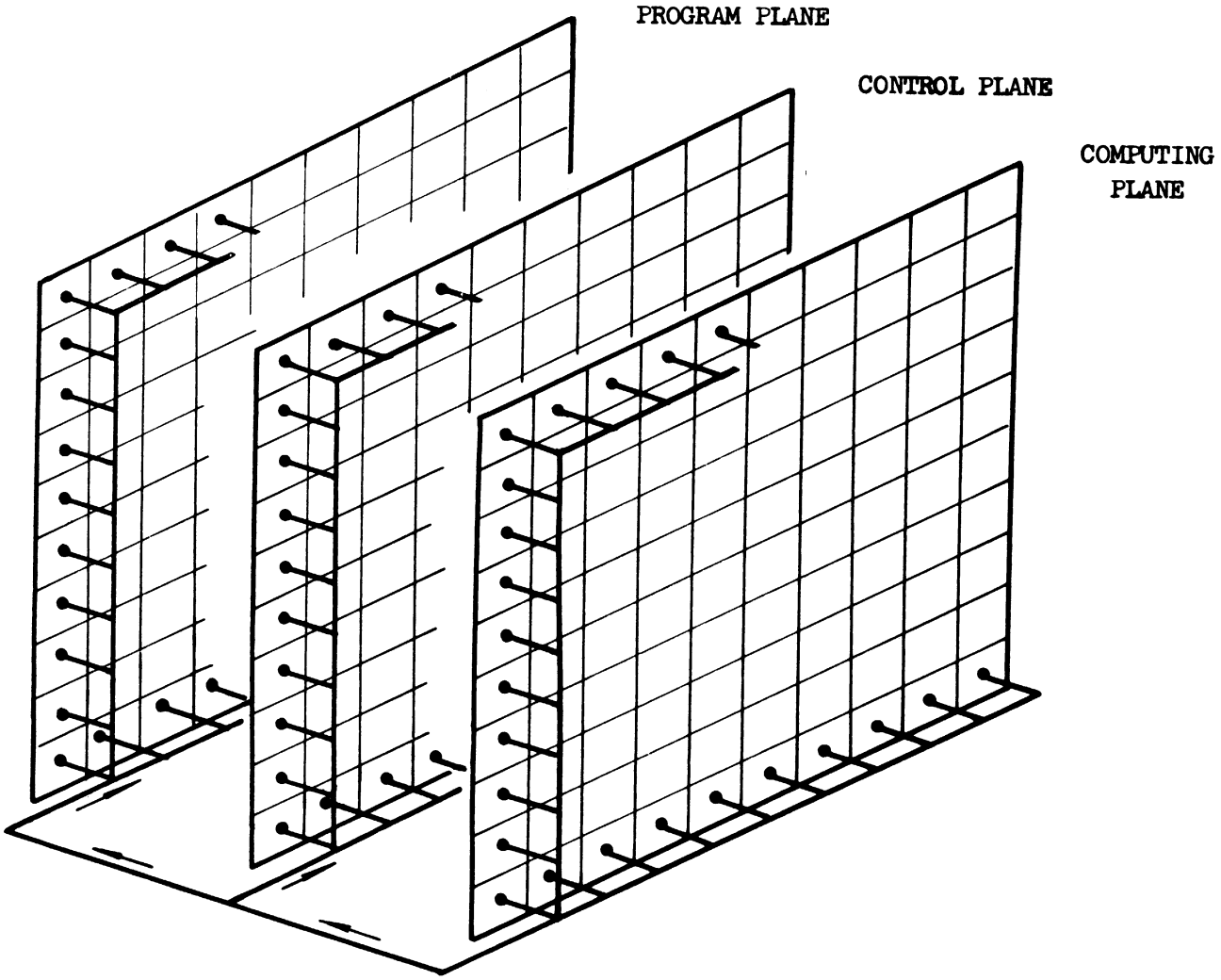


Fig. 42. Three-plane structure and common buses.

setting signals for the matrix are supplied by a decoder, described in (g).

(d) A similar switching matrix, whose input is the output of the register, and whose output feeds any of the four terminals.

(e) A pair of gates, normally conducting, that connect the terminals belonging to opposite sides of the module, thereby maintaining the continuity of the vertical and horizontal information lines across the module, with no connection between them.

(f) A switching matrix that can connect any input terminal to its immediate neighbor. In this way, a corner in the transmission path can be formed.

(g) A decoder, which receives the complete instruction on the normally continuous information line, and which compares the address in the instruction with its own address, producing output signals that govern the setting of (c), (d), (e), and (f).

(h) A similar decoder for the other information line.

(i) A gate connecting the output of the register with the activation line going to the input of the correlative module in the computing plane.

3.1.5 Word Format

A word is composed of four fields: the operand I field, the code field, the operand II field, and the successor field.

The operand I field contains two pairs of symbols; the first pair indicates the rows to which the first and last modules in the pattern or group of modules

belong; that is, it gives an indication of the extension of the pattern of operands I. The second pair does the same with respect to the initial and final column coordinates.

Example:

Operand I field	Code field	Operand II field	Successor field
(36;22)	Load	(22;33)	(33;77)

Since we are dealing with linear patterns, that is modules that are all in one column or row, at least one of the pairs in the operand I or operand II fields must contain a repeated number to indicate that only one column or row is involved. Example: (36;22) indicates the pattern extending from row 3 through row 6 and belonging to column 2. Therefore, the operand I field indicates which modules will be operands I when the instruction is executed. Similarly, the operand II field gives the address of the group of modules which will become operands II.

The successor field specifies the address of the location of the next instruction, and therefore is always of the form (XX;YY) since it must necessarily refer to a single module.

3.1.6 Path-Building Procedure

The method used for communicating between modules in an iterative circuit computer is one of the key factors that determine the efficiency of the machine. The method described here is not strictly a path-building procedure since the

connections are permanently established as row and column information lines. The procedure only determines the operator and operand locations, and sectionalizes the corresponding row and column information lines into segments that are connected together at the cross-over point.

The general structure of the switching arrangement within each module is shown in Fig. 43. The gates are shown as bi-directional to simplify the diagrams. The row and column information lines run through all the modules in the corresponding row or column forming a closed loop since all the switches in the path are normally closed. Fig. 44.

The sequence of operations is as follows: The current instruction is stored in the active module, in this case the module at the upper left corner of Fig. 45. The instruction word is transmitted over one of the two information lines, the choice depending on the shape of the pattern of operand I. If the operands I are all in one column, then the information is transmitted on the row information line and vice-versa. Since we deal with linear patterns, one of the pairs of coordinates in the operand I field will always be of the form XX; the repeated number indicating that the pattern extends linearly over the X column or row. The instruction transmitted on the information line that spans the whole row or column where the operand I is located is received by the decoders in all the modules in that row or column. Each decoder checks for coincidences between the operand I and II addresses contained in the instruction and the corresponding addresses of its own module. This includes the module originating the information.

In Fig. 45, the operand I has the address (33;22) and the operand (55;66). When the decoders in the modules of the first column compare these addresses

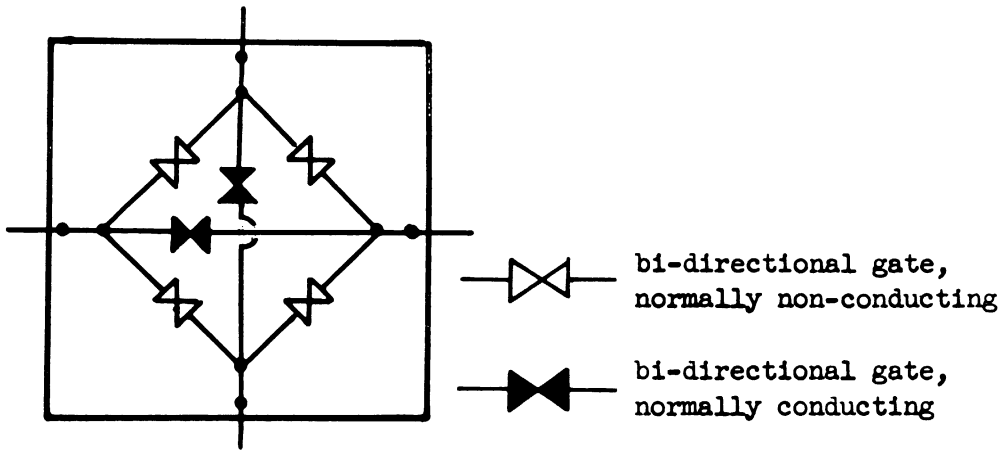


Fig. 43. Information-line switching in a module.

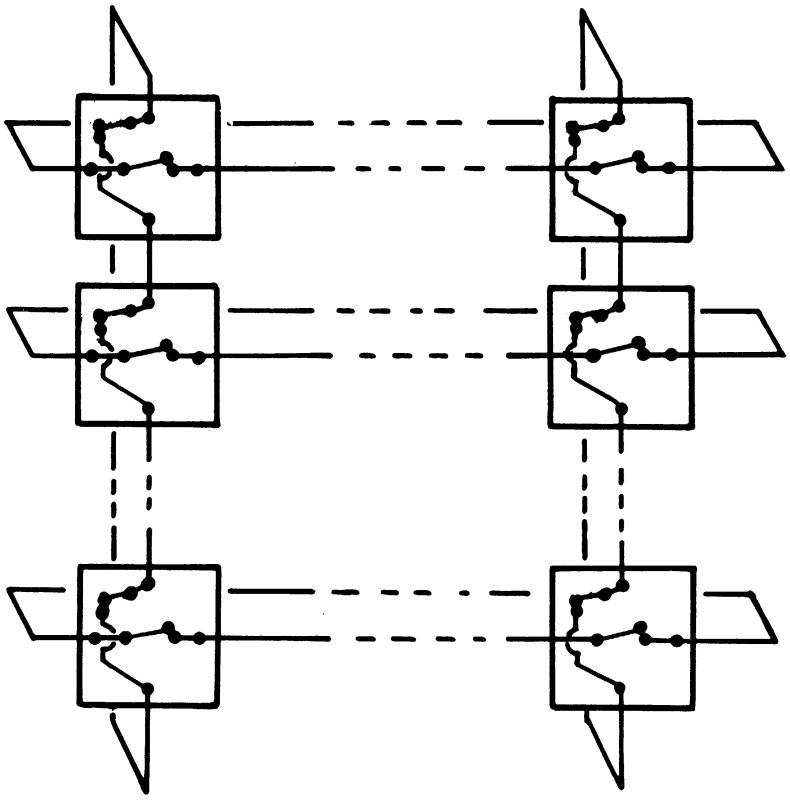


Fig. 44. Column and row information lines.

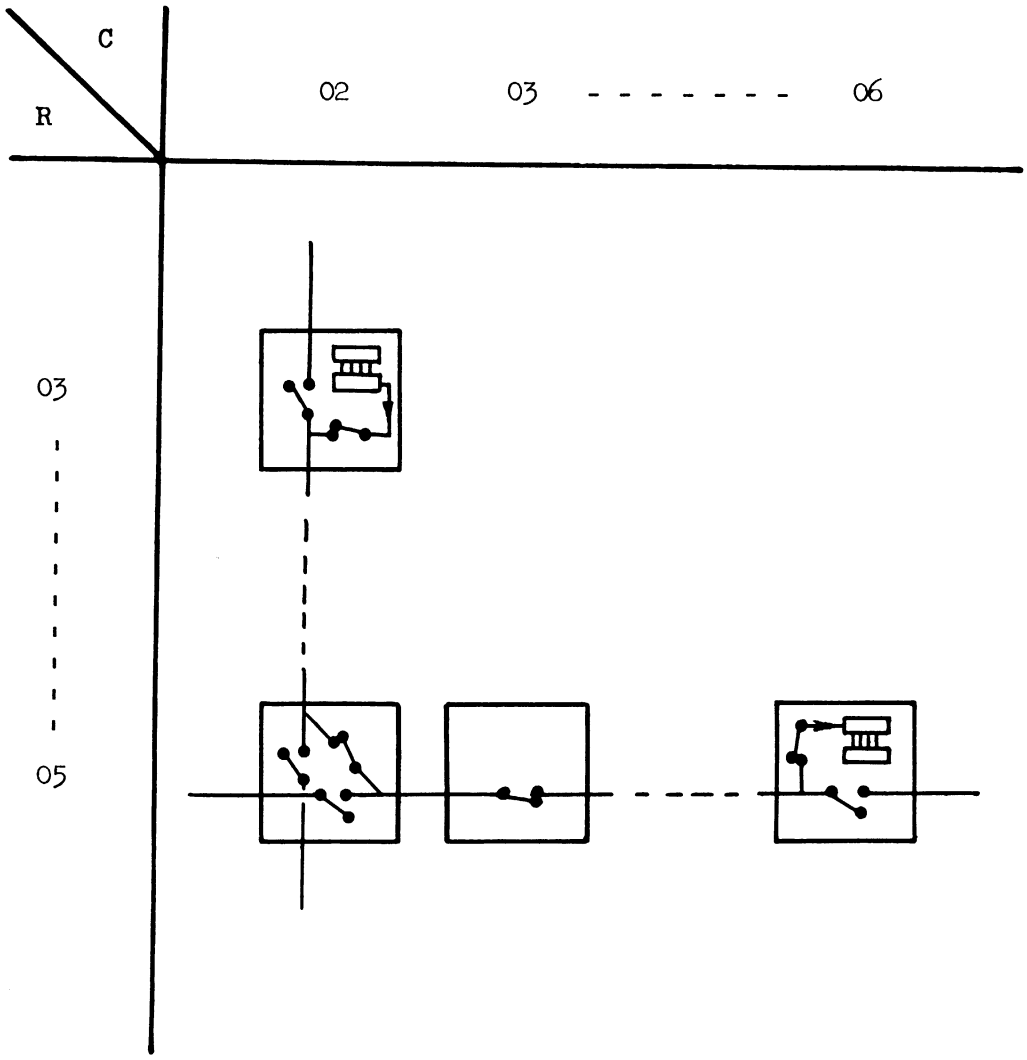


Fig. 45. Path connection for the instruction: (33;22)(store)(55;66).

with their own, two types of coincidence may arise:

(a) Double coincidence between the row and column coordinates belonging to one of the operand fields in the instruction, and the corresponding ones in the module address.

(b) Double coincidence between one row address in one field, one column in the other field and the corresponding ones in the module address.

It is evident that case (a) occurs only when checking the addresses of either the operand I or the operand II. In the first instance, the addresses in the operand I field will coincide with the addresses in the corresponding field in the module address. When the operand II is checked, the operand II field addresses will coincide.

The second case will occur at the module situated in the intersection of the column and row to which the operand I and operand II belong. In Fig. 45, the following situation will arise:

Instruction: (33;22) Store (55;66)

Intersection address: (55;22)

The double coincidence is between row and column addresses belonging to different fields in the instruction word.

The different results of the coincidence checking procedure are used to trigger two different sequences of events:

(A) If case (a) occurs, then either an operand I or II location has been reached, and the decoder activates one unit (e) and either (c) or (d), as described in Section 3.1.4. As a consequence, the following operations take place (i) The switch in the information line is opened, isolating the rest of the line.

(ii) Either the input or the output of the accumulator is connected to the information line. The operand I is always the source of information and therefore the transmission is from the operand I location to that of operand II.

(B) If case (b) occurs, then a corner in the path has been reached, and the decoder activates both (e) units and the (f) unit. As a consequence, the following operations take place: (i) Both switches in the row and column information lines are opened, completing the isolation of a piece of line from the terminal module to the corner in each information line. (ii) One of the switches connecting adjacent sides is closed, connecting the two isolated pieces of line and forming a continuous path from operand I to operand II.

The above procedure takes only two pulse times because all the decoding takes place simultaneously in all the modules in a row or column. Furthermore, it doesn't depend on the relative position of the modules to be connected, but only on the addresses of the operands.

In the case of instructions with multiple operands I and/or multiple operands II, it is possible to connect them in a one-to-one, one-to-many, or many-to-many way.

3.1.7 List of Instructions

It is very common to speak of the operand I and the operand II when referring to an instruction, and usually no further distinctions are needed because there is only one accumulator. However in the case of the iterative circuit computer, where both operands are stored in modules that have the same capabilities, the distinction is no longer adequate. It then becomes

necessary to specify the direction in which information must flow since both modules can process the instruction and store the result.

Actually, any of the two modules could perform the role of accumulator and then we would have a left and a right instruction of each type, depending on which module executes the instruction and stores the result. In order to simplify the list of instructions, it is arbitrarily agreed that the operand II will always be the accumulator. Following this convention, the list of instructions for elementary arithmetic and logical operations is reduced to the following:

LOAD: Loads the contents of the location specified by the operand I into the location specified by operand II. The result appears in the location of operand II.

ADD: Adds the contents of operand I to the contents of operand II. The result remains in operand II.

COMPLEMENT: The contents of operand I are complemented.

TRANSFER ON NON-ZERO: If the contents of operand I are different from zero, control is transferred to the instruction located in the module specified by operand II. If the contents are equal to zero, the normal sequence of instructions is followed, that is, the next instruction executed will be the one specified by the successor field.

3.1.8 Operation of the Computer

The execution of an instruction takes place in three phases, and each phase is performed in a different layer. Once a plane has executed its phase on an instruction, it waits until the next operation complete pulse from the computing plane causes the transfer of a new instruction to be operated on. Thus, the execution of the three phases proceeds simultaneously in the three layers, but with a different instruction in each layer. As a result, the effective operation time is one instruction per phase; the duration of the phases being determined by the computing phase being executed at the moment.

Fig. 46 shows the sequence of phases and the transfer of each instruction from plane to plane after the execution of each phase. The roman numerals indicate the phase, and the subscript the particular instruction being operated on. Thus, III_2 indicates that the second instruction is undergoing phase III. Phase III is the one that takes the longest time to perform and therefore is the one that generates the operation complete pulse that triggers the initiation of all phases in the three layers.

The sequence of operations that an instruction undergoes during the three phases is as follows:

PHASE I: The initiation of this phase is triggered either by an operation complete pulse or a start pulse. During this phase, the instruction following the one already in the control plane is made ready to be copied from the program plane onto the control plane in the same relative position. The net effect is to choose the successor to the current instruction already in phase

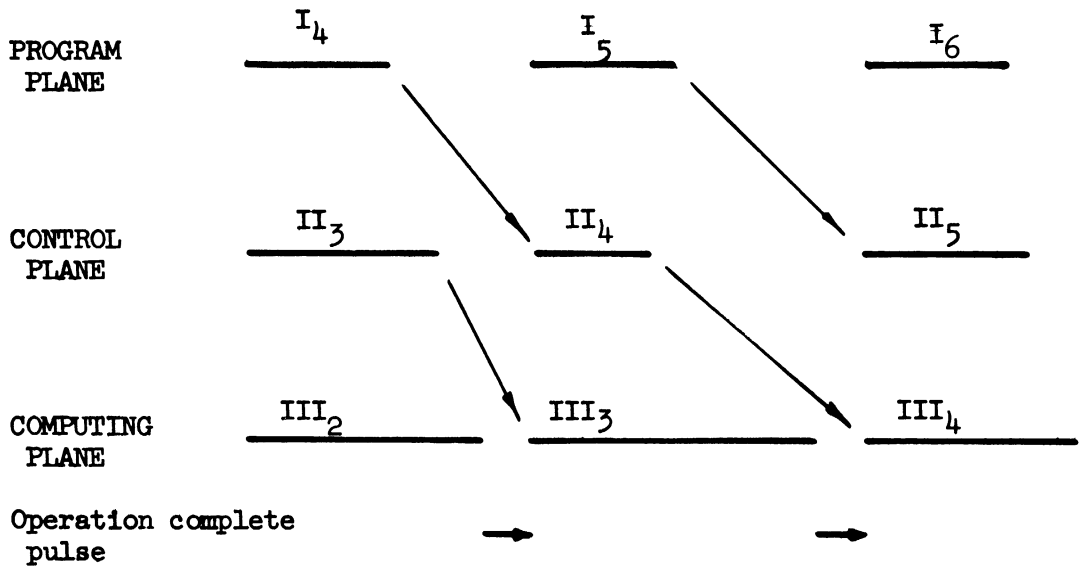


Fig. 46. Overlapping of phases.

II. The successor is specified by the address in the successor field and can be any location in the plane. In other words, it is not required that it be a contiguous neighbor.

PHASE II: The instruction activated in the program plane is copied in the same position in the control plane. In this plane, the operand II field of the instruction is interpreted to determine which modules are to be active (operands II) during the computing phase. Once the positions of the future operands II are determined, the whole instruction is transmitted and copied in these "image" positions in the control plane. The location of the "image" operators is found following a path-building procedure, as described in Section 3.1.6. Furthermore, the necessary data are copied from the correlative modules in the program plane. In this way, each future operand II position is now loaded with the instruction and the operand II itself.

PHASE III: The next operation complete pulse from the computing plane bus line initiates the third phase. The instruction and data now stored in the module or modules in the control plane are now transferred to the correlative modules in the computing plane, and each of these modules initiates a path-building procedure to connect itself to its operand or operands I.

At the end of this process, the modules containing the instruction are connected to their respective operands I. This connection can be from one module to another, from one to many, or from many to many, depending on the operation specified by the current instruction contained in the operands II. The locations thus selected receive the necessary data from the correlative positions in the program plane.

Once the connections have been established, the instruction is executed with information flowing in the correct direction. Operand I is always the source of information and therefore the output of its register has been connected to the information line. Similarly, operand II acts as the accumulator and the information line is connected to the input of its accumulator. The result is then transmitted to the correlative module in the program plane, where it is stored. Simultaneously with this activity in the computing plane, the control plane is now executing phase II on the next instruction, since it remains free once the "image" operators have been transferred to the computing plane. The completion of the execution phase is signalled by an "operation complete" pulse which is transmitted over the common bus from the computing plane to the similar buses in the program and control planes. This pulse initiates phase II in the control plane and phase I in the program plane.

This sequence of operations can be visualized following the transfer of instructions between the layers, in Figs. 47 through 50, while the computer executes the following sequence of instructions, supposedly part of a program:

1. (77;88) Load (44;55) (33;44)
2. (57;77) Add (57;33) (33;55)
3. (56;77) Load (66;23) (33;66)
4. (33;24) Load (66;22) (33;77)
5. (00;00) Clear(56;33) (22;77)
6. (55;66) T.Ø (33;99) (22;88) (Transfer on non-zero)
7. (00;00) No op(00;00) (22;99) (No operation)

Figure 47 shows the computer at the moment the first instruction is undergoing phase III. The state of the machine can be indicated by: III_1; II_2; I_3. The program plane is executing phase I on instruction 3, that is, it activates instruction 3 as the successor of instruction 2. In the control plane, both instruction 2 and the data corresponding to the operands II are being loaded into the locations asignated to the operands II. In the computing plane, instruction number 1 is being executed, with the contents of module (77;88) going into module (44;55).

Figure 48 shows the machine in the state III_4; II_3; I_2. The control plane is interpreting instruction 3, locating the positions of the image operands II, in this case the modules in row 6 and columns 2 and 3. Both instruction number 3 and the contents of modules (66;23) in the program plane are now copied into the correlative positions just determined in the control plane. The computing plane is executing instruction number 2, in this case, adding the contents of (57;77) into (57;33).

In a similar way, Figs. 49 and 50 show the execution phases of instruction numbers 3 and 4.

All instructions except the Transfer on Non-Zero instruction are treated in a similar way. The execution of instruction number 6, which is a Transfer on Non-Zero, gives an opportunity to explain in more detail the sequence of operations for this type of instruction.

When instruction number 4 is executed, an Operation Complete pulse is sent back to the program plane, and instruction number 6 is activated. It has to be remembered that instruction number 5 is already in the control plane.

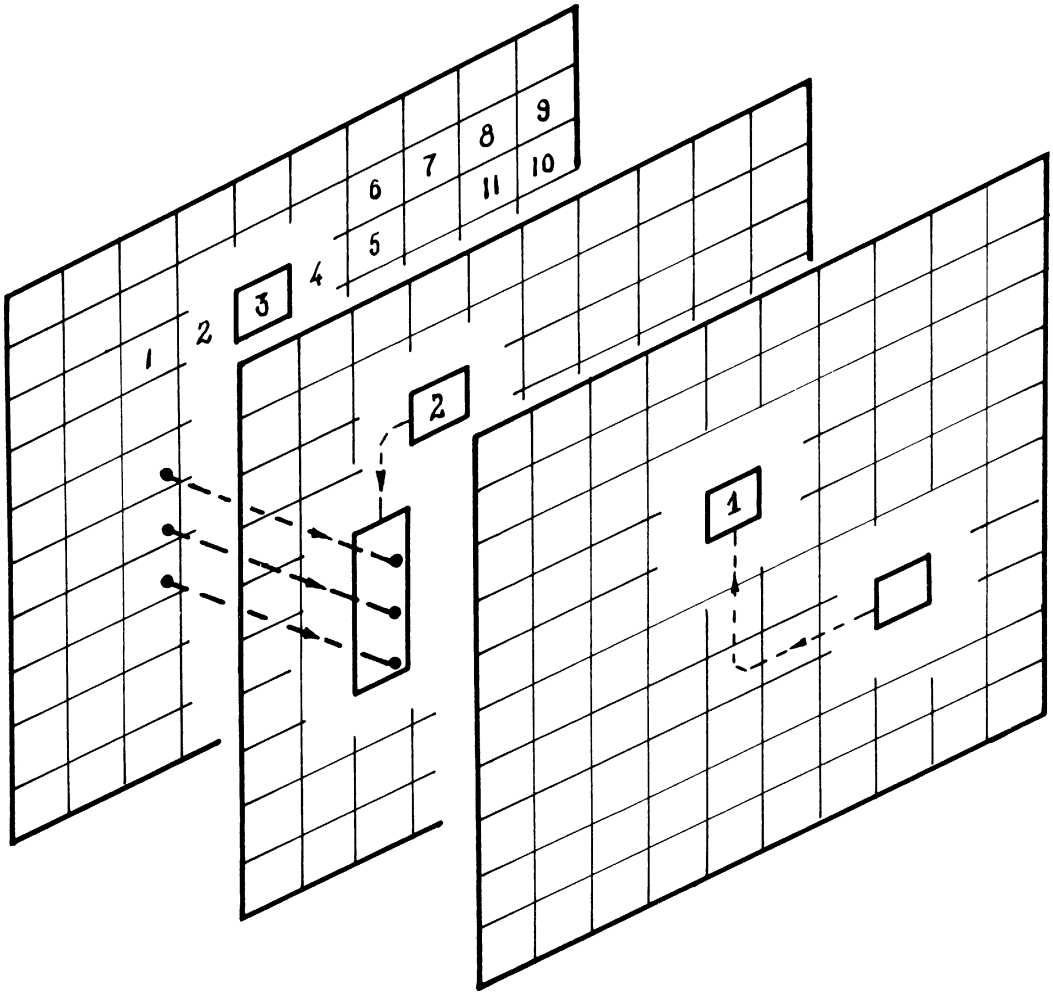


Fig. 47. Execution phase of instruction 1:
(77;88) Load (44;55) (33;44).

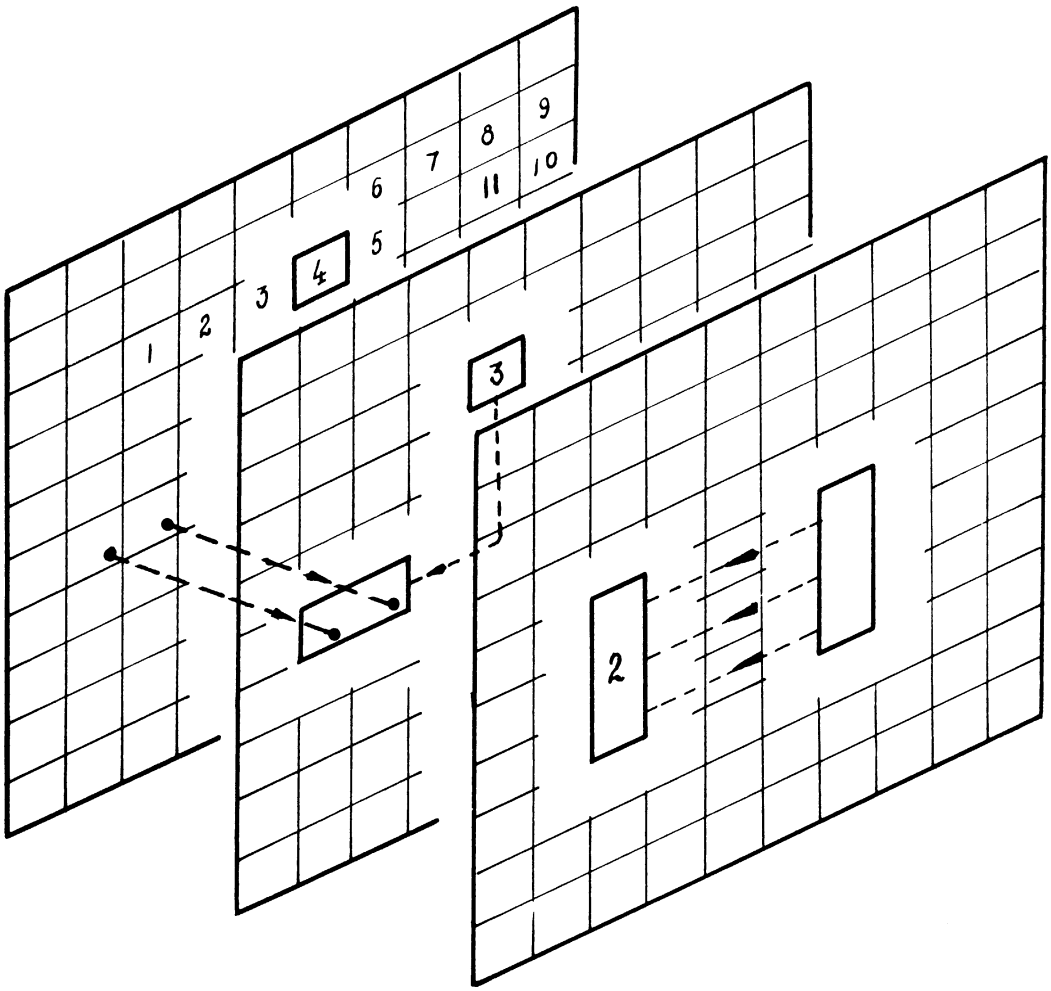


Fig. 48. Execution phase of instruction 2:
(57;77) Add (57;33) (33;55).

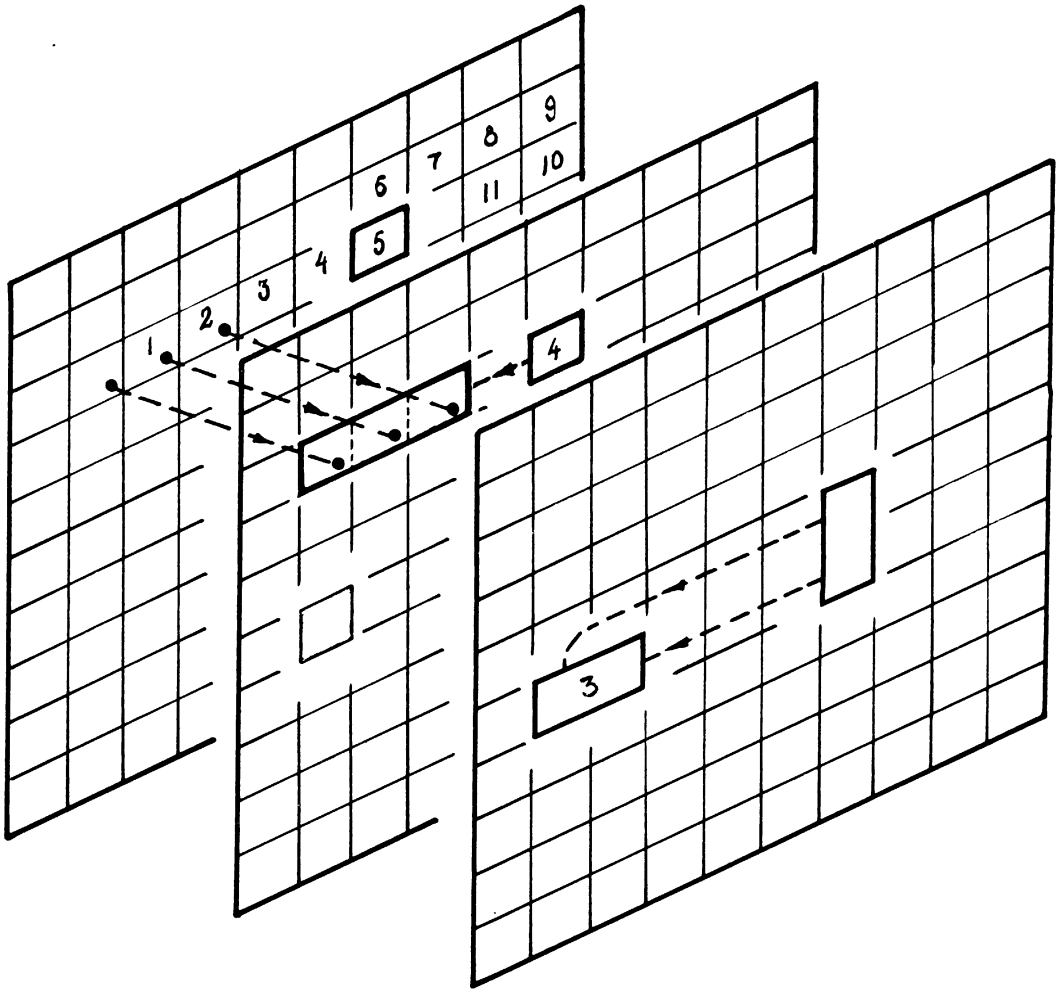


Fig. 49. Execution phase of instruction 3:
(56;77) Load (66;23) (33;66).

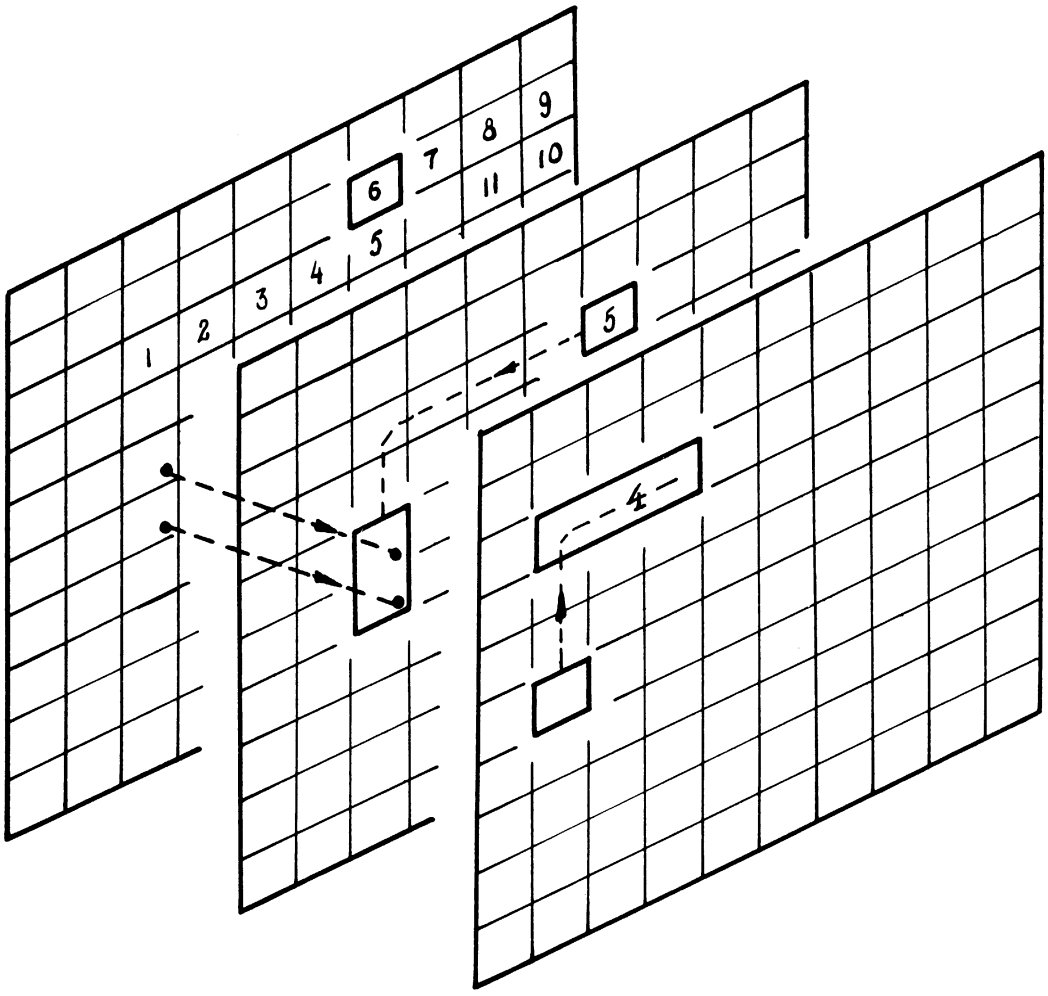


Fig. 50. Execution phase of instruction 4: (33;24) Load (66;22) (33;77).

The situation is that illustrated in Fig. 50, and again in Fig. 51, but this time in a lateral view.

The operation complete pulse changes the situation to that illustrated in Fig. 52. Instruction number 6 is transferred to the control plane, and a path is built there connecting the module containing the instruction with the operand II, in this case module (33;99). The instruction is then stored in this module which receives the alternate address from the correlative module in the program plane.

The next operation complete pulse, signalling the termination of instruction number 5, produces a copy of module (33;99) in the computing plane. Fig. 53. This module is then connected to the operand I module, in this case (55;66). The operand I module contains the word of data on which the result of the transfer instruction depends.

The active module (33;99) then determines if the number in (55;66) is equal to zero or not. If the number is equal to zero, an operation complete pulse is emitted and the normal sequence of operations is resumed. If the number is not equal to zero, the active module (33;99) sends an activation signal to the correlative module in the program plane, activating it as the immediate successor and overriding the active status already obtained by instruction number 8. After a suitable delay, an operation complete pulse is emitted, and the normal sequence of operations is resumed. The delay is necessary in order to allow the newly designated successor to activate its own successor, which may be any position in the plane, not necessarily a contiguous neighbor. Fig. 54.

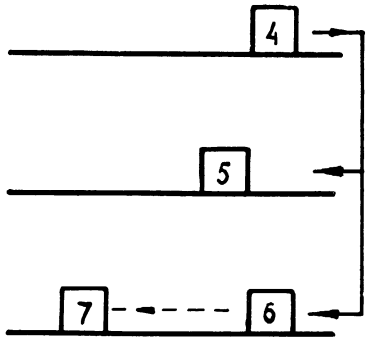


Fig. 51. Operation complete pulse.

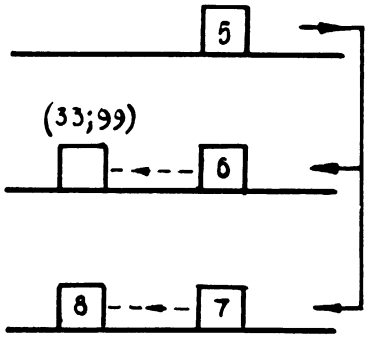


Fig. 52. Transfer of instruction 6.

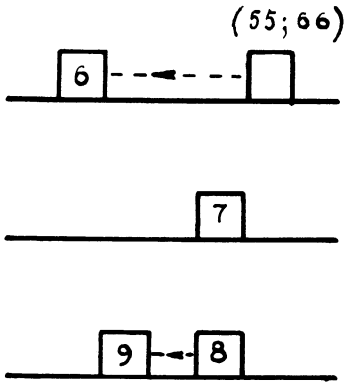


Fig. 53. Operation 6 executed.

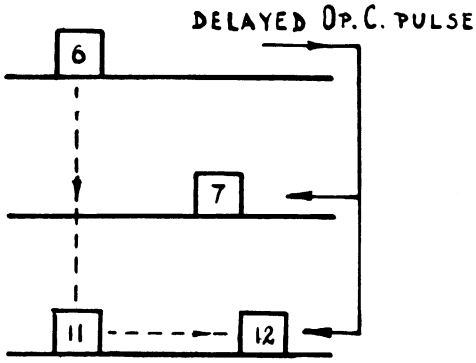


Fig. 54. Delayed operation complete pulse.

Therefore, the sequence of instructions resulting from the transfer instruction is: 6, 7, 11, 12 ... instead of the normal sequence: 6, 7, 8, 9 ...

3.1.9 Geometrical Operations

When an attempt is made to process geometrical patterns in a computer in which the instructions refer to only two operands, it is necessary to divide the pattern into individual elements and operate on them one at a time. In the machine described here, the availability of multiple operand instructions reduces most of the geometrical operations to one of the arithmetic or logical ones.

In order to simplify the operation code, it is convenient to establish the relationship between the geometrical and arithmetic operations since most of the former can be interpreted as a particular case of multiple operand I and/or multiple operand II arithmetic operations. Thus, a Store operation can refer to a One-to-One (OTO), One-to-Many (OTM) or to a Many-to-Many (MTM) operation.

The OTO Store operation is the normal one, and in the geometrical interpretation would be called a COPY instruction.

The OTM Store instruction has two versions in the geometrical case: (i) The pattern of one module is to be repeated contiguously and linearly. Fig. 55. The corresponding geometrical operation is called EXTEND. (ii) The module has to be copied in repeated positions, all consecutive, but not contiguous to the original one. Fig. 56. The corresponding geometrical operation is called REPRODUCE.

The MTM Store operation repeats the pattern in a position parallel to the original one. Fig. 57.

The corresponding geometrical operation is called DISPLACE and reproduces the pattern in a parallel position. It implies a simultaneous one-to-one copy operation on many modules.

Therefore, a correspondence between the geometrical and arithmetic operations can be established, in which the first column can roughly be assimilated to a compiler language and the second one to a machine language.

COPY - - - - - - - - - Store OTO

EXTEND - - - - - - - - - Store OTM

REPRODUCE - - - - - - - - - Store OTM

DISPLACE - - - - - - - - - Store MTM

3.1.10 Conclusion

The organization presented here is not intended to be an ultimate design. Rather it presents one possible way of combining the intrinsic capabilities of the iterative structure with the advantages of an organization having some form of specialized control unit.

The principal advantage of the proposed organization resides in the fact that the multi-layer structure makes it possible to include a control plane which acts as a look-ahead unit, interpreting the instructions before the actual execution takes place.

This disposition provides the capability of dealing with instructions

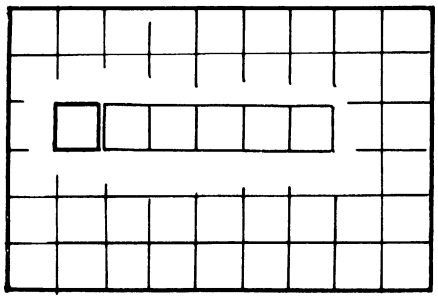


Fig. 55. EXTEND operation.

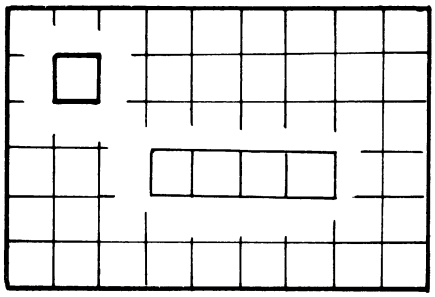


Fig. 56. REPRODUCE operation.

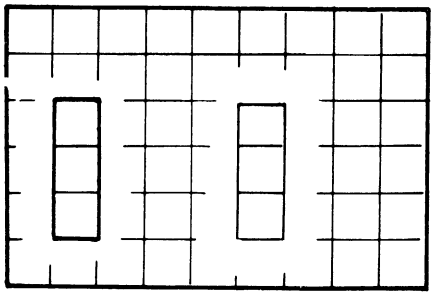


Fig. 57. DISPLACE operation.

that operate on any number of modules simultaneously, yet retaining in every step the possibility of true simultaneous operation of several programs with an unlimited degree of interaction between them. Moreover, the introduction of the look-ahead feature doesn't detract from the effective speed of computation, since the delay introduced by the pre-interpretation phase is compensated for by the overlapping of the sequence of phases which process consecutive instruction in different places simultaneously.

Furthermore, the method used for path building provides communication between the modules in the network with a very short access time.

Therefore, the combination of features given by the pre-interpretation of instructions and by the overlapping of phases can be regarded as a net advantage, with no penalty in time or complexity of the individual modules. The sole and inevitable penalty is the inclusion of two more layers. While this increases the number of modules by a factor of three, it must be remembered that the whole feasibility of this type of machine organization depends on the availability of components whose cost depends very weakly on the internal structure, and whose easy reproducibility assures a low cost per unit when used in large numbers.

3.2 PHYSICAL AND LOGICAL DESIGN OF A HIGHLY PARALLEL COMPUTER

3.2.1 Introduction

There are a number of features each programmer would like to have in a large-scale digital computer. The desires are as numerous as the programmers and often serve opposite purposes. Thus, one of the foremost problems in developing an improved computer organization is to recognize the fundamental requirements of the programmer. It can be safely said that, in general, programmers desire computers to be large, fast, versatile, and easy to program.

We are presenting a machine organization for a general-purpose computer that could be superior to existing computers for some problems and would extend the range of problems solvable on computers.

3.2.2 Objectives

Our primary objective will be to increase the amount of computation that can be done in a given time by means of a new organization rather than faster components. To this end, our organization provides for simultaneous execution of many instructions, each by a separate processing unit. The machine's ability to do parallel processing leads to the desire for providing unlimited interaction among processing units, i.e., each processor not only has circuitry that interprets an instruction and causes control action, but also is able to communicate with every other processor and to operate on any data. In a parallel computer these abilities are needed for the efficient running of large programs and for programs well suited to parallel computation. But the

organization must also provide for a partitioning of the machine so that a number of small programs could run simultaneously without interaction. This is usually referred to as "interprogram protection" and would need to be under program control to be completely versatile. Since processing units can interact directly with each other there is no need for a central control. In fact, any distinguished or superior processing unit would unreasonably complicate programming.

The organization should allow the size of the machine to be flexible, a variety of I-O devices to be provided, and a powerful set of operations to be available for the benefit of the programmer. In particular, there should be complete flexibility with respect to the number of instructions and number of data; the only restriction being that their sum does not exceed the storage capacity of the machine. Further, it would be convenient to the programmer and conservative of storage if one instruction could cause an operation to be performed at a number of locations simultaneously. For example, a single instruction could cause one number to be added to the contents of many memory locations.

In addition, the hardware should be able to accept an arbitrary number of instructions for execution at any given program step. If all instructions can not be processed simultaneously, then the computer should process them in groups. When all instructions for a program step have been completed, the next program step should begin executing all instructions designated as successors by the instructions of the immediately proceeding program step.

And finally, in the light of previous requirements, it would be unrea-

sonable to cause any computation bottleneck due to a shortage of arithmetic units or delay while accessing data. Therefore, every memory location should be directly accessible by an arithmetic unit. The computer, being equally limited by computation and memory access, could employ lookahead efficiently, thus enabling the greatest overall computation speed.

These objectives form a basis from which a very powerful computer organization can be developed. We realize, of course, that there are other objectives which might be added or substituted to meet other criteria. Our choice of objectives is based on the fact that a machine organization fulfilling these objectives could substantially reduce average computation time for some problems as compared to a computer with a single processor constructed from similar components.

As might be expected, any computer fulfilling these objectives would require many times the number of components in existing computers. Potentially inexpensive components and useful construction techniques are presented in Section 3.5.2 of this report. A brief look at cost versus problem-solution time indicates such a machine would be uneconomical in the next year or so. Yet, technological improvements that reduce the cost of logical components without necessarily increasing their speed, plus reasonable development of parallel programming techniques, could make such a machine economically competitive in the near future.

3.2.3 Organization

The following computer organization meets the objectives outlined above

and has some novel features for logical design and physical construction.

The computer consists of many identical modules, blocks of logical circuits, imbedded in a passive connecting network. Each module contains a basic arithmetic unit, storage for one word of data or one instruction, and some control circuitry. There is a central timing and synchronization but no other common memory or control units. This is sufficient to form a complete general-purpose computer minus input-output equipment.

Our main consideration is the description of a module, of module connection, and of program execution within this computer. Input-output devices are to be connected directly to modules, with, at most, one per module. In this way, arbitrarily many I-O devices can be operating simultaneously without slowing down computation in other modules.

Since we are considering a highly parallel computer we wish to allow arbitrarily many instructions to be executed simultaneously. Rather than having instruction counters hold the locations of the next instructions to be executed, an additional bit position, the execution bit, is appended to each memory location. At the time when execution is to begin, the contents of each memory location having an execution bit equal to 1 is executed as an instruction. A 0 then replaces the 1 in the execution bit of those locations from which instructions were just executed. Thus an instruction specifies its successors, if any, by setting the execution bit to 1 in the memory locations of the instructions to execute next. To avoid the priority problems of assigning instructions to be executed to processing units, each memory location has an instruction processor directly connected to it.

The instruction processor operates, or is active, only when the execute bit is 1 and the execute signal is received from the central timing circuits. The function of the instruction processor is to route operands to an arithmetic unit with information as to what operation is to be performed.

To avoid the problem of assigning arithmetic units to active instruction processors, each memory location has its own arithmetic unit. The memory register itself serves as the accumulator, the register that contains the operand which is to be used and then replaced by the result of an arithmetic operation. Thus, by using the arithmetic unit at the location where the result is to be stored, there will never be a time when an instruction processor must wait for an arithmetic unit. An attempt by two instructions to store information in the same location at the same time is considered a programming error.

An additional feature of having many arithmetic units is to allow one instruction to specify that an operation is to take place at many locations simultaneously. The addressing of many locations by a single instruction is accomplished by indirect addressing.

As might be expected, in this machine the data accessing for arithmetic operations is considerably different from conventional computers. No fetching of instructions is required, thus the data accessing circuits can be simplified and computation speed is increased. Even with this simplification, far too much circuitry would be required if each instruction processor needed the ability to access every memory location directly. To cut down the number of components and yet keep flexibility of accessing the following or-

ganization is used:

For each memory location there is a module containing the memory register, instruction processor, arithmetic unit, and what we will call path-connecting circuitry. Each module has direct connection, by wire without gates, to a few other modules. For one module, say X, to gain access to a module not directly connected, say Y, the destination (address of Y) is gated onto the wire that directly connects X to a module closer to Y. As soon as a path has been completed from X to Y, X has access to the memory register and arithmetic unit in the Y module. In this way every module can have access to every other module while having a direct connection to only a few modules.

One of the most significant factors in the design of such a machine is the logical organization of path segments (directly connected modules). The two extremes of path segment organization are: (1) every module connected to just two other modules (geometrically the modules could be placed in a line with wire connecting adjacent points on the line and the two end points), and (2) every module connected to every other module (geometrically, k modules would form a k-1 dimensional simplex).

Neither of these seems acceptable for the general-purpose computer being considered by this report. The line of modules uses less components than any other but relatively few accesses could be made simultaneously, e.g., several short paths could isolate many modules from others to which access is required.

Let the machine under consideration have 2^n modules. Now, as a compro-

mise between the number of components and expected number of simultaneous accesses, let each module have a direct connection to n other modules. Thus the number of direct connections is a function of the size of the machine. For a 32,768-word machine each module would be connected to 15 other modules. The logical organization of these connections would be to have the modules as the vertices of a 15-dimensional cube with the edges of the cube being the direct connections between modules. Since each module can be represented by a unique 15-bit number, the direct connections correspond to a wire from each module to the 15 other modules whose numbers differ in one bit position, i.e., unit Hamming distance.

There is a physical construction whereby the wires for the direct connections can be laid out in n layers for a machine with 2^n modules. The modules are laid out in a two-dimensional square array as shown below. Each layer contains exactly one connection for each module and no connections cross within a layer. The layers could be made by deposition or printed circuit techniques. Only n/2 masks would be required and each mask would have 2^{n-1} lines on it formed from $2^{n/2}$ repetitions of a $2^{\frac{n}{2}-1}$ line pattern, e.g., for a 4096-module machine n = 12; thus each mask would have 2048 lines formed from 64 copies of a 32-line pattern.

The mask, layers, and composite view of a machine with 16 modules are given below:

There are several interesting measures of accessibility which depend on the logical organization. First, the maximum length, in number of segments, that any minimal path may be is n in a machine with 2^n modules, i.e., maxi-

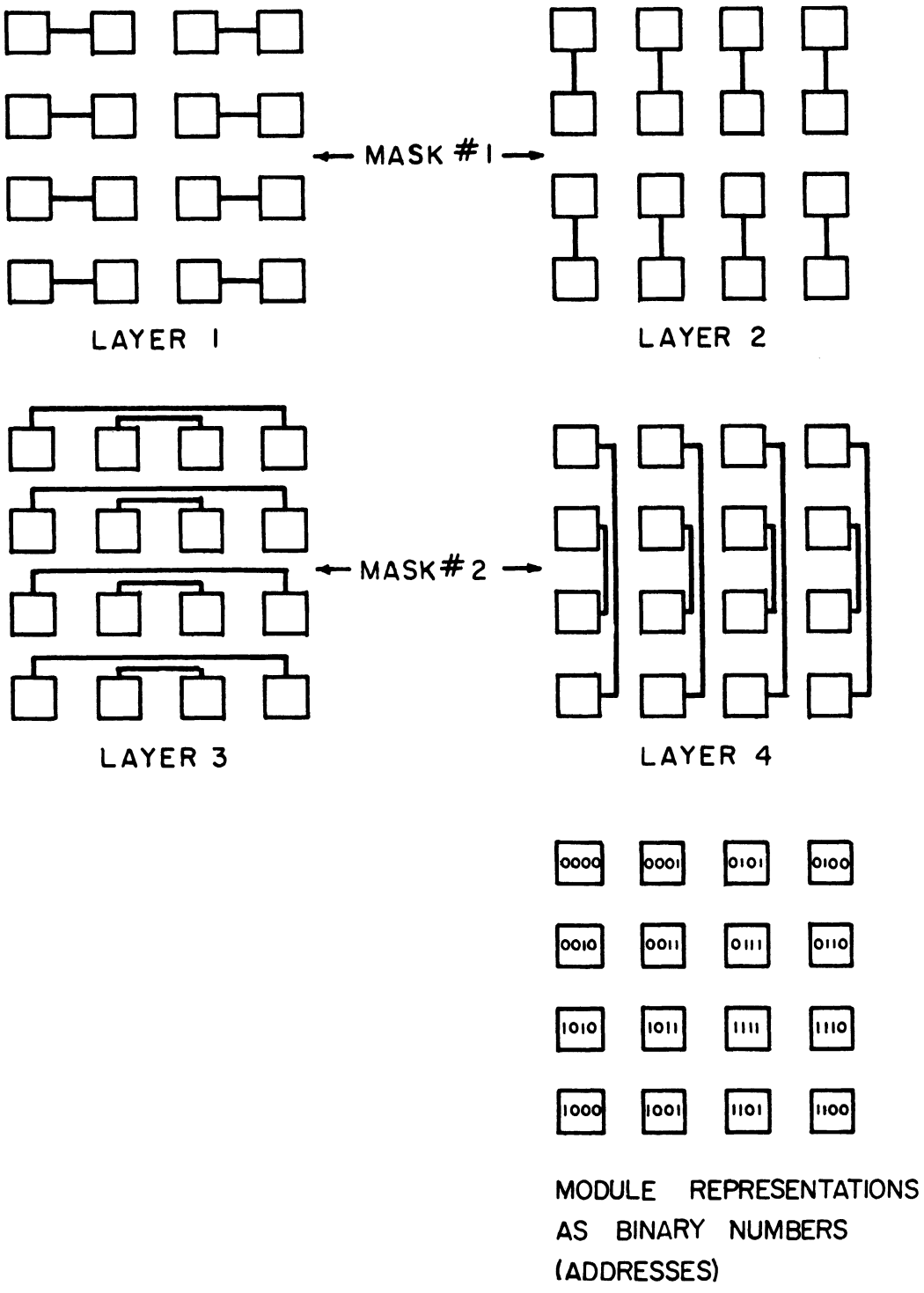


Fig. 58. Detail of path segment wiring.

mum Hamming distance between two n-bit numbers is n. Second, the number of different paths between two modules differing in k bits is k! i.e., all permutations in the order of reducing the Hamming distance by 1 each step for k steps. Finally, statistically the expected number of simultaneous accesses that could be made in a 4096-module machine is over 300, assuming random storage assignment of data and instructions. Of course, instructions and data are not randomly assigned storage. Considering the timing factor on accessibility that each module is directly connected to only a few others, it is not difficult to see that clever programming could yield many more simultaneous accesses than the random case, while intentionally poor programming could yield many less.

Due to the inherent limitation on parallel accessing as the number of paths increases, it seems advisable to remove all path connections when the access has been completed. In this way each step in the execution of a program starts with an uncluttered machine.

Actually, by allowing the machine to have some paths still connected when the next execution step begins, there can be a path-connecting lookahead which could, in general, speed up computation more than no lookahead and an uncluttered machine. Preliminary logical design of a module indicates that clever logical circuit design could make the average path-connecting time about the same as the longest arithmetic operation time. Thus the average time to perform an operation becomes equal to the time required by the slowest operation, but there is no accessing time required during a sequence of execution cycles.

To allow simultaneous path connecting from an active instruction to the

first operand (also arithmetic unit), to the second operand, and to the succeeding instruction, three independent path-connecting circuits are provided.

There are no index registers (relative addressing) as exist in conventional computers. This is necessary due to the unconventional scheme of accessing. In place of index registers, operations are provided so that one instruction can do arithmetic directly on the address part of another instruction, i.e., the address part of every instruction is essentially an index register.

To further supplement addressing an indirect address can be specified. When a path has been connected from an instruction to some module, say X, and if the instruction specified indirect addressing, the path is extended according to the address in the memory register of X. The address in X may also be designated as indirect. Since the memory register of X is large enough to hold several addresses, each address position is interpreted and each can start an extension of the path into X. In this way one instruction with an indirect address can refer to a memory location with several indirect addresses, each of which can refer to other locations, etc. Thus, one instruction can control the arithmetic units of many randomly placed modules simultaneously.

A more detailed description of this machine's operation is given in the next section which is essentially a programming manual. A more detailed description of the logic follows that section.

3.2.4 Instruction Code

The programming of an iterative circuit computer must be flexible enough

to justify having a highly parallel computer rather than a number of single-processor computers. Since it is possible that hundreds of instructions could be executing simultaneously and these instructions could be using the same data, the hardware must provide some basic synchronization of instructions. Therefore, in order to simplify programming, the computer execution cycle proceeds as follows: a number of instructions are being executed simultaneously; each specifies locations of instructions to be executed next; when all instructions have completed execution all of the "next" instructions start executing simultaneously; and so on.

Thus the execution of individual instructions is asynchronous, but the execution of sequences is synchronous. Even if the programmer specifies more instructions than the machine can execute simultaneously, the hardware is set up to process all of them in several bunches. Then, when all have been executed, the "next" instructions are started.

3.2.4.1 Instruction Format

Every instruction has the same basic format: an operation code and three addresses.

The first address, α, may designate the location of one operand for arithmetic and logical operations. The result of the arithmetic and logical operations replaces the contents of α. With conditional transfer operations, α may be used to specify the location of the next instruction.

The second address, β, may designate the location of the second operand for arithmetic and logical operations, i.e., multiplier, divisor, etc. For some conditional transfer instructions, β is the location tested for the con-

dition. With shift instructions, β is the shift count rather than an address.

The third address, γ, designates the location of the next instruction. For some conditional transfer operations, the condition determines whether α or γ specifies the next instruction.

The number of bits and relative positions of an instruction are shown in the following figure: (n might range from 10 to 20 depending on the number of locations, 2^n, in the computer).

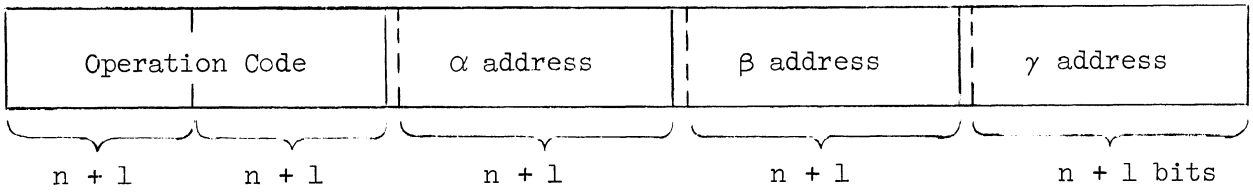


Fig. 59. Instruction format.

The word length is $5(n+1)$ bits. If the leftmost bit of α, β, or γ is 1, then that address is indirect. The remaining n bits specify the location to be used.

The three-address scheme allows flexible arithmetic and control instructions to aid parallel programming and spatial program organization.

3.2.4.2 Execution Bits

There are three more bits at each location to control execution instructions, called e_1, e_2, e_3. The e_2 bit of a location is set by any instruction referring to that location as a successor by a γ address, i.e., the contents of the location where the e_2 bit is set will be active, or execute as an instruction, during the next execution cycle. The e_3 bit of a location is set if the contents of the location are to be inactive during the next execution

cycle. The e_3 bit is set rather than the e_2 bit depending on the operation code.

At the beginning of each execution cycle, the e_1 bit is set if the e_2 bit was set and the e_3 bit was not set during the previous instruction cycle. If both e_2 and e_3 were set (by different instructions), the e_1 bit is not set. Once the e_1 bit is computed, both e_2 and e_3 are reset.

These three bits influence execution in the following way: once the e_1 bit has been determined at every location, the instructions in all these locations become active. Those instructions which successfully completed paths for α, β, and γ accesses reset their e_1 bit and perform their operations. Some of these instructions will be setting e_2 and e_3 bits of other locations. While operations are being performed by these instructions, the others with e_1 bit set but paths not completed try again to connect their α, β, and γ paths. This process repeats itself, possibly requiring a number of attempts for some instruction to complete its path. The execution cycle terminates when all e_1 bits have been reset. At this time, the next execution cycle begins with the computations of e_1 bits as specified by the instructions of the preceding execution cycle setting the e_2 and e_3 bits.

The hardware has been designed so that a large number of instructions can simultaneously have their paths connected. After an instruction has completed its operation, its paths are removed (disconnected). A priority scheme has been developed which allows any number of paths to be forming simultaneously, and which also guarantees that at least one path will be connected on each attempt. Therefore, in the worst possible case, the time re-

quired to execute a group of instructions activated during a given execution cycle will never exceed the time required to execute them sequentially.

3.2.4.3 Interprogram Protection

To provide isolation of instructions and data, an additional bit is required at each memory location. If this 'isolation' bit is set in some module, the hardware will not allow a path to be built through the module, but the module may still be a path termination. The ability to be a termination is necessary in order to allow for the resetting of the isolation bit.

To isolate a program, those locations containing instructions on data which form a spatial boundary must have their isolation bits sets. If all programs in the machine have their boundary isolation bits set, there will be a barrier that prevents any program from accessing any other program.*

3.2.4.4 Indirect Addressing

The contents of a location referred to as an indirect address are interpreted as shown in the figure below. There are five possible addresses and any combination may be used. An unused address is recognized by the fact that it is all zero. Any combination of addresses may be specified as indirect by setting the leftmost bit of these addresses. The computer timing imposes a limit of 40, at most, on the length of a sequence of dependent indirect addresses.

---

*The isolation bit can be set or reset by the LOAD instruction with the appropriate modification of the operation code.

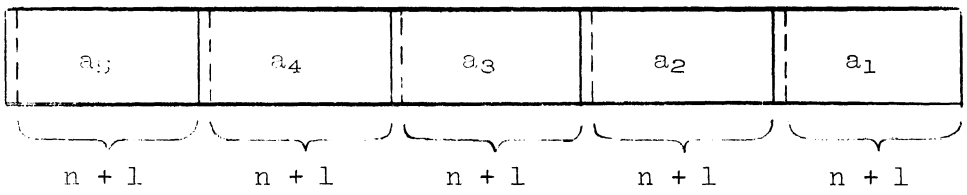


Fig. 60. Location referred to by indirect address.

Through the use of indirect addressing it is possible to have one instruction perform its operation on the contents of many locations simultaneously. This is done by having α be an indirect address. The contents of the location that α refers to can have up to five more indirect addresses, each of which can refer to five more, etc. Thus, a tree structure of paths is connected from the instruction to many modules. Upon execution of the instruction, the operation code followed by the second operand is sent down the tree and all the terminal modules perform the operation simultaneously.

Similarly, γ can specify one successor directly, or many successors, through indirect addressing.

3.2.4.5 Arithmetic Operations

The four basic arithmetic operations -- addition, subtraction, multiplication, and division — are available. The normal mode of full-word arithmetic is floating point. The mantissa is shifted to make the characteristic zero whenever no loss of accuracy occurs. In this way the programmer has the benefits of high speed when working with integers, and full accuracy by automatic scaling of non-integers. The format for full-word numbers is given below. The magnitude of a number is the mantissa (binary point to right of low-order bit) times two raised to the power of the characteristic.

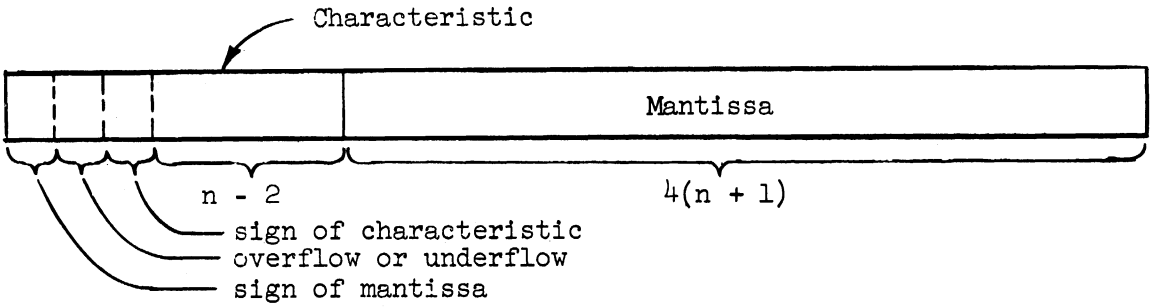


Fig. 61. Format for full-word number.

Each number has an overflow-underflow bit which is set if at any time the magnitude exceeds $2^{[2^{n-2}-1]\left(2^{[4(n+1)]}-1\right)}$ or is less than $2^{-[2^{n-2}-1]}$. When the overflow bit is set, the remaining bits of the number are reset, made zero. Any succeeding arithmetic operation on a number with its overflow bit set results in another number which also has its overflow bit set. Normal arithmetic is performed on numbers even if their overflow bits are set. Overflow or underflow can occur only on a full-word addition, subtraction, multiplication, or division. There is also a conditional transfer instruction capable of testing the overflow bit.

3.2.4.6 Byte Modification

To facilitate relative addressing and instruction composition, the arithmetic operations add and subtract, as well as load, complement, and, or, exclusive or, and some conditional transfer instructions can refer to any of the five n+1 bit bytes. For designating which bytes are to be operated on there are five bits in the operation code which can be used to modify these operations.

0 0 0 0 0	specifies full-word arithmetic. Fig. 60.
0 0 0 0 1	specifies the operation is to be performed on the low-order n+1 bits (a_1) of the operands.
.	
.	
:	
1 0 1 1 0	specifies the operation is to be performed on the a_5, a_3 and a_2 bytes.
.	
.	
.	
1 1 1 1 1	specifies the operation is to be performed on all five n+1 bit bytes.

All bytes are assumed positive, a negative result remains in the byte as its 1's complement. There is no carry from one byte to the next. Carry-out of the high-order position is lost.

These partial word operations may be used directly on instructions or indirect address words as if the locations were index registers.

3.2.4.7 Transfer Instructions

Since each instruction has a γ address to specify the location of its successor, only conditional transfer instructions are assigned specific operation codes. There are two basic forms of conditional transfer instructions: BRANCH and PROCEED.

The BRANCH operation tests the contents of the location specified by the β address. If the condition (specified in the three low-order bits of the operation code) is met, a signal is sent to the modules designated by the α address. Otherwise, the signal is sent as usual to the modules designated by the γ address.

The PROCEED operation compares the contents of the locations specified by α and β. The comparisons such as $=$, $\neq$, $>$, etc. are specified by the three

low-order bits of the operation code. If the comparison is not true, the instruction is treated as if no successor were specified. Otherwise, the successor specified by the γ address is treated in the normal way.

3.2.4.8 Inhibit Modification

The use of the word 'signaled' rather than 'transferred to' is necessary because in this machine many instructions can be executing simultaneously. It is possible for an instruction to specify itself as its successor for incrementing or counting purposes. Another instruction could be testing for a desired value. When this value is reached there must be a way to stop the instruction which is transferring to itself. The ability to stop an instruction from executing during the next execution cycle is called the inhibit modification. Every instruction has a bit in its operation code which if 1 causes the signal to the successor to set the e_3 bit (described on page 146). If the inhibit modification bit is 0, the signal goes to the e_2 bit of the successor. In either case, the effect of a signal applies only to the next execution cycle.

Any instruction can be a local HALT instruction by having an all-zero γ address. i.e. no successor.

3.2.4.9 Input-Output Instruction

The α address of the input-output instruction refers to a module which is directly connected to a particular I-O device of the desired type. The memory register of this module may contain control information for the I-O device. The location of information which is entering or leaving the com-

puter is specified by the β address of the I-O instruction.

Each I-O device has its own simple buffer between itself and the main computer. For a magnetic tape unit, the buffer may be core storage which holds several blocks of information. As long as there is information available on reading or space available on writing, the main computer uses only a normal length of execution cycle for an I-O operation. If the buffer is empty or full, execution is held up until the I-O operation is completed. Backspace, rewind, skip file, etc. are determined by the information in the location connected to the I-O device. These require only a normal length execution cycle unless the queue of commands exceeds the buffer capacity, in which case further execution in the main computer must wait.

With this type of I-O, the programmmer should give control information as early as possible and do information I-O at the last possible moment.

3.2.4.10 Operation Codes

Arithmetic

1.	ADD α, β, γ	The contents of α are replaced by $\alpha+\beta$. (Byte or full word)
2.	SUBTRACT α, β, γ	The contents of α are replaced by $\alpha - \beta$ or $\beta - \alpha$, depending on the high-order bit of the operation code being 0 or 1 respectively. (Byte or full word)
3.	MULTIPLY α, β, γ	The contents of α are replaced by $\alpha \cdot \beta$. (Full word only)
4.	DIVIDE α, β, γ	The contents of α are replaced by α/β or β/α, depending on the high-order bit of the operation code being 0 or 1 respectively. (Full word only)

Logical - Byte modification applies to 5 thru 9

5. LOAD α, β, γ — The contents of α are replaced by the contents of β.

6. AND α, β, γ — The contents of 2 are replaced by the bit wise AND of α with β.

7. OR α, β, γ — The contents of α are replaced by the bit wise OR of α with β.

8. EXCLUSIVE OR α, β, γ — The contents of α are replaced by the bit wise EXCLUSIVE OR, ring sum, of α with β.

9. COMPLEMENT α, γ — The contents of α are bit wise complemented.

Shifting - Full word only

10. SHIFT α, β, γ — β is not an address. β is the number of bit positions the contents of α are to be shifted. The first and second bits of the operation code being 1 and 0 respectively determine left or right and end around or linear. Vacated positions on linear shifts are filled with zeros. A shift instruction with $\beta = 0$ is a NO OPERATION that requires 1 execution cycle and can specify a successor.

11. SCALE α, β, γ — The contents of α are treated as a floating point number. The low-order n-1 bits of β are treated as a sign and magnitude of a characteristic. β is not an address. If the first bit of the operation code is 1, then the mantissa in α is shifted so as to make the characteristic equal to β. If the first bit of the operation code is 0, then β is added to the characteristic in α and the mantissa in α is shifted accordingly. The second bit of the operation code being 1 or 0 specifies rounding or truncation respectively.

Transfer

12. BRANCH if $\mathcal{R}(\beta)$ α, β, γ — If $\mathcal{R}(\beta)$ is true, a signal is sent to location α, otherwise the signal is sent to γ. $\mathcal{R}(\beta)$ may be any of the following:
a) $\beta = 0$ (Byte or full word)
b) β negative

c) β has overflow bit set

13. PROCEED if α ~ β α, β, γ

If α ~ β is true, a signal is sent to location γ, otherwise no signal is sent. ~ may be any of the following:
a) $\alpha > \beta$
b) $\alpha \geq \beta$
c) $\alpha = \beta$
d) $\alpha \leq \beta$
e) $\alpha < \beta$
f) $\alpha \neq \beta$
All relations above may apply to byte or full word.
g) the βth bit of α is a 1
h) the βth bit of α is a 0
If α refers to more than one location through indirect addressing, the logical OR of the α's will be used to test the relation.

Other - Full word only

14. INPUT-OUTPUT α, β, γ

α is the module which controls the I-O device. The memory register of α contains the command for the I-O device while β specifies the location into which information is read, or out of which information is written.

15. SENSE PANEL α, γ

The contents of the display panel is the address of the last module where an unresolvable programming error was detected. e.g., trying to execute an undefined opeation code. The contents of the display panel replace the β address position of location α.

16. SET ISOLATION α, γ

The isolation bit is set to 1 at location α. α may still be referred to by other instructions but no access can be made which would use a path through α.

17. RESET ISOLATION α, γ

The isolation bit is reset to 0.

18. ERROR MODE γ

Depending on the first two bits of the operation code being 1 or 0, this instruction sets the mode of operation to:

continue or stop executing instructions of type 1 thru 18 and activate or inhibit ERROR START instructions respectively. The mode remains set until changed.

19. ERROR START γ — If there is an error and the computer is in ERROR START activate mode, all instructions with this operation code become active during the next execution cycle.

This concludes the description of instructions available in the hardware of the computer. Because many operations have bits which further qualify them, an assembly language distinguishing the various operations would be useful to programmers. The operations are meant to be convenient to the general-purpose programmer. Many special instructions, symbol and list manipulation, etc., have purposely been omitted to keep the amount of hardware to a minimum. This should cause no loss of speed since special instructions can be achieved by clever programming using simultaneous application of those instructions given above.

3.2.5 Physical and Logical Design

Now comes the problem of determining how much circuitry would be required by a machine as described in the earlier sections of this report. The most accurate way to determine the required number of components is to do a complete logical design. Even then, the cleverness of the logical designer and the choice of component types could affect the result by a factor of two or three. Considering the time involved to do a detailed logical design and considering that we are far from the cleverest logical designers available, the following approach was taken: The part of this machine not found in con-

ventional computers, the path-connecting circuitry, was designed in some detail. The logical circuitry for arithmetic operations, timing pulse generations, etc., was not designed. Instead, their requirements are given with estimates for the number of components required based on current technology.

We will first consider the somewhat conventional hardware that must be in each module. Even here the logical design would not really be conventional. Where, at most, hundreds of computers of a given type may have been built, we are talking of building thousands of modules for a single machine. Although a module is versatile when embedded in an I.C.C., it is far from being a complete computer. Thus, due to the greater importance of economical design and lesser requirements, a greater effort could be justified for a fully integrated, clever, logical design. A number of trial modules could be built, tested and perfected with the goal being low costing mass-produced modules.

There are several components which could be used in the construction of modules. For example, RTL circuits can be produced fairly inexpensively in quantity using low-speed, low-power transistors. The RTL circuits which are currently being manufactured by deposition techniques have a density of about 100 transistors and 400 resistors per square inch. Circuits such as these used in an I.C.C. have the advantage of less noise pick-up since the physical size of a module can be small and the connecting leads between modules can be correspondingly short. A second component potentially useful for module construction is all-magnetic logic. Again, this is not the fastest possible logical component but it is reliable and potentially can be manufactured by automated equipment. Multi-aperture cores and other types of all-

magnetic logic require fairly close tolerances, thus careful design. Moreover, this type of design is well suited to modules which have relatively few external connections to other modules. A final example of a potentially inexpensive and fast component is the cryotron. Again, automated production may be possible, and making a large number of identical modules should reduce considerably the cost per module.

At first glance, everyone considers an I.C.C. impractical, even with inexpensive construction, since it could have thousands of modules which seem to be simplified versions of processing units in conventional computers. Although an I.C.C. requires many times the number of components in conventional computers, one cannot expect to get simultaneous accessing, simultaneous arithmetic,and simultaneous instruction processing without more components. To show that a module is far less than the processing unit in conventional computers we will list all the circuitry that is not in a module but is in conventional processing units.

First, there are several obvious registers that are not required in a module. There is no sense (storage) register or address register since there is no store to access. There is no instruction register or instruction counter since execution, not instructions, moves from module to module. The one-word store in the module corresponds to a conventional accumulator. By having numbers in the integer form with scale factors, the multiply and divide operations can be performed in a single-length accumulator.

The next major block of circuitry, not within modules, is for timing. There could be one timing unit which would make all the required sequences

of control pulses available to all modules. By having more specific timing sequences available than in conventional computers, the amount of logic in a module can be greatly decreased. The central timing unit becomes correspondingly larger, but the component saving in one module multiplied by the number of modules should be far greater.

Considering that instructions are all the same basic format and are relatively stationary, a scheme exists for having various bits in the memory register of a module directly control gates when the module is executing an instruction. This would eliminate most of the instruction decoding circuitry existing in conventional machines.

There would have to be a basic adder and the arithmetic control logic in every module. Here, a decision between serial and parallel arithmetic would have to be made based on the differences in speed and cost.

The remaining circuitry in a module is the path-connecting logic. In place of drivers, cores, and sense amplifiers, the path-connecting logic closes gates in various modules, forming a path to access information in other modules. To give an idea of how much circuitry is required for path connecting, a fairly complete logical design of this follows:

The basic segments from which paths are formed are conductors from the periphery of one module to the periphery of another. Fig. 62 shows the top view of an I.C.C. with the modules appearing as squares. Fig. 63 shows how each module passes through a number of thin layers on which the path segments (the conductors) are placed (by printed circuit or deposition techniques). The addresses α, β, and γ are shown here to have completely isolated path

structures. Correspondingly three times as much path-connecting circuitry would be required in each module. The choice of separate layers for each address can stem from the fact that this is well over three times as fast from computation standpoint and yet requires less than three times the circuitry since each layer can be specialized to its particular address bit positions.

Before describing the logical circuitry for path connecting, we will explain the function the circuitry must perform. Basically, the problem can be stated as follows: There is a binary number in the memory register of some module. This binary number refers to another module. The circuitry must close gates to form a path between these two modules. The path must allow information to flow in both directions. A path need not be a single wire; it could be physically a bunch of wires for transmission by bytes or in parallel, and there could be two separate circuits for transmission in each direction. For convenience of explanation and simplification of logical circuitry, a scheme with one wire for each direction will be used. See Fig. 64. The connections to the module labelled P_1 through P_6 are physically n wires (where there are 2^n modules in the computer). Information can flow in P_1 out P_2 , in P_3 out P_4 , and in P_5 out P_6 without affecting the operation of this module. This module may initiate paths along a wire of P_2 , P_4 , and P_6 as determined by the α, β, and γ addresses. Other modules may access this module by having their paths terminate in the P_1 , P_3, or P_5 lines of this module. When this module is used as the α address of some instruction, the operation code of the instruction enters before the β operand. The operand code enters via some wire of P_3 and is placed in the arithmetic control ope-

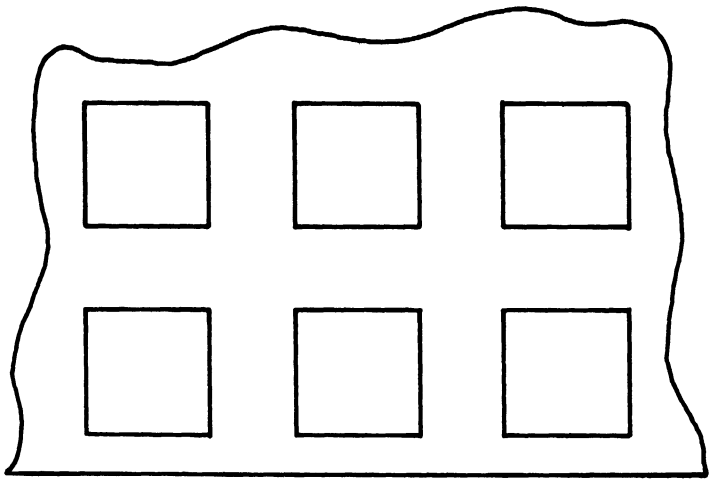


Fig. 62. Top view of the I.C.C.

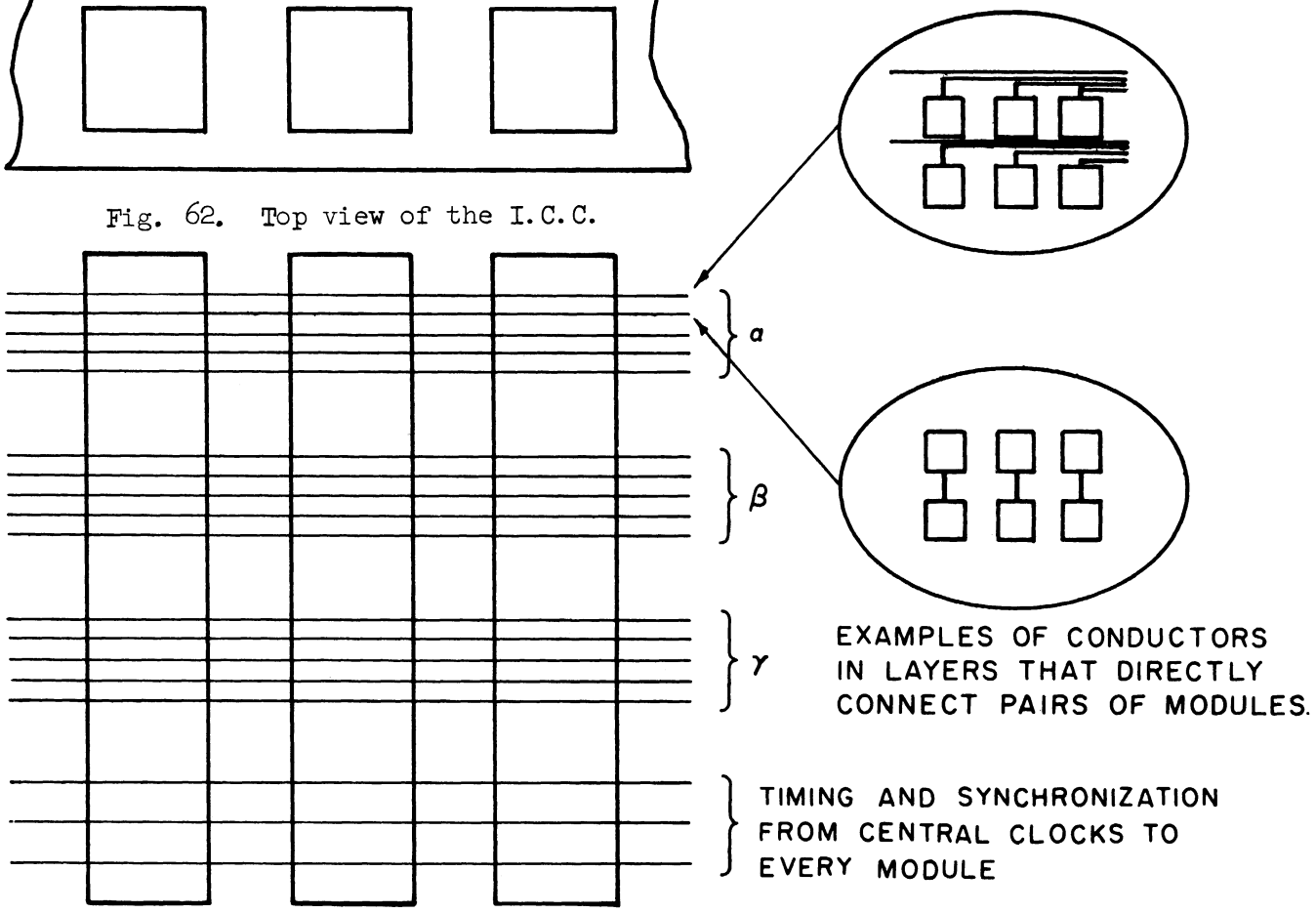


Fig. 63. Side view of the I.C.C.

ration register.

The control of arithmetic operations in this module comes from the operation register and not from the memory register. No addresses need be sent to the module acting as an arithmetic unit since the module containing the active instruction is doing the required switching to set up the operand accesses.

Figure 65 shows the significant information flow when an instruction executes. In this example, the contents of the memory register of module Y are being added to the memory registers of modules X_1 and X_2. The ADD instruction is in module R, and indirect addresses are in module X.

Execution proceeds as follows:

Step 0 — This e_1 bit (described on page 146 of this report) is set assuming an activate signal was sent to module R over one of its P_5 paths. The e_2 and e_3 bits in R are reset.

Step 1a — A path is connected from R to X. (The prime indicates that X is an indirect address.) Then two paths connect from X to X_1 and X_2 respectively.

1b — A path is connected from R to Y (second operand).

1c — A path is connected from R to S (next instruction).

Step 2a — The e_2 bit in module S is set. Removal of the path between R and S begins at S.

2b — The operation code from the memory register of module R is sent to the operation registers of X_1 and X_2 via X.

Step 3 — Module R controls the gating of the contents of the memory register of Y to the path into X. Module X controls the gating from its P_1 input to the two lines to X_1 and X_2. X_1 and X_2 set their gates to send the contents of their memory registers and the incoming path from X to their adders respectively.

Step 4a — X_1 and X_2 gate the resultant sum back to their respective memory registers.

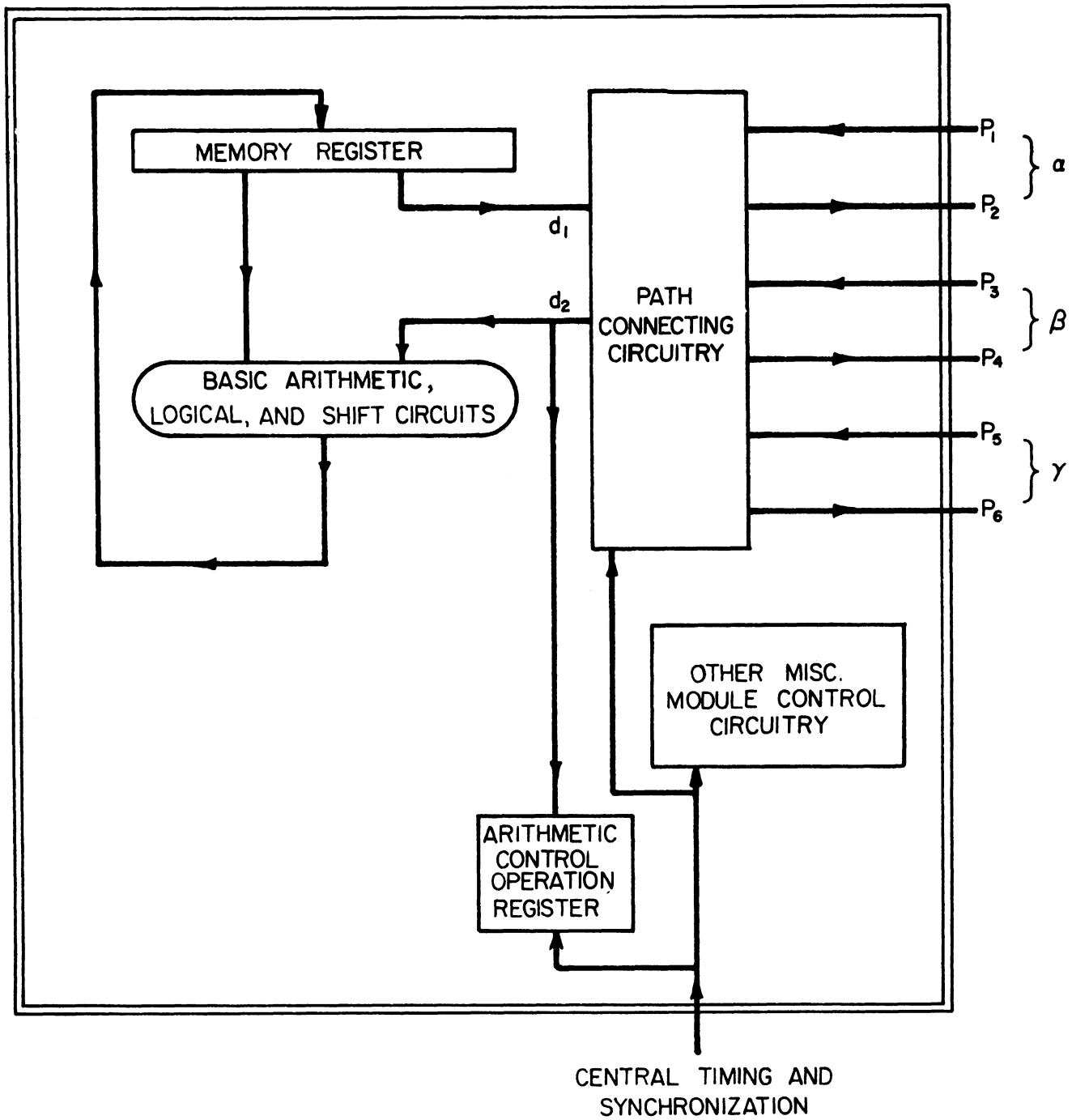


Fig. 64. Function and flow block diagram of module.

4b Removal of the paths into X_1 and X_2 is begun at X_1 and X_2 respectively.

4c Removal of the path from R to Y is begun at Y.

Step 5 When all paths have been removed to R, the e_1 bit of R is reset.

Step 6 When all e_1 bits are reset the central synchronization emits a signal to all modules which compute the new e_1 bits and Step 0 begins again.

This completes the description of Fig. 65 involving the overall path structure. We will now concentrate on one type of path, say α. (For convenience of construction, all three types of paths, α, β, and γ would probably be the same logic, or all three could be operating simultaneously in the same circuitry if some restrictions were placed on programming.)

The decision procedure for connecting a path that must be performed in each module requires two pieces of information. Each module must know its own binary representation as an address, called 'HERE (This can be wired into the layers shown in Fig. 58 thus allowing all modules to be identical and interchangeable.) Also, each module must know which of its accessible n path segments are busy. (This we will call the 'BUSY' register.)

The n bit address of the termination of a path can come from n+5 places, i.e., n from the n path segments connected to this module plus 5 from the 5 byte positions of the memory register. For instructions, only the three low-order bytes are addresses which could initiate paths but an indirect address can cause all 5 bytes to initiate paths.

Suppose an n bit address has reached a module. This is the address of the termination of a path. By taking the bit-wise exclusive or of this n bit

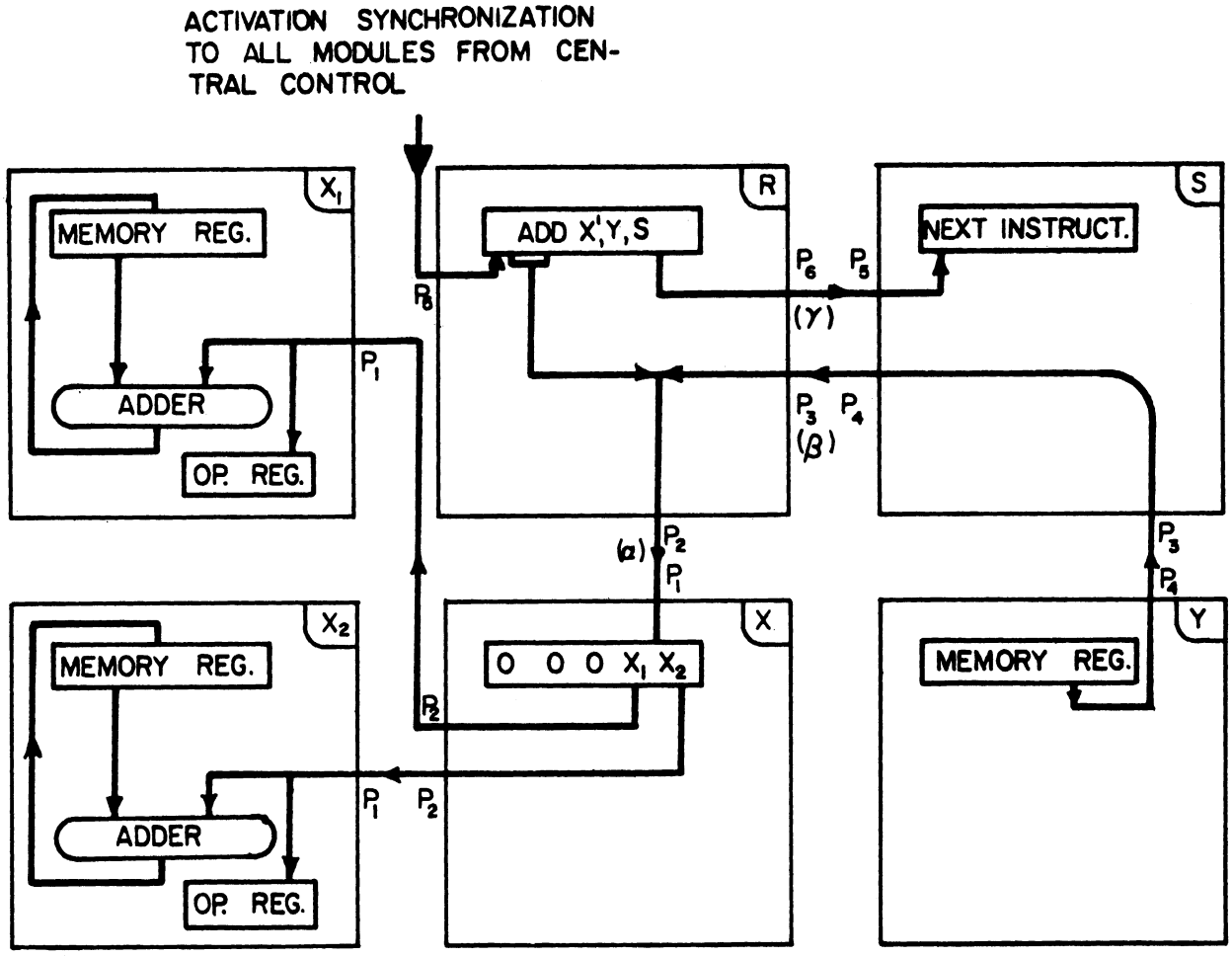


Fig. 65. Information flow during execution.

address with the n bit representation 'HERE', those positions of the result which are 1 denote the possible path segments which can serve as extensions for the path. This is just a reduction of 1 in the Hamming distance since each neighbor of a module differs from it in exactly one bit position. We will establish the convention that the lowest 1 bit resulting from the exclusive or will be tried first as an extension of the path. It may be that the desired segment is already being used by another path, in which case the BUSY register has a 1 in that position. To eliminate busy segments from potentially useful segments, the complement of BUSY is added to the result of the previous exclusive or. This result is retained in a 'GOING' register.

To connect from the n possible incoming path segments to the n possible outgoing segments an **n x n** switching matrix is used. Five more inputs are appended to the switching matrix to allow for path initiation, and a diagonal pair of wires allow for path termination at a module.

The operations of path connecting are staggered such that all modules with an even number of 1's in their addresses, 'HERE', extend (or remove) their paths one segment during alternate times with modules having an odd number of 1's in their addresses. In this way, priority problems are avoided which involve two adjacent modules trying to connect to their common segment. It is possible to have two modules connect a path to the same module at the same time and have the same destination for both paths. This priority decision is made by the circuitry just prior to setting the gates of the switching matrix. The circuit for this two-dimensional priority selector is shown in Fig. 67. The composite of the logic just described is shown in Fig. 66. The

one-directional segments are shown as B_1, B_2, B_3, ..., B_n grouped under the P_1 designation. The one-directional outputs are grouped under the P_2 designation. For a path to pass through a module two inputs will be connected to two outputs with reversed subscripts, thus forming a piece of a two-directional path.

If a path cannot be extended due to complete blockage by other paths, a NO-GO signal is sent back towards the origination of the path. Upon receipt of a NO-GO signal, a module selects the next (higher order) potentially useful segment from the 'GO' register.

To further explain the logic involved, an example of the progression of a path connection is given in Fig. 68. Here we have a machine with n equal to 4. Only 8 of the 16 total modules are shown and only the values of 'HERE', 'DESTINATION', 'BUSY' and 'GO' are shown in boxes. The path-segment connections B_1, B_2, B_3, B_4 each correspond to a pair P_1 and P_2 shown in Fig. 66.

We will concern ourselves with the path originating at module 0001 with the destination 1010. We assume two other paths, indicated by ..... and ----, are already present. The path connecting proceeds in two phases. During phase A, the modules with an odd number of 1's in their address perform the logic to compute the contents of their 'GO' registers, and the modules with an even number of 1's in their address transmit the 'destination' over the path segment specified by the lowest 1 bit in their 'GO' registers. During phase B, the roles of the two sets of modules are reversed. To get the phasing started there is no transmission between modules on the first step and no logic on the last step of path connecting. Thus we have for Fig. 66:

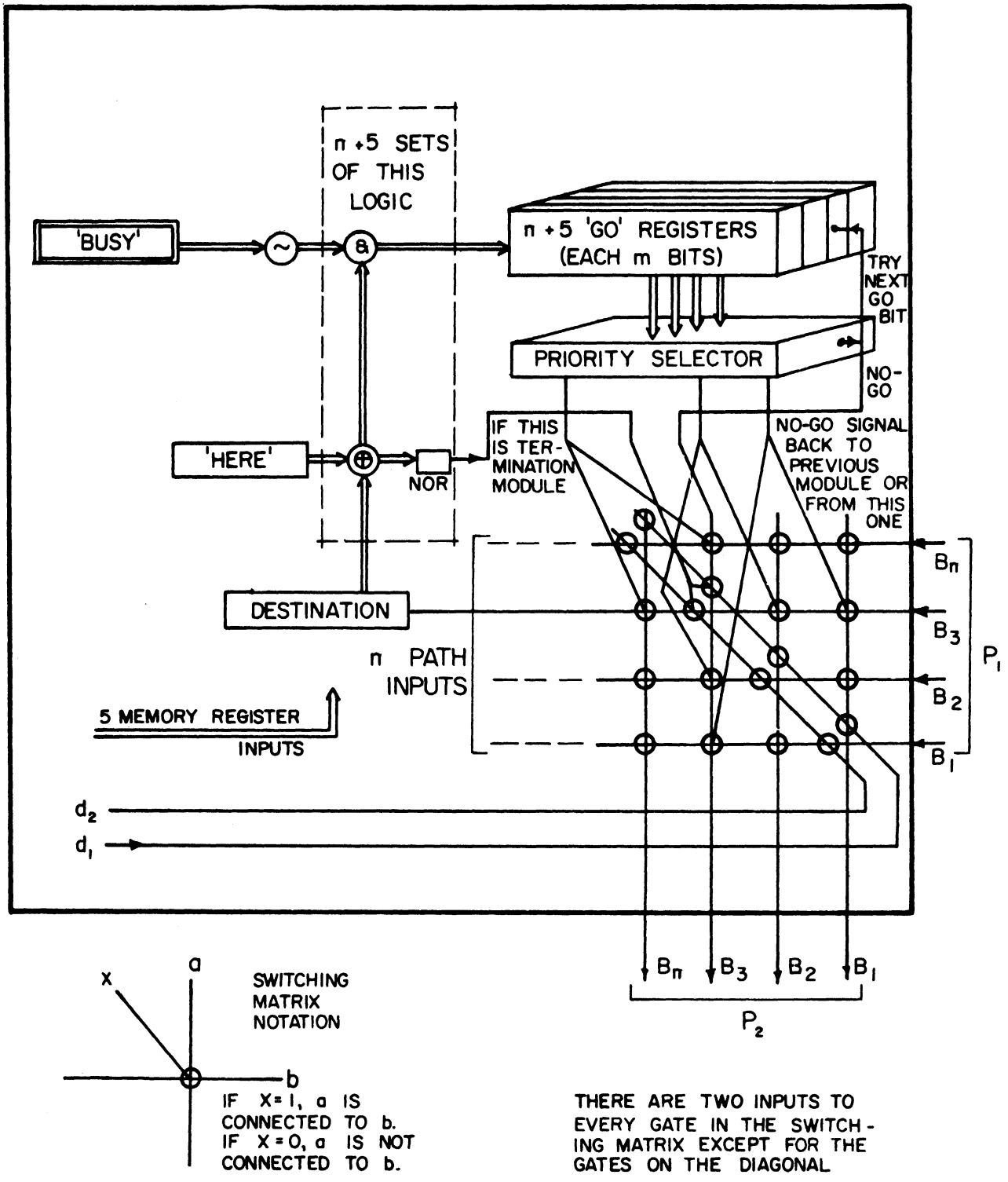


Fig. 66. Function and flow diagram for path-connecting circuitry.

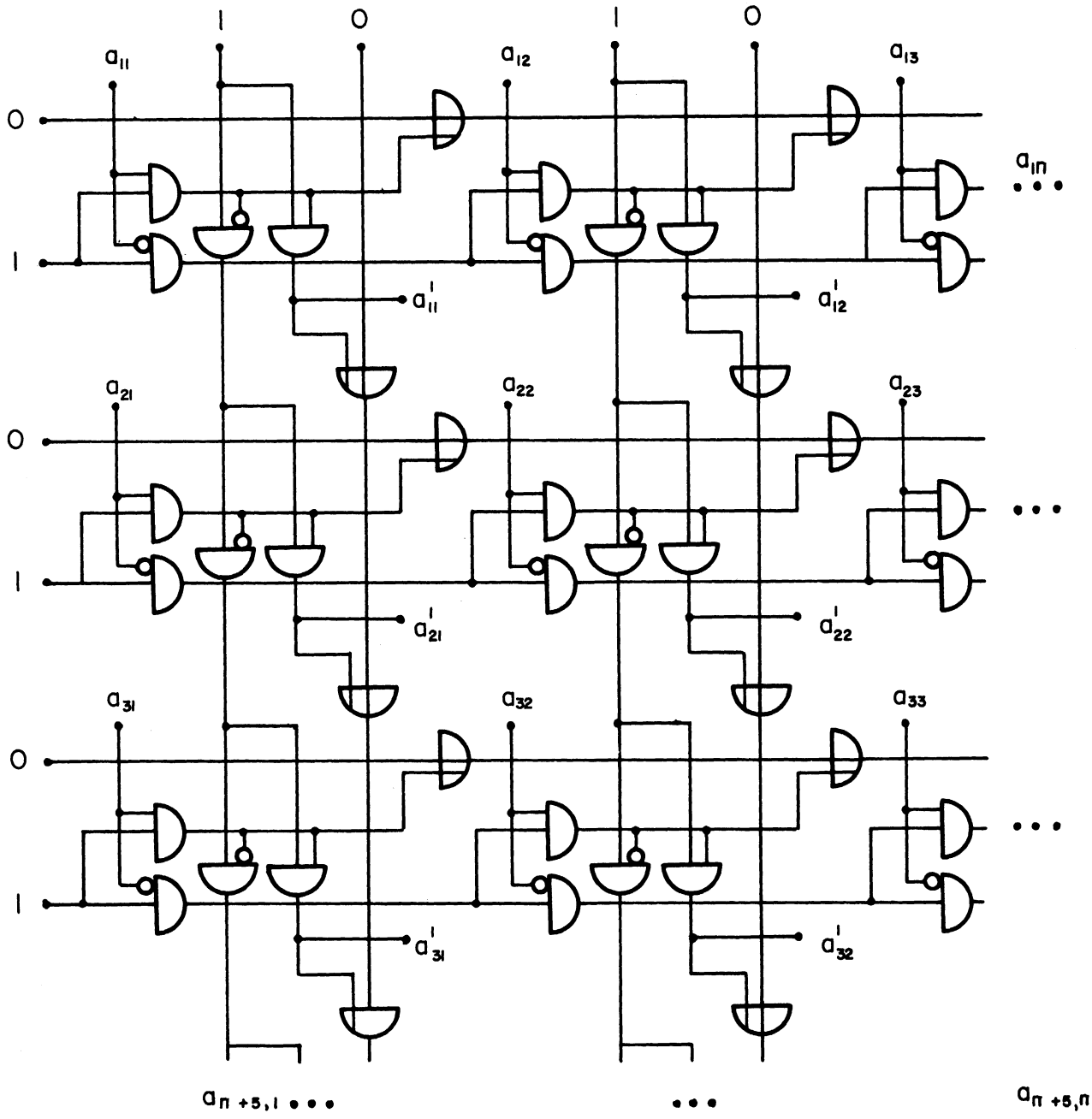


Fig. 67. Two-dimensional priority selector.

Step 1A — The module 0001 is initiating a path and therefore computes the contents of its GO register.

(HERE ⊕ DESTINATION) ~ (BUSY) ,→ GO.

is (0001 ⊕ 1010) ~ (0000) = 1011

Other modules could simultaneously be initiating or extending paths but for simplicity of explanation only one path is being considered.

The lowest 1 in the GO register of 0001 determines which segment will become a part of the path. This segment connects to module 0000 which is one step closer to the destination than 0001.

Step 1B — The BUSY bit for the B_1 segment is set in both 0001 and 0000. The destination is sent along the segment (1) to 0000.

The GO register of module 0000 is set to (0000 ⊕ 1010) ~ (0111) = 1000. The second and third BUSY bits had been previously set when the path coming in B_3 and going out B_2 was built.

The lowest 1 in the GO resister of 0000 specifies that the B_4 segment, 2 is to become a part of the path.

Step 2A — The 4th BUSY bit is set in 0000 and 1000 and the destination is sent from 0000 to 1000.

the GO register of module 1000 is set to (1000 ⊕ 1010) ~ (1010) = 0000.

Step 2B — Since the GO register is all zero a NO-GO signal is sent back along (2). The 4th BUSY bits in 1000 and 0000 are reset.

Upon receipt of the NO-GO signal the module 0000 sets the lowest 1 in its GO register to zero. (In this case making it all zero.)

Step 3A — Since the GO register is all zero a NO-GO signal is sent back along (1). The first BUSY bits of 0000 and 0001 are set to zero.

Upon receipt of the NO-GO signal, 0001 sets the lowest 1 in its GO register to zero.

Step 3B — The lowest 1 in the GO register of 0001 is now in position 2, thus a segment 3 is added to the path. The 2nd BUSY bits are set in 0001 and 0011 and the destination is sent from 0001 to

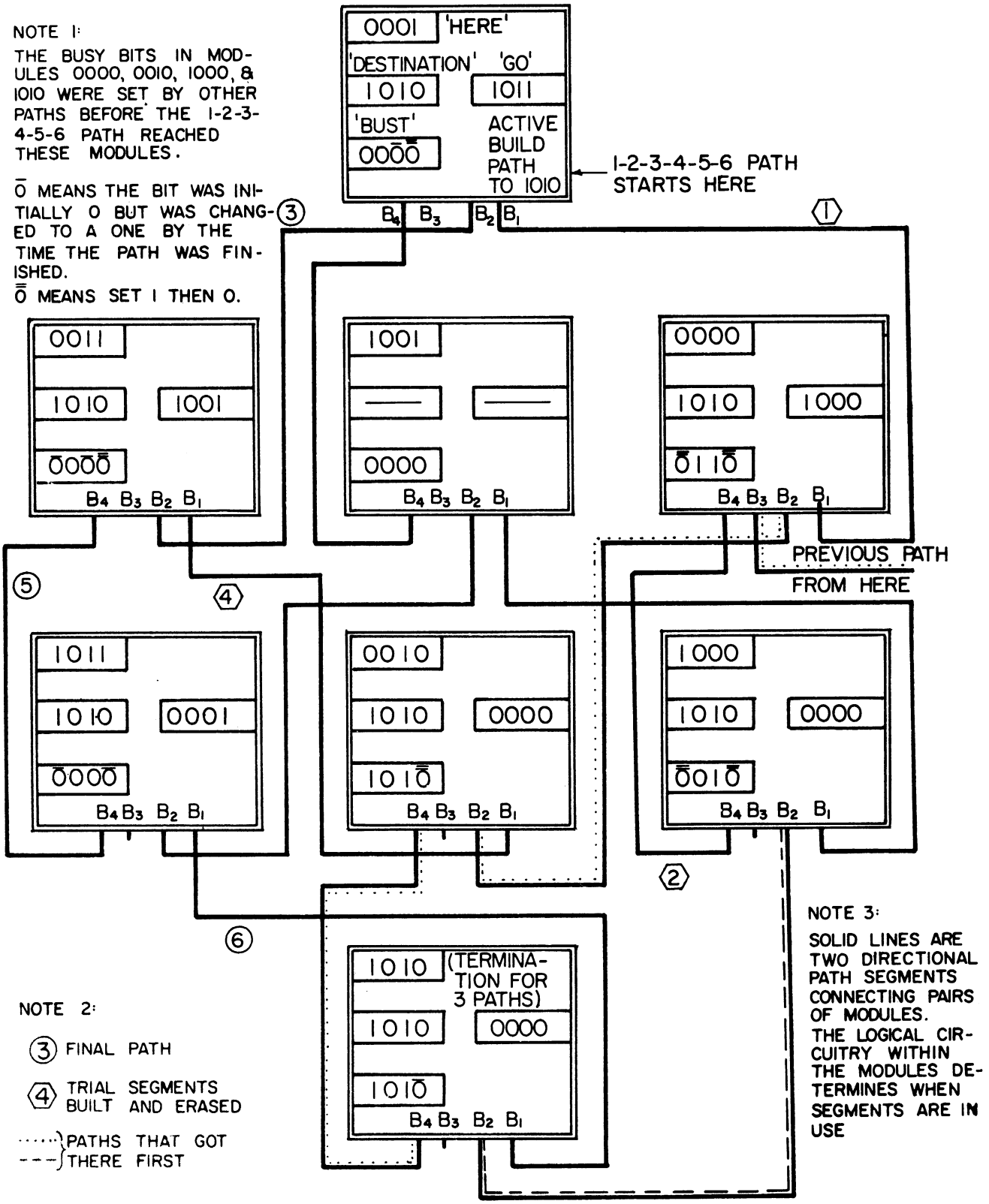


Fig. 68. Progression of a path connection.

0011.

The GO register of 0011 is set to (0011 ⊕ 1010)∧ ~ (0010) = 1001.

Step 4A — The segment ⟨4⟩ is added.

The GO register of 0010 is set to (0010 ⊕ 1010)∧ ~ (1011) = 0000.

Step 4B — A NO-GO signal is sent back along ⟨4⟩.

The lowest GO bit is set to 0 in 0011.

Step 5A — The segment (5) is added.

The GO register of 1011 is set to (1011 ⊕ 1010)∧ ~ (1000) = 0001.

Step 5B — The segment (6) is added.

The termination is detected since (1010 ⊕ 1010) = 0000. Execution using this path can now take place.

The logic and transmission properties could be designed to perform one phase per basic clock time. Thus the example given above would require 10 basic clock times to complete the path. Lookahead could be accomplished by having paths connecting by successors while the arithmetic operations of the predecessors are being performed. It seems that the average path-connecting time will require about the same time as an average arithmetic operation.

3.2.6 Conclusion

The basic question of the economics of an I.C.C. is: How much is fast computation worth? We have yet to hear a concrete answer to this question, and indeed there will probably never be a simple answer. The consensus seems to be that a computer twice as fast in every respect is not worth twice the

cost. To determine how much speed is worth, the reliability, type of problem, qualifications of the programmers and numerous other factors must be considered. For a computer such as being described here, there is one further factor to consider. That is: How parallel is highly parallel? There are examples of problems that could be done on this machine in 1/1000 the time required by a conventional computer built from the same components. There are other examples where this machine could barely cut the computation time in half. We have no accurate measure of the average parallelism possible in this machine. Based on our experience in considering a few problems, we estimate that on the average between 10 and 100 instructions could be executing simultaneously on a medium-sized computer. This is to be contrasted with our educated guess of a cost 10 to 100 times that of a conventional machine.

A medium-sized I.C.C. is certainly within engineering feasibility. Perhaps the first machine of this type should be designed for a user with much computation suitable for parallel processing, i.e., on problems involving matrices, solving systems of equations, inverting matrices, finding eigenvalues; or in other specific problems such as solving boundary value differential equations by the relaxation method. In these and some other problems, hundreds of calculations could be made simultaneously. By specifically choosing the command structure and size of the machine for a few specific problems, an economically competitive computer could be built today.

The rather powerful machine described in detail in this report is tailored to a need not yet fully developed. Until some good programmers and numerical analysts have such a machine in their hands, it is difficult to pre-

dict how much potential a computer of this type will have. We are optimistic that the iterative circuit computer organization is one of the methods that will enable computers to do much more computation in a given time.

3.3 HARDWARE REQUIREMENTS FOR MACHINE AS DESCRIBED

The path-connecting logic and registers require the following hardware in each module:

Quantity	Bits of Storage	Logical Elements	Description
n+5	n		'GO' storage memory
1	5(n+1)+4		storage 'BUSY' storage
1	n		
3(n+5)		n input logic	&, ⊕, NOR logic
n^2		switching logic	Switching matrix*
n(n+5)		one stage priority	priority circuit

For a 4096-module machine n would be 12. The number of bits of storage would be (204 + 69 + 12) 4096 = 1,167,360. (This is a few less than the number of storage bits in conventional memory of 32k 36-bit words). There would also need to be about another 1.6 million simple logical elements.

We estimate approximately 400 logical elements for the arithmetic unit in addition to 26 bits of storage for the operation control register.

Another 100 logical elements would be needed for miscellaneous module control. These would be for computing execution bits, signaling successors, and routing operands to the appropriate paths, etc.

Assuming the central timing and synchronization to be less than 10% of the machine, the total number of storage bits and logical elements would be less than five million.

* For serial two-way transmission, multiply by the number of bits to be sent in parallel.

3.4 MATRIX INVERSION PROGRAM FOR AN I.C.C.

INSTRUCTION LOCATION	OPERATION	ADDRESSES α,β,γ
*		
*		THE NXN MATRIX IN THE 'A' REGION IS
*		INVERTED BY A GAUSS-JORDAN METHOD.
*		THE INVERTED MATRIX REPLACES THE
*		ORIGINAL CONTENTS OF THE 'A' REGION
*		
ENTRY	PROCEED=	0,0,SET1'(1)
*		
*		FIRST EXECUTION STEP OF A THREE-EXECUTION
*		STEP LOOP THAT WILL BE PERFORMED N TIMES.
*		
SET1'(1)...SET1'(N)	INDADR	LA'(1),E(1),Q'(1)...LA'(N),E(N),Q'(N)
SET1'(N+1)	INDADR	EXIT
*		
E(1)...E(N)	LOAD	AKK,A(1,1),F(1)...AKK,A(N,N),F(N)
*		
LA'(1)...LA'(N)	INDADR	L(1,1)...L(N,1),...,L(1,N)...L(N,N)
L(1,1)...L(1,N)	LOAD	B'(1),A(1,1),M(1,1)...B'(1),A(1,N),M(N,1)
L(N,1)...L(N,N)	...	B'(N),A(N,1),M(1,N)...B'(N),A(N,N),M(N,N)
B'(1)...B'(N)	INDADR	AT(1,1)...AT(1,N),...,AT(N,1)...AT(N,N)
*		
Q'(1)...Q'(N)	INDADR	U(1,2)...U(1,N),,U(I,1)...U(I,I-1),U(I,I+1)...
	ETC	U(I,N),,U(N,1)...U(N,N-1)
U(1,1)...U(1,N)	DIVIDE	A(1,1),A(1,1)...A(1,N),A(1,1),T(1)
U(N,1)...U(N,N)	...	A(N,1),A(N,N)...A(N,N),A(N,N),T(N)
*		
*		SECOND EXECUTION STEP
*		
F(1)...F(N)	DIVIDE	A(1,1),AKK,G(1)...A(N,N),AKK,G(N)
*		
M(1,1)...M(1,N)	MULTIPLY	C'(1),A(1,1),S(1,1)...C'(N),A(1,N),S(1,N)
M(N,1)...M(N,N)	...	C'(1),A(N,1),S(N,1)...C'(N),A(N,N),S(N,N)
C'(1)...C'(N)	INDADR	AT(1,1)...AT(N,1),...,AT(1,N)...AT(N,N)
*		
T(1)...T(N)	LOAD	V'(1),ZERO,Y(1)...V'(N),ZERO,Y(N)
V'(1)...V'(N)	INDADR	A(2,1)...A(N,1),,A(1,I)...A(I-1,I),A(I+1,I)...
	ETC	A(N,I),,A(1,N)...A(N-1,N)

INSTRUCTION LOCATION	OPERATION	ADDRESSES α,β,γ
*		
*		
*		THIRD EXECUTION STEP
*		
G(1). .G(N)	DIVIDE	A(1,1),AKK,SET1'(1+1)...A(N,N),AKK,SET1'(N+1)
*		
S(1,1)...S(1,N)	SUBTRACT	A(1,1),AT(1,1)...A(N,1),AT(N,1)
S(N,1)...S(N,N)	...	A(N,1),AT(N,1)...A(N,N),AT(N,N)
*		
*		ON THE ITH PASS THROUGH THE LOOP
*		THE ITH ROW OF SUBTRACT INSTRUCTIONS
*		IS INHIBITED.
*		
Y(1)...Y(N)	INHIBIT	-,-,Z'(1)...-,-,Z'(N)
*		
Z'(1)...Z'(N)	INDADR	S(1,1)...S(1,N),...,S(N,1)...S(N,N)
*		
*		END OF COMPUTATION LOOP
*		
*		STORAGE ASSIGNMENT
*		
A(1,1)...A(1,N)	DATA	
A(N,1)...A(N,N)	...	
*		
AT(1,1)...AT(1,N)	TMPSTR	
AT(N,1)...AT(N,N)	...	
*		
AKK	TMPSTR	
*		
ZERO	DEC	0
*		
	END	

To illustrate the Gauss-Jordan method the following ALGOL program is given.

```
procedure   INVERT (N,A); value N; integer N; real array A;
comment     The N by N matrix in the A region is inverted by a Gauss-Jordan
            method. The inverted matrix replaces the original contents
            of the A region;
begin       integer I,J,K; real AKK,AIK;
    for     K:=0 step 1 until N do begin
            AKK:=A[K,K]; A[K,K]:=1.;
            for   J:=0 step 1 until N do A[K,J]:=A[K,J]/AKK;
            for   I:=0 step 1 until N do begin
                  if   I≠K then begin
                       AIK:=A[I,K]; A[I,K]:=0.;
                       for   J:=0 step 1 until N do
                       A[I,J]:=A[I,J]-AIK X A[K,J];
                  end   skip reduction of the Kth row;
            end   Kth column finished and matrix to left reduced;
    end     all N columns finished;
EXIT: end INVERT
```

A few additional comments should enable the interested reader to understand the details of the I.C.C. program.

First, the sequence of instructions which form the "loop" are:

ENTRY —(SET1'(1))→ E(1)→F(1)→G(1) —(SET1'(2))→ E(2)→F(2)→ ... →F(n)→ G(N) —(SET1'(N+1))→ EXIT. Thus, the number of execution steps is 3N+1.

Next, the instructions are the 3-address type described in Part 4. The first address is one operand and the location of the result; the second address is the second operand; and the third address is the next instruction to be executed. The last address being omitted implies it is not used, while a "—" implies an intermediate address is not used. An address with a prime, ', is indirect. Indirect address words may refer to more than one other word indirectly, thus enabling one instruction address to refer to many locations.

Further, the "*" at the front of a line implies that the line is a comment.

Finally, the ... notation has the usual meaning: to generate all intermediate subscripts.

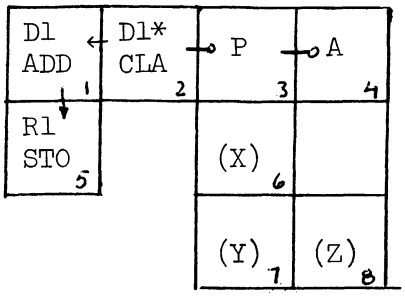
4. DETERMINATION OF ACCESSIBILITY

4.1 DESCRIPTION OF AN ITERATIVE CIRCUIT COMPUTER

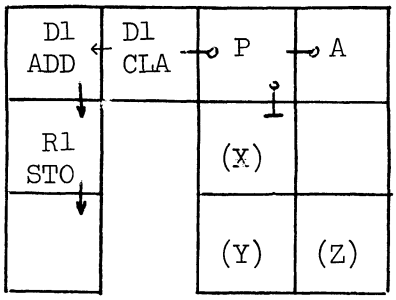
Only a brief summary of the I.C.C. concept is presented here. For additional details the reader is referred to References 8, 30 . Basically the I.C.C. consists of a large number of identical modules. The complexity of the module may be high or low. For example, each module could be considered to be a general-purpose digital computer. At the other extreme, the module could consist of the logic required for some logical primitive. Furthermore, the communication paths between modules are established in a uniform manner. In general every module may communicate directly only with the modules which are immediate neighbors. The number of neighbors is determined by the geometry of the I.C.C.

In the Holland I.C.C. concept, data are accessed and instructions are sequenced by the construction of paths to and from modules. A module which is excuting or interpreting an instruction is termed an "active module." The module from which a path originates is called a "P module." Any number of modules may be active simultaneously. Thus, within a given I.C.C. many different programs may be executed simultaneously. Since all modules are identical every module is required to have the capability for instruction interpretation and execution, data or instruction storage, and path interconnections between modules.

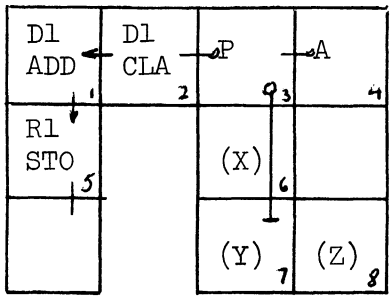
The nature of the Holland I.C.C. concept is shown by the example presented in Figure 69. In this example, the simple problem "Add X to Y and store in Z" is executed by an I.C.C. consisting of eight modules. The machine operation is divided into two phases: path building and execution. These two phases are not permitted to overlap in time.



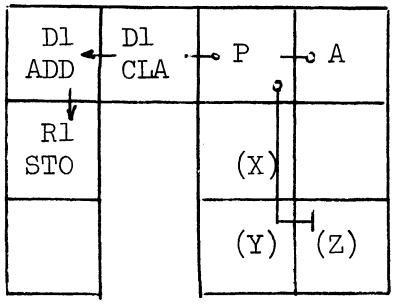
This is the program as initially stored in the computer. The program starts in the path-building phase with the module marked '*' active. The symbol ← indicates the next module to be activated. The symbol —o indicates the path from the active module to the accumulator (A). The circled numbers are for module reference only and are not part of the program.



The path is built down 1 (D1) from 'P' module under the control of module 2. The operation, clear and add (CLA), clears the accumulator of the 'A' module and adds the contents of module 6 to the accumulator.



Module 1 is the successor, denoted by →, of module 2. Module 1 becomes active when the CLA instruction of module number 2 is completed. Under the control of module 1, the path is extended down 1 (D1), and the contents of module 7 are added to the accumulator of the 'A' module. The successor is now module 5.



The path is extended (R1) right one to access module 8 under the control of module 5. The store operation causes the contents of the accumulator of the 'A' module to be placed in the storage register of module 8.

Figure 69. Simple I.C.C. Program

Note that computation of this program could be halted temporarily from storing the sum in module 8 if some other program had a path passing horizontally through modules 7 and 8. Although two paths can cross in a module, there is only one path segment (communication channel) permitted between any pair of modules. Thus if some segment is needed by two paths at the same time, one path must be deferred or altered. In the I.C.C. concept, the term accessibility refers to the ability of active modules to complete the path building phase. This criterion of path interference is used in the analysis of path building which follows.

4.2 EVALUATION TECHNIQUES

In general the evaluation of the effectiveness or the efficiency of a given machine design is difficult. Specific machine parameters which contribute to machine utility are known but the relative weights which should be assigned to each are usually impossible to determine. The difficulties in the evaluation of general-purpose machine are due in part to the fact that the class of solvable problems is not defined in a manner which facilitates a discussion of such things as efficiency of effectiveness.

In the absence of a general measure for machine utilization two specific types of studies are conducted in an attempt to evaluate a given machine design: (1) The machine's performance can be analyzed with respect to different types of problems. The analysis can be conducted by direct simulation or by means of analytical techniques. (2) Often it is possible to isolate basic characteristics of the machine design. Such basic characteristics frequently may be evaluated independently of problem characteristics or other machine characteristics.

Studies of the first type become meaningful only when a large number of different problems have been considered. In fact it is difficult to obtain sufficiently large samples except over a long period of actual computation and evaluation by machine users. Thus, when only small samples of actual computation experience are available, studies of the second type may be of great value to the machine designer.

Both types of studies are presently being conducted to determine the relative effectiveness of the Holland I.C.C. concept.

4.3 THE MATRIX INVERSION PROBLEM

The material in this section is a comparative study of three different machines applied to the general problem of matrix inversion. The first computer considered is the conventional computer containing a single processing unit. The second machine is a two-dimensional I.C.C. of the Holland type. The preliminary studies indicated specific difficulties due to path interference. Thus the third machine considered is an I.C.C. of the Holland type constructed on an N-cube. This type of geometry greatly facilitates path building since each module has N immediate neighbors. The interconnections required between neighbors in an N-cube geometry are obtained by N connection planes. Each plane has a uniform connection pattern.

There exist many different numerical methods for matrix inversion. The degree of local control required is different for each method. The Gauss-Jordan method was chosen since this appears to permit the greatest degree of parallel computation.

The comparative results are given in Table 2. Only the relative order of

TABLE 2

Comparison of Three Computer Organizations
For the Matrix Inversion Problem (M x M Matrix)

	Conventional Computer	2-Dimensional I.C.C.	N-Cube I.C.C.
Number of data words	M^2	M^2	M^2
Number of instruction words and temporary storage	150	5 to $9M^2$	$7M^2$
Total words	$M^2 + 150$	6 to $10M^2$	$8M^2$
Minimum possible length of the program	$5M^3$	$3M^2$	$3M + 1$
Maximum number of instructions executed simultaneously	1	$5M$	$2M^2$
Total words times length of program	$5M^3$	18 to $30M^4$	$24M^3$

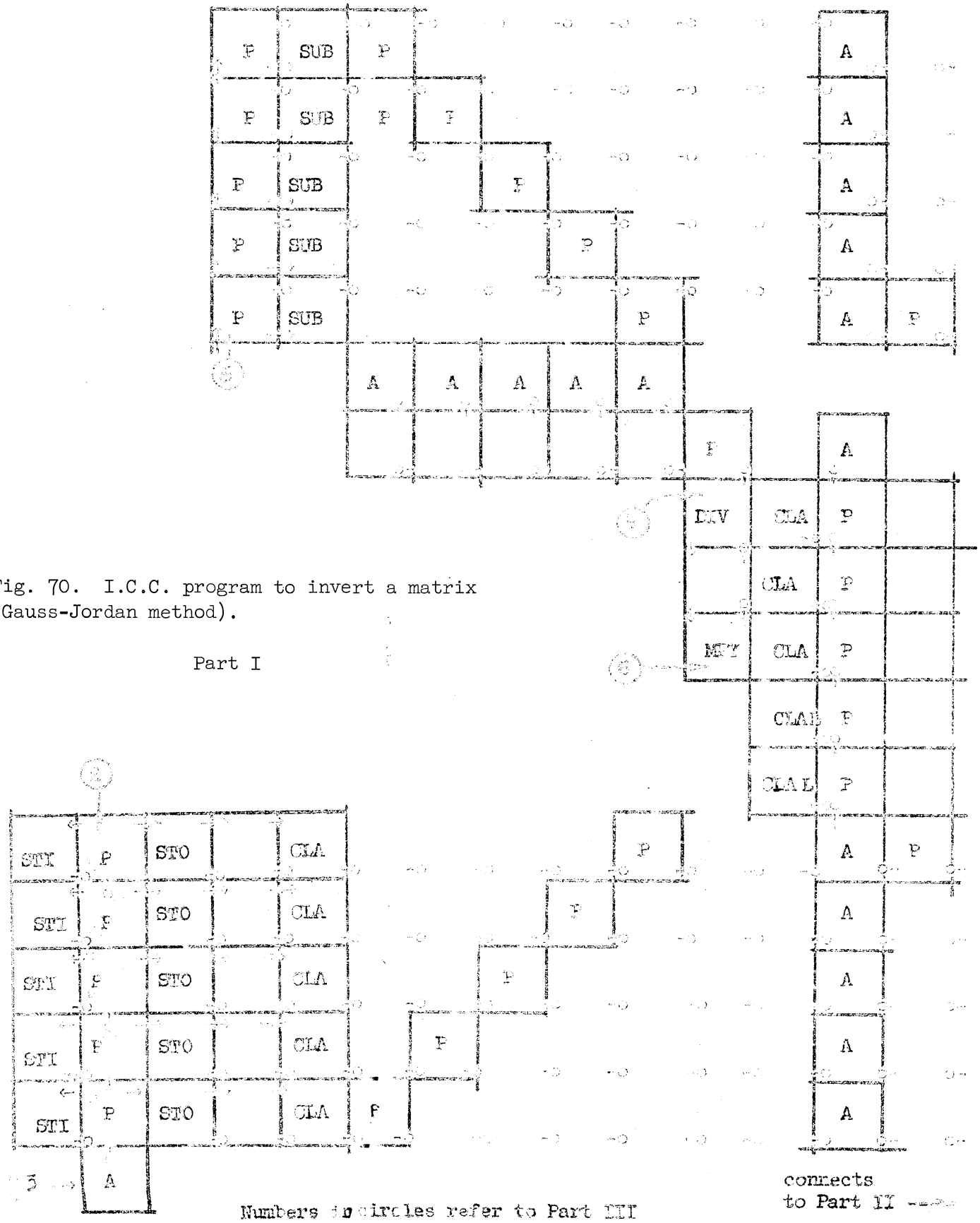


Fig. 70. I.C.C. program to invert a matrix (Gauss-Jordan method).

Part I

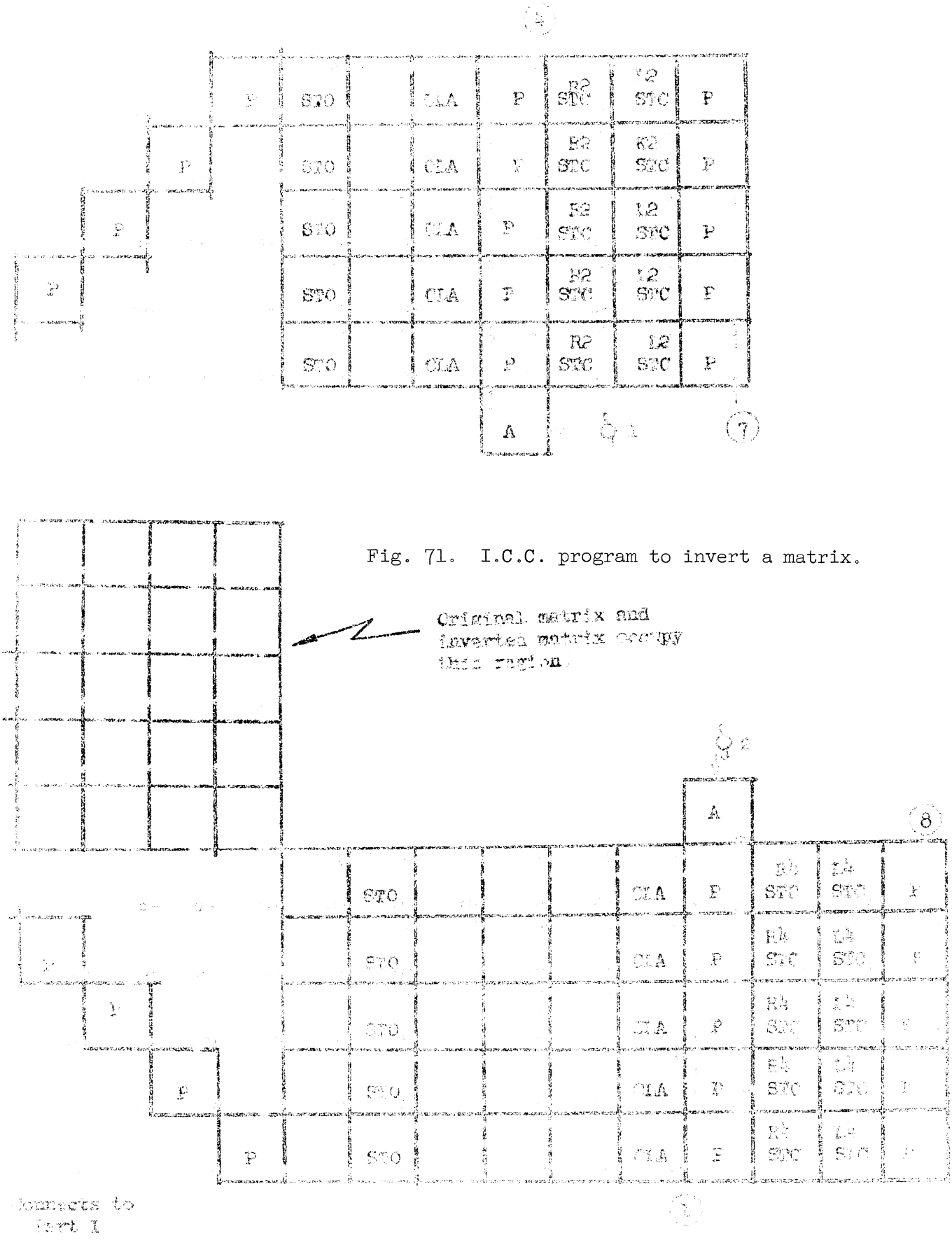


Fig. 71. I.C.C. program to invert a matrix.

the magnitude of these figures is significant. Observe that an order of magnitude decrease in the length of the program exists between the conventional machine and the two-dimensional I.C.C. and the N-cube I.C.C. At the same time, the number of instructions executed in parallel changes by a factor of M between the conventional machine and the two-dimensional I.C.C., and by another factor of M between the two-dimensional I.C.C. and the N-cube I.C.C. The two-dimensional I.C.C. performance is limited because of path building conflicts. Often it is impossible to access data, and instruction cycles are wasted until the desired data can be accessed. This situation occurs less frequently in the N-cube I.C.C. Path building conflicts could be removed by the multiple storage of data but this is an expensive solution in the I.C.C. structure. Frequently additional instructions must be used in the two-dimensional I.C.C. to alleviate accessibility problems. Because of this, the two-dimensional I.C.C. programs may require more modules than the N-cube I.C.C. In both I.C.C. structures, a work requires a module. Thus, a good estimate of the number of modules required to perform the inversion of an M x M matrix is $8M^2$. The matrix inversion program for a 5 x 5 matrix using a two-dimensional I.C.C. is shown in Figures 70, 71. In this specific case, $\approx$ 215 modules are used. The I.C.C. has no unprogrammed central control between modules. The central control program required for the matrix inversion is shown in Figure 72. Notice that the I.C.C. organization generally requires more instructions and temporary data locations. This is due directly to the increased degree of parallel computation. Ordinarily, modules active at the same time period are not executing identical instructions. Therefore different instruction sequences are required for different module as-

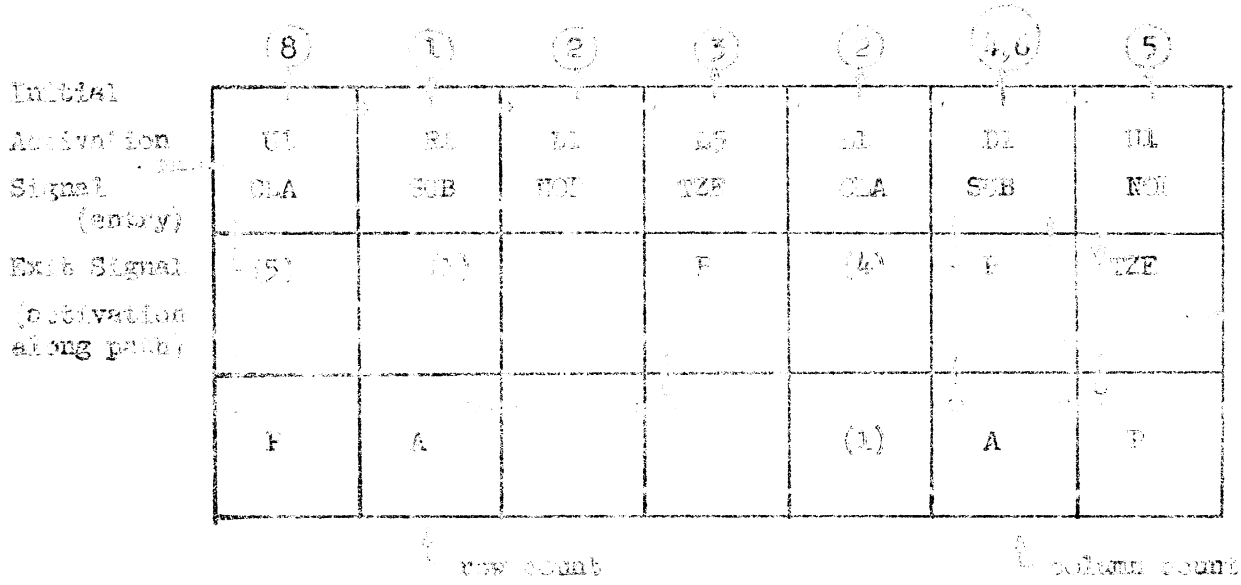


Fig. 72. Master control timing program for a 5x5 matrix inversion.

semblies. The minimum possible program length is 3M + 1. This limitation is due to the nature of the Gauss-Jordan algorithm and cannot be lowered. This program length can be achieved by the N-cube machine but is difficult to achieve with the two-dimensional machine because of the limitation in the accessibility of data.

The results of other programming studies are similar to those presented here for the matrix inversion problem. Specific types of problems are certainly non-ideal for an I.C.C. For example, the finite difference solution for Laplace's equation in two dimensions requires 17 modules per node if each node is locally controlled. This is obviously inefficient since each node can be controlled from a global control program. However, this approach leads to path building conflicts.

The study of the performance of specific I.C.C.'s for specific types of problems has shown that path building is a major factor in the design and programming of an I.C.C. A statistical approach has been pursued in an attempt to evaluate the path building capability inherent in different I.C.C. designs.

4.4 ANALYSIS OF THE PATH BUILDING PROBLEM

In this section two intuitive measures of path building are considered and comparisons of the two-dimensional and the N-cube I.C.C. are made. Finally, the results are presented of a statistical analysis and simulation of the path connection problem.

One estimate of the ability to build a path from module A to module A' is the number of possible paths there are from A to A'. Only paths that are of minimal length (non-regressive) are considered.

In a two-dimensional machine, a path between two modules consisting of d_1 horizontal path segments and d_2 vertical path segments has a path length equal to $d_1 + d_2$. It makes no difference in which order the horizontal and vertical segments are chosen. No path can connect the modules with less than $d_1 + d_2$ segments; therefore $d_1 + d_2$ is the minimal length. The number of minimal, but not necessarily independent, paths between two modules is given by $\frac{(d_1 + d_2)!}{d_1!\ d_2!}$.

Notice that d_1 is the number of horizontal path segments which is one less than the number of modules in a horizontal row. These numbers grow very fast. For a 5 x 5 array of modules there are $\binom{4+4}{4} = 70$ connecting paths between opposite corners. For a 10 x 10 array, there are 48,620 and for a 20 x 20 array there are about 10^{10}. This is a very large number of potential paths but they are not disjoint; there are only two disjoint paths.

The path structure of an N-cube I.C.C. is most easily visualized by considering each module as an N-bit number. There are the same number of modules and N-bit numbers, i.e., 2^N. Again, only nonregressive paths are considered. For these, it is possible to state reasonable path connecting algorithms in the N-cube machine. Further, with only minimal paths definite limits for maximum data access along a path can be determined. A minimal path is generated by choosing path segments to other modules such that the Hamming distance between the current module and the termination module is reduced by one each step.

Thus, we can easily note two facts about paths. The maximum path length is N since two N-bit numbers can differ at most in N positions. And, the number of ways of connecting two modules at distance k is k!, i.e., all permutations

in the order of reducing the Hamming distance by one for k steps. The k! possible paths between two modules are different but not disjoint and involve 2^k modules.[5]

Intuitively, the more paths that can be built in a given area, the better. In other words, there is less limitation by accessibility. Therefore, the number of possible paths divided by the area used by the paths is a possible figure of merit of accessibility for a particular I.C.C. configuration. Thus for a two-dimensional I.C.C. this figure of merit is given as

$$\frac{\frac{(d_1 + d_2)!}{d_1!\, d_2!}}{(d_1 + 1)(d_2 + 1)}$$

and for the N-cube I.C.C. $k!/2^K$.

If an I.C.C. consisting of 4096 modules is considered, the number of ways of connecting two modules at a distance k = 6 is 20 for the two-dimensional I.C.C. and 720 for the N-cube machine (N = 12). The accessibility figure of merit is 1.25 and 11.25 for the respective machines.

Another measure of accessibility is the average distance between module pairs. This shall be called the module distance. A low average path distance is desirable. The average path distance and the distribution of path distances can be determined. Note that W(p), the number of paths of length p for a two-dimensional I.C.C. with D modules on a side, is given by:

$$W(p) = 2 \sum_{j=0}^{p-1} \max(0,D-p+j)\, \max(0,D-j).$$

W(p) is the number of distinct pairs of modules at Manhattan distance, p, and does not refer to the number of ways of connecting a pair of modules. The range of p is from 1 to 2(D-1),

The average module distance A is given by:

$$A = \frac{\sum_{p=1}^{2(D-1)} pW(p)}{\sum_{p=1}^{2(D-1)} W(p)}$$

For computation purposes

$$\sum_{p=1}^{2(D-1)} W(p)$$

can be expressed as the total number of possible paths T in the machine. Thus

$$T = 2 \sum_{i=0}^{D-1} \sum_{j=1}^{D-1} (D-i)(D-j) = \frac{D^2(D^2-1)}{2}.$$

Also, the numerator can be written as

$$S = 2 \sum_{i=0}^{D-1} \sum_{j=1}^{D-1} (D-i)(D-j)(i+j).$$

The resultant expression for the average module distance simplifies to:

$$A = \frac{S}{T} = \frac{2D}{3}$$

The average module distance in a two-dimensional I.C.C. is proportional to the number of modules along one side of the machine or to the square root of the total number of modules.

For an N-cube I.C.C. with 2^N modules, the range of p is from 1 to N. The average module distance, A, can now be computed from:

$$A = \frac{\sum_{p=1}^{N} W(p)p}{\sum_{p=1}^{N} W(p)}$$

W(p), the number of non-regressive paths of length p, is given as $W(p) = \binom{N}{p} 2^{N-1}$.

For computational purposes $\sum_{p=1}^{N} W(p)$ can be expressed as the total number of possible paths, T, in the machine.

$$T = \frac{2^N(2^N-1)}{2} = 2^{N-1}(2^N-1)$$

N is the dimension of the N-cube. The numerator can be simplified to

$$S = \sum_{p=1}^{N} pW(p) = \sum_{p=1}^{N} p\binom{N}{p}2^{N-1} = 2^{N-1}N2^{N-1}$$

Thus the average module distance

$$A = \frac{S}{T} \approx \frac{N}{2}$$

Here the average module distance is proportional to the $\log_2$ of the number of modules in the machine.

The average module distance for the two-dimensional I.C.C. consisting of 4096 modules is 43, while for the N-cube I.C.C. (N=12) the average module distance is 6. The maximum module distance is then 126 and 12, respectively.

General results for the average module distance have been calculated and the results are presented in Figures 73 and 74.

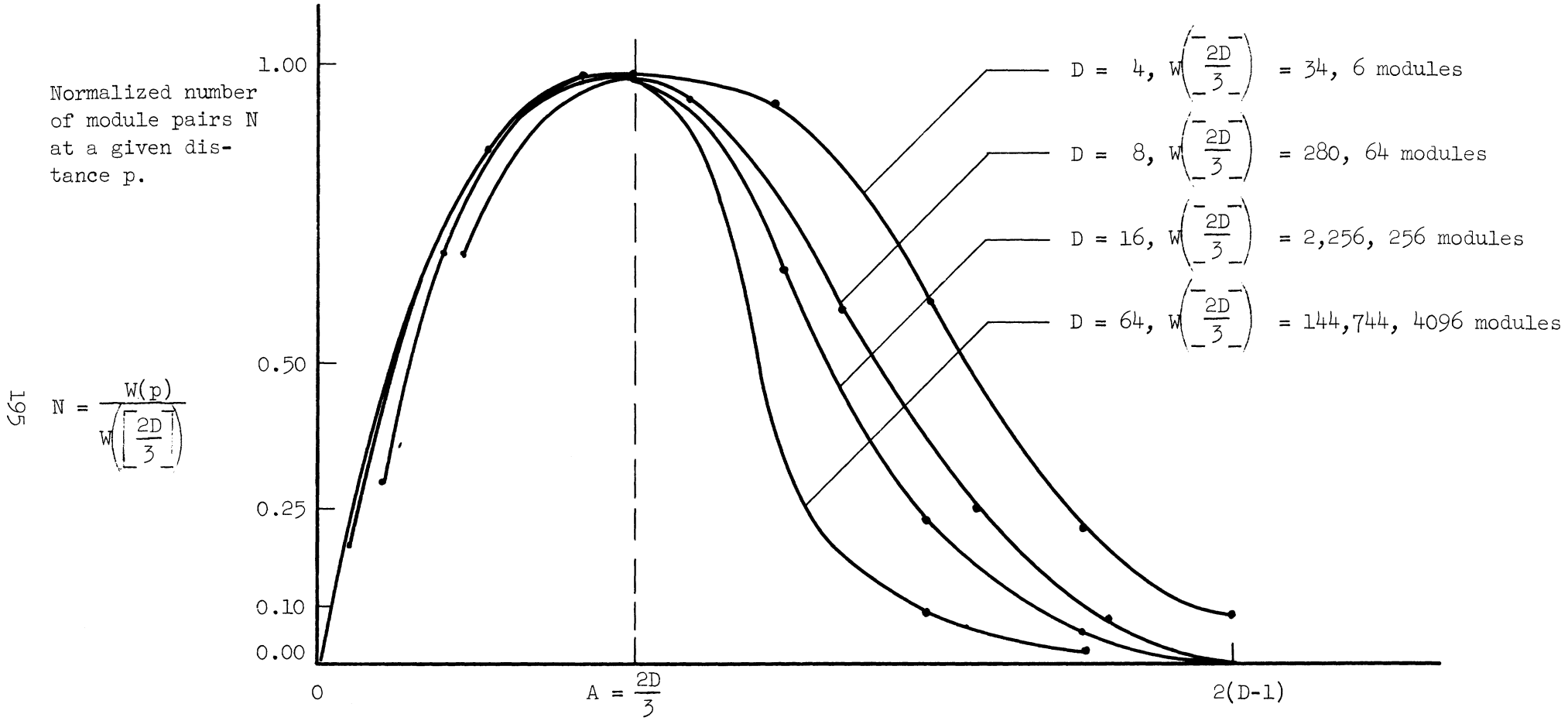
One interesting aspect is that the maximum number of module pairs are separated by the average module distance. Note the values exist only for integer values of distance p greater than one. The points are connected by a smooth curve to give a better presentation. Interesting comparisons can be made between the graphs for the two-dimensional I.C.C. and the N-cube I.C.C. since the total number of modules is equal to D^2 and 2^N respectively. $D^2 = 2^N$ for the values of D and N, chosen for respective curves in Figures 73 and 74.

We now wish to get an estimate of the number of instructions that could be executed simultaneously. Since each module which holds an active instruction is capable of executing the instructions, the limiting factor is the accessibility of the operands. In order to get an estimate of the accessibility, two assumptions are made: (1) instructions and data are randomly located, and (2) all path lengths are equal to the average module distance. In general, programmers should be able to do much better than the statistical estimates by a planned placement of data and instructions to keep operand paths short, and by dispersing simultaneously active instructions as much as possible.

For the two-dimensional machine the average module distance is 2D/3. If

DISTRIBUTION OF MODULE DISTANCE FOR THE 2-DIMENSIONAL I.C.C.

(Normalized graph)

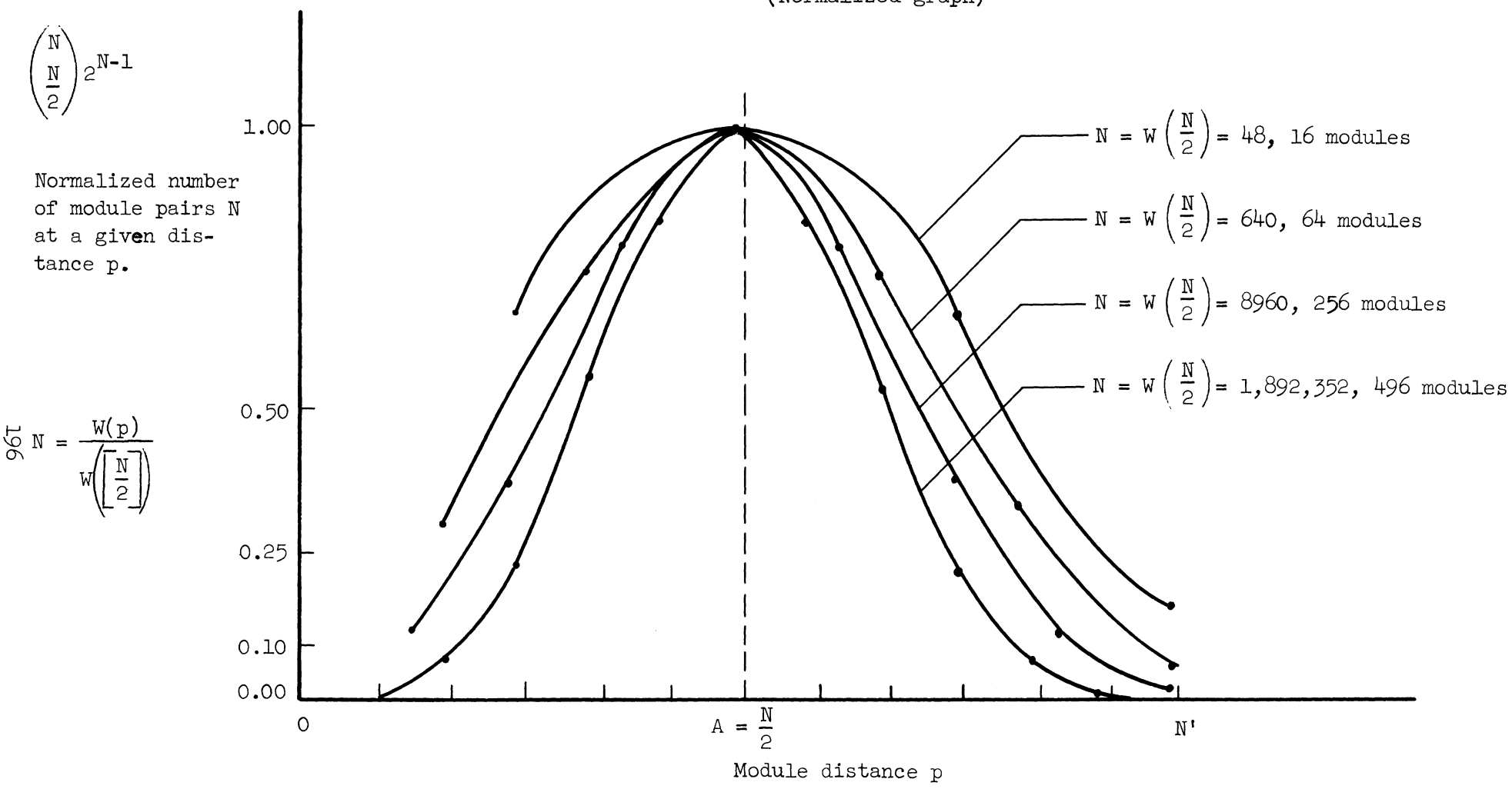


(Manhattan distance between modules)

Figure 73

DISTRIBUTION OF MODULE DISTANCE FOR THE N-CUBE I.C.C.

(Normalized gráph)



(Hamming distance between modules)

Figure 74

i instructions are executed simultaneously, then i.2D/3 path segments are active. The probability that a given segment is used is

$$\frac{\text{segments used by i paths}}{\text{total number segments}} = \frac{i.2D/3}{2D(D-1)} = \frac{i}{3(D-1)}$$

An approximation is made. It is assumed that connectedness of segments can be ignored and only the number of segments are considered. The probability that a given segment is available for the i+1 path is:

$$\left(1 - \frac{i}{3(D-1)}\right)$$

The probability that a given set of 2D/3 segments is available is:

$$\left(1 - \frac{i}{3(D-1)}\right)^{2D/3}$$

Finally, if there were k independent ways of building the $i+1^{st}$ path, the probability that it could be built is:

$$p_{i+1} = \left(1-(1-i/3(D-1))^{2D/3}\right)^{k}$$

k is approximated by

$$k = \frac{\sum_{i+j=A} 2(m(m+1))+(N-m)(2m+1)}{A(A+1)}$$

where: m = minimum (i,j)
N = maximum (i,j)
i and j are positive integers
A = [2D/3]

There are two statistical measures of accessibility which use p_{i+1}. E is the expected number of paths of average length that can be built; and H the number of average length paths that can be built with a probability of 1/2 of building all paths. These are computed as follows:

$$E = \sum_{i=0}^{\infty} (i+1)p_{i+1} = \sum_{i+1}^{3(D-1)} (i+1)\left(1-\left(1-(1-i/3(D-1))\right)^{Ak}\right)$$

$$H = \max r \text{ such that } p_1 \cdot p_2 \ldots p_r \geq 1/2, \text{ i.e.;}$$

$$\prod_{i=0}^{r}\left(1-\left(1-(1-i/3(D-1))\right)^{Ak}\right) \geq 1/2$$

The significance of these measures will be discussed after a similar development has been presented for an N-dimensional I.C.C.

For the N-cube I.C.C. the average path length is N/2. The total number of path segments in the machine is needed to compute p_i, the probability that the i^{th} path can be built successfully. This is

$$\sum_{i=1}^{N}\binom{N}{i}i = N.2^{N-1}$$

Assume that i average length paths have been built. This requires i·N/2 segments. The probability of a given segment being used is:

$$\frac{\text{used segments}}{\text{total segments}} = \frac{i \cdot N/2}{N2^{N-1}} = \frac{i}{2^N}$$

The probability of a given segment being available is

$$1 - \frac{i}{2^N}$$

The distribution of segments is assumed to be random; thus the probability of N/2 segments being available is

$$\left(1 - \frac{i}{2^N}\right)^{N/2}$$

If there are k independent ways of building the i + 1st path, then the probability that it could be built is:

$$P_{i+1} = 1-1-(1-i/2^N)^{N/2 \cdot k}$$

The number of independent ways of building the i+1 path is approximately

$$\frac{\sum_{p=1}^{N/2} \binom{N}{2}p}{N/2_{+1}} = \frac{2^{N/2_{+1}}}{N/2_{+1}}$$

The expected value, E, is a statistic which is an estimate of the mean number paths simultaneously connected in an infinite number of experiments. Each experiment consists of selecting random pairs of coordinates at Hamming distance N/2, then attempting to connect the paths. The expected value is

$$E = \sum_{i=0}^{2^N} (i+1)P_{i+1} = \sum_{i=0}^{2^N} (i+1)\left(1-\left(1-(1-i/2^N)^{N/2}\right)^k\right)$$

A more stringent requirement, that there is a probability of 1/2 that all H paths of average length could be built, is given by

$$H = \max r \text{ such that } P_1 \cdot P_2, \ldots P_r \geq 1/2, \text{ i.e.,}$$

$$\prod_{i=0}^{r} \left(1-\left(1-(1-i/2^N)^{N/2}\right)^k\right) \geq 1/2$$

Where the probability of building both the first and second is $P_1 \cdot P_2$ etc. until the probability of building all r + 1 paths becomes less than 1/2.

Simulation of the accessibility problem was performed to supplement the statistical analysis. This was desirable because of the approximations used in the statistical analysis. In particular, the approximation on the number of independent paths k is critical to the analysis. For comparison, a program was developed to be run on a 7090 which simulated the path connection (1) in an N-cube I.C.C., and (2) in a two-dimensional I.C.C. The program represented each module by a storage location and used the bits of the words to keep track of paths previously connected and being connected.

The simulation results are more accurate than the calculations obtained from the analysis. The only difference between the simulation and actual running conditions is the use of random starting and ending points of the paths.

The statistics for the distributed path length case were considered too complicated for analytical study. However, the distributed path length situation was not difficult to simulate. The distributed path length situation is a closer approximation to the actual I.C.C.'s operating conditions. For each path built, a random starting location was generated; then, either an average length was used or a length was determined by a weighted random selection from the distribution of lengths. Bits were then randomly inverted in the starting

location until an end location, the proper distance from the starting location, was generated. Each simulation started with no paths in the machine; the statistics were compiled both on the basis of the number of paths built (until the first path that could not be built was encountered) and on the basis of the number of paths built allowing for many paths to be unsuccessful.

The results obtained from the simulation and the analytical studies are presented in Table 3. The estimates for the maximum number of active modules are obtained by dividing the total number of path segments by the average path length.

The results from the simulation using distributed paths yield slightly fewer connections when compared to the simulation results using average path lengths, as would be expected. However, the percentage difference is small. This verifies that the assumptions used in the definition of an average path length are valid, so that the more complicated distributed path case need not be used in future investigations except for occasional verification.

The estimate of the probability of successfully connecting the I + 1st path, P_{i+1}, was used in the analytic derivation of E and H. This estimate for P_{i+1} considers the path segments of all paths in the machine as well as the I + 1st path to be randomly scattered, i.e., the connectiveness of a path is neglected. To compensate the analytic expression for E, a factor would have to be introduced to make the probability higher that a segment is available if all its neighbors are also available, and lower according to the number of neighbors not available. The simulation runs produced statistics which accounted for connectedness, and are thus more accurate. The simulation results indicate that connectedness does not seriously affect H, the number of paths built before the

TABLE 3

ANALYTIC AND SIMULATION RESULTS

ASSUMING ALL PATHS TO BE AVERAGE LENGTH								USING DISTRIBUTED PATH LENGTHS				
N	H_A*	H_S**	Std. Dev. of H_S	E_A	E_S	Std. Dev. of H_S	Maximum No. of Active Modules	H_S	Std. Dev. of E_S	E_S	Std. Dev. of E_S	T
					N-CUBE I.C.C.							
4	6	5	2	34	14	.6	16	4	2	15	1	32
6	19	16	5	506	51	2	64	13	5	49	5	192
8	69	66	21	7,925	175	4	512	55	23	173	6	1094
12	1151	986	158	2,022,879	2368	23	4096	549	160	2117	62	24576
					2-DIM. I.C.C.							
4	4	3	1	11	16	.5	12	4	2	15	2	24
8	5	5	2	20	14	.8	24	6	3	24	3	112
16	6	8	3	39	29	1.5	48	8	3	38	4	480
64	12	15	9	114	93	--	192	19	3	118	--	8064

*The subscript A is for analytic results.

**The subscript S is for simulation results.

first unsuccessful try. The simulation statistics, H_S, are from 5 to 10% lower than H_A obtained by analysis.

On the other hand, E_S disagreed by as much as a factor of 1000 with the corresponding E_A found analytically. Notice that E_A, (col. 5, Table 3) is even larger than the maximum possible number of active modules (col. 8, Table 3). The disagreement is primarily due to neglecting connectedness and a better estimate would be, heuristically, about 1/2 of the maximum possible number of active modules. For this case, the simulation is accurate, but the analysis must be refined.

4.5 CONCLUSIONS

The simulations and the analysis results of the path building problem are consistent with the results obtained from the study of the matrix inversion program for the I.C.C. structure. The relative decrease in path conflicts accounts for the superiority of the N-cube I.C.C. over the two-dimensional I.C.C. for the matrix inversion problem. The limiting factor in parallel computation, at least on matrix problems, is accessibility of operands. The results show conclusively that path building is a limiting factor in the performance of an I.C.C.

Attempts to find other configurations for path structure which could do better than the N-cube have not been successful. Generally, the amount of hardware increases exponentially as the accessibility increases. In the upper limit every module is able to access every other module directly by a unit length path segment. This is approaching the point where every module is a conventional computer and thus far too costly. The N-cube I.C.C. requires over twice as

much circuitry as the two-dimensional I.C.C., but the increased accessibility justifies the increase on problems such as matrix inversion and others where the advantage of highly parallel computation can be used.

5. RELIABILITY IN ITERATIVE MACHINES

5.1. REDUNDANCY AT THE MODULE LEVEL

The reliability problem in an iterative structure machine can be thought of as composed of two different solutions:

a) Each module has some computational capability, but the failure of a module does not necessarily influence the behavior of the rest of the machine, if there exists some way of checking and marking those modules which are not in working order. It is still desirable, of course, to have the reliability of each module as high as possible.

b) The functional redundancy of the machine, and its ability to execute more than one program at a time, allows the introduction of a checking program, whose mission is to periodically conduct tests on the modules and to indicate their condition by means of tags on the modules themselves.

In this section we are concerned with the first problem. It is very similar to the reliability problems in standard machines, with the introduction of a few particular characteristics.

Since the whole concept of an iterative machine depends on the availability of components and fabrication methods capable of producing large numbers of identical modules with a low unit price, it is almost mandatory to assume that the technology employed will be some variation of the vacuum decomposition techniques.

Also, since the module is here the basic unit, it is imperative to keep

it as simple as possible, and therefore, any redundancy scheme with a high penalty ratio in number of components is not suitable.

It is usual to compare the penalty in equipment for a given redundancy scheme by referring to a redundancy ratio defined as the ratio between the number of components in the redundant circuit to the number of components in the basic original circuit.

While this gives an idea of how much the complexity of the circuit will increase, it is completely misleading to think of this ratio as an indicator of relative cost. The relative cost is a weak function of complexity or number of components and depends strongly on the technology used to manufacture the components and also on the technique used to put together the individual components in stages and then the stages in units.

This fact is clearly shown in the following actual case. In an optimal design, two-out-of-three majority voting requires triplication of the logically active networks, plus the addition of triplicate voters, plus some additional complexity in the wiring. Roughly, this is a six-fold increase in complexity.[17]

If micrologic techniques are employed, as described in Ref. 19, the majority voters can be obtained by the simple addition of only four resistors to the original circuit.

Furthermore, even the triplication of flip-flops doesn't produce a corresponding increase in cost. The influence of the increased complexity in wiring is negligible. Thus, we see that the technological aspect influences the cost in two ways:

a) The fabrication and assembly procedures can make the cost very

lightly dependent on the complexity of the circuit.

b) The type of logic used can allow the introduction of majority voting in a very simple and economical way.

As in most physical systems, there exists in information processing and transmitting networks a natural balance between performance and cost. These two words are taken here in a generalized meaning, but no matter what parameter is used to measure them, it will be found that an increase in one will produce a correspondent increase in the other. The ratio performance/cost will not be a constant, and at least in computing systems, it seems to be a monotonically increasing function of complexity or size. The opposite is true for transmission or contact networks, in which very soon a point of diminishing returns is reached.

In summary, there seems to be an "ethereal" nondefinable constant that warns us to expect to pay more or to find restrictions or degraded performance in some part of the system when improvements are made in some other aspect.

While it is true that one cannot get anything for free, sometimes a bargain can be found. This is especially easy where the system has one or more characteristics that are not important or relevant to our problem. The designer's ability should then be dedicated to locating these more or less elusive areas containing parameters onto which he can "discharge" the inevitable increase that will follow as a consequence of the improvement forced on the relevant or useful characteristics.

In reliability problems this is especially difficult, since there are basically only two factors to be traded for increased reliability: time and complex-

ity. Inevitably, an increase in reliability will bring about an increase in either the time needed to perform the operation or in the amount of hardware needed.

Since we must be resigned to pay one price or the other, one can proceed to a finer inspection of the factors and determine in what ways time or complexity can be used to introduce redundance.

In general, time is traded for reliability when the following methods are used: repetition of message or error correcting codes. It is true that some time will be lost too when methods like majority voting are used, since the number of stages through which the information has to pass is increased, but this is a second-order effect, introducing only timing problems that are easily solved. Furthermore, the increased time delay is a consequence of the increase in complexity, and not the original cause or factor which we chose to produce better reliability. The rest of the methods produce an increase in the amount of hardware, that is, in components plus wiring and they differ only in the levels at which they are applied, in the size of the "boxes" which are replicated, or in the introduction of new "boxes" whose function is strange to the main purpose of the network.

It is therefore, timely to try to extricate the basic factors in these approaches and to present them in the form of a classification.

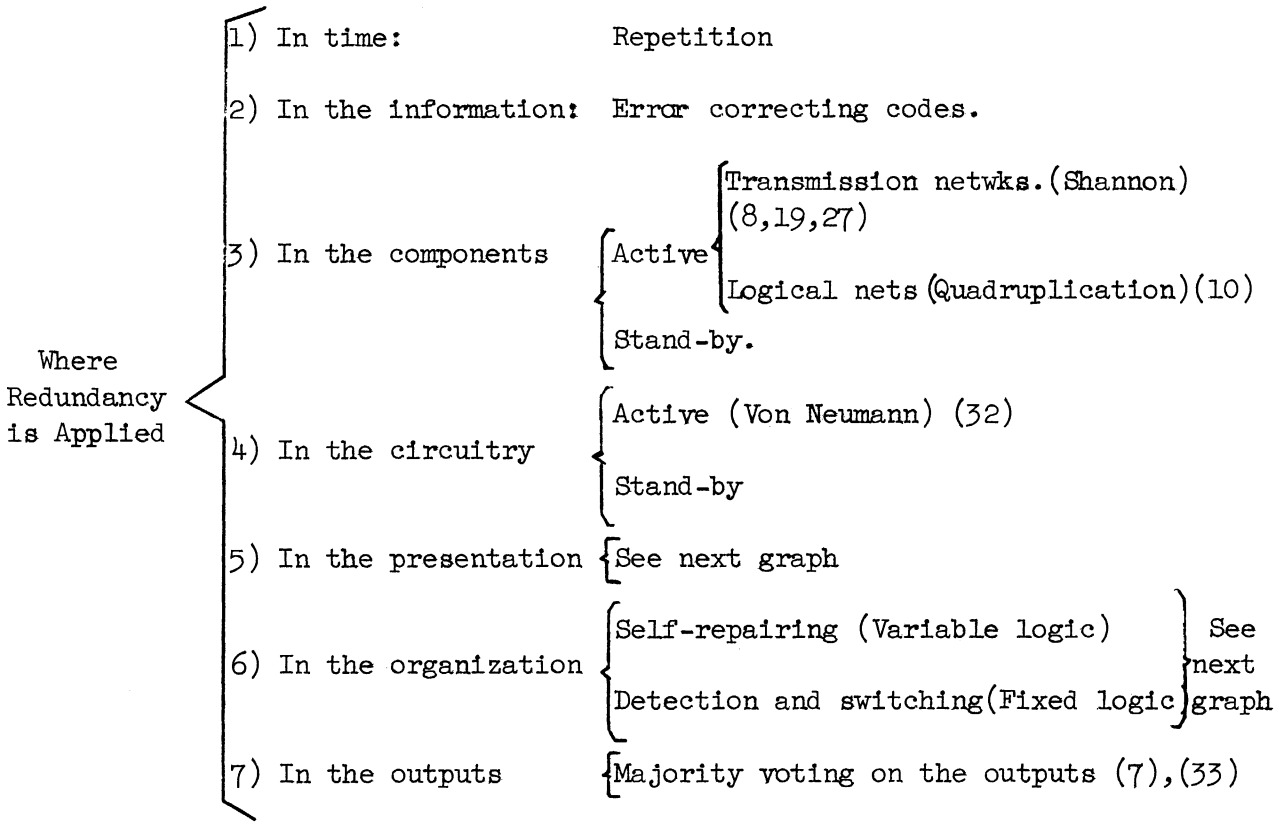
Due to the very nature of the problem, it is very difficult to present a picture of mutually exclusive methods, since even at the very first level of generality, some methods imply the use of some others.

Table 4 presents a summary of these ideas. It is to be remembered that

these are not mutually exclusive methods; since the use of one will most certainly force the use of other method or methods at some other levels.

TABLE 4

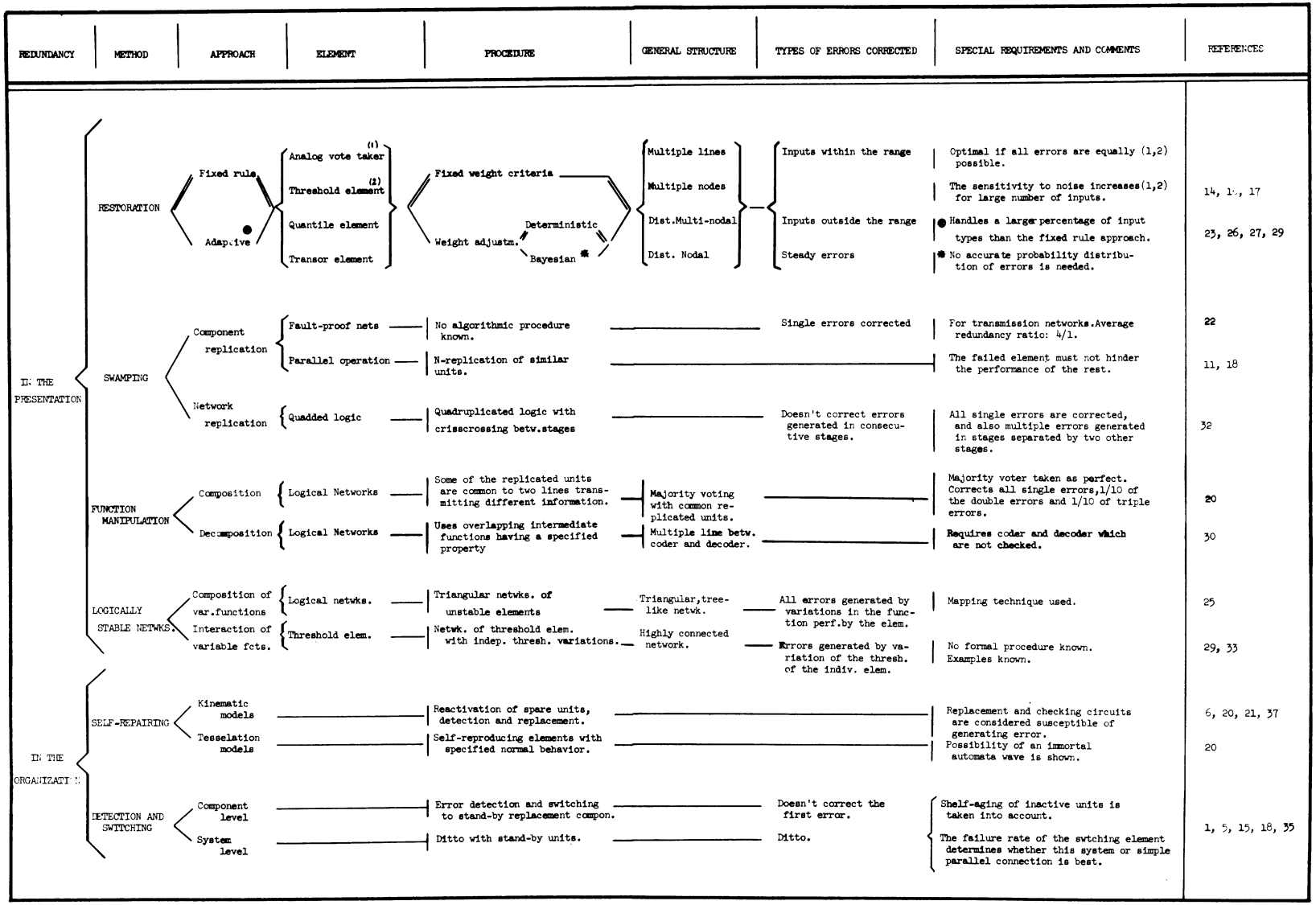
METHODS OF APPLYING REDUNDANCY



This is even more evident in the expanded Table 5, where lines 5 and 6 of the Table 4 are presented in detail. A comparison between the corresponding horizontal entries in columns 2 and 5 will show that for the most of the methods used, introducing redundance in the presentation, makes it necessary to introduce a replication of units or networks.

TABLE 5

DETAIL OF LINES 5 AND 6 OF TABLE 4



5.2. CHECKING PROGRAM APPROACH

The second possibility stated in 5.1 refers to the capability of the iterative machine to be able to afford running an extra program in the form of a checking program. In order to avoid interference between the main program or programs and the checking program, it is necessary to think of a machine organized in the shape of a torus or a cylinder. In these geometrical configurations there is continuity of the medium in which the programs move. Therefore, both the main and the checking program can incorporate in themselves the necessary instructions to move themselves in a uniform way, for example, one position to the right after each instruction.

If one thinks of the main program as a "stationary" program, then the net effect would be that of having a machine whose basic array is constantly circulating under the pattern occupied by the instructions.

A column of modules is taken as a reference starting point for each of the cycles or revolutions of the programs. Under this condition the failure of a module can occur at three different times during a cycle:

I) A module fails after the main program has used it and before the test program.

II) A module fails after the test program and before or during the main program.

III) A module fails while it is a part of the test program.

In case I, the test program merely detects the failure and "tags" the failed module to prevent its further use.

In case II, the failure will not be detected until the next cycle, and the test program not only has to tag the failed module but has to

generate a "repeat" cycle signal. This signal produces an interruption of the present program, and the instruction counter and all register are reset to the condition existing at the beginning of the cycle in which the failure occurred.

This back-tracking of the main program is a very stringent condition since it means that the state of the machine has to be reproducible, and therefore, the contents of the accumulators, registers and memory, have to be restored to the original values at the starting point of the previous cycle. Since the starting instructions for each cycle are well defined and come spaced a fixed number of positions apart, it is a matter of storing the contents of the registers and accumulator only for those instructions which are starting instructions, and to use temporary storage locations for all the store instructions in the cycle.

The use of the test program has reduced the reliability problem from one of keeping n x n modules in working condition during the whole length of the program to one of assuring the good condition of the modules during the time it takes to complete one cycle. Actually, the situation is even better since the modules left behind by the program can fail with no ill effects, even before the cycle is completed. Therefore, the last column has to be active during the whole cycle, the $(n-1)^{th}$ column has only to be active during the $[n - 1]/n$ steps of the cycle, and so on.

An even more critical analysis will show that when a module fails between the test and main program, or within the main program itself, the probability of producing an error in the computation is zero, since the condition will be detected and the whole cycle will be repeated, <u>if</u> the test program is able to survive until the end of the cycle. Survival

of the test program implies that no module should go bad while part of the test program, but it otherwise could go wrong before the test program uses it, since it will then be detected, "flagged" and avoided.

Therefore, the probability that an error will occur in the computation is equal to the probability that a module goes bad between the test and the main program, or while it is a part of the main program and that a module goes bad while part of the test program at any time between the occurrence of the bad module in the main program, and the moment the test program reaches the reference column to initiate a new cycle.

The only possible combination that can generate an undetected error in the computation is the one described above, with the specified sequence of errors. If the sequence is reversed, that is, if the error occurs in the test program first, the same failed module will introduce an error in the main program, but the change introduced in the test program will induce a "repeat cycle" condition, since the test program is compared after each cycle with a stored copy of the same.

6. A MATHEMATICAL MODEL OF AN I.C.C.

6.1 INTRODUCTION

The preceding sections have presented studies in the areas of programming, accessibility evaluation, machine organization, and reliability.

In order to determine the full capabilities of this novel organization in the form of an iterative array of modules, it is necessary either to resort to simulation procedures or to construct a mathematical model.

Once the behavior and the language of the model are determined, the model serves as a suitable tool with which one might expect to be able to answer questions relating to the computational capabilities of the machine, to the types of problems in which this power is more evident, and to the dependency of computational speed on the path connection mode.

Developing a model of an I.C.C. in the more general sense is a formidable task. We may have to simplify slightly the problem, without losing any valuable properties by establishing a valid analogy between the I.C.C. and an ensemble of Turing machines acting on a multi-dimensional tape. The next step is to keep the tape-space multi-dimensional but to replace the set of Turing machines with finite-state machines. Even then the problem is far from being simple, as shown by the fact that except for an occasional instance, the literature is barren of discussion dealing with multiple head automata.

The subject is not without merit for multiple-head automata possess capabilities beyond those of single-head machines—capabilities yet to be thoroughly explored.

This section extends the results of automata theory beyond the usual limit of one-dimensional one-tape single-headed non-halting finite state machines to encompass, in the most general case, multi-dimensional multi-tape multi-head self-halting finite state machines.

A familiarity with the material contained in the papers by McNaughton and Yamada,[41] E. F. Moore,[42] Minsky,[45] and Rabin and Scott,[43] will be necessary and sufficient for an intelligent reading of this section. Established results of other persons will usually be stated and used without proof. The author has attempted to give proper credit to the work of others. Thus, all theorems and remarks contained in this section which are not credited to others are, to the best of the author's knowledge, original.

The material presented in this section is arranged into seven subsections.

Section 6.1 is the introduction.

Section 6.2 introduces the concepts of alphabet, tape, and n-head machine. The operation of n-head machines on tapes is defined and the manner in which n-head machines accept and reject inputs is described along with the notion of how n-head machines define sets of inputs. Section 6.3 presents a number of operations on alphabets, tapes and sets of tapes which constitutes a language by which, beginning with primitive alphabets, one can represent certain sets of m-tuples of tapes. The language developed in this section includes as one of its parts the language of regular expressions. Section 6.4 contains a set of six pairs of analysis-synthesis theorems relating the sets of inputs defined by n-head machines to expressions in the language of Section 6.3. The theorem pairs are ordered according to the complexity of the machines involved, beginning with 1-way 1-dim 1-head machines and terminating with 2-

way D-dim n-head m-tape machines. Section 6.5 consists of a collection of algorithms and theorems pertaining to n-head machines. In particular, algorithms are given to decide if any given n-head machine is 1-way and to decide if any given regular expression is realizable. The section also develops theorems dealing with the questions:

1) Does a given machine accept a given input (the "particular input decision question+)?
2) Does a given machine accept any input (the "emptiness decision question")?
3) What is the relationship between state and transition accessibility and the emptiness decision question?
4) What are the Boolean properties of n-head machines?
5) What is the relationship between the number of heads a particular machine possesses and the speed with which this machine reacts to inputs?

Section 6.6 suggests several topics for further study. The topic areas are described and some partial results pertinent to each area are given. Section 6.7 is the concluding section and the results are summarized and discussed.

6.2 n-HEAD FINITE STATE MACHINES—A DESCRIPTION

6.2.1 Alphabets

Def. 2.1 An alphabet is a finite collection of symbols.

By convention alphabets will be denoted by some variation on the letter Σ. Thus $\Sigma_1 = \{B, 0, 1\}$, $\Sigma_2 = \{B,a,b,c\}$ and $\Sigma_3 = \{\#,!,?,\S\}$ are all examples of alphabets.

6.2.2 Tapes

Def. 2.2 If D is a positive integer then D-space is defined as a space of dimension D in which a Cartesian coordinate system has been embedded, each coordinate ranging over the integers from $-\infty$ to $+\infty$; around each coordinate point is centered a unit D-cube called a cell.

Thus D-space consists of a D-dimensional space divided and covered by an orderly array of unit D-cubes (or cells) where each cell is labelled with a unique coordinate point.

Def. 2.3 Let Σ be a alphabet; t is defined as a D-dimensional (D-dim) tape over Σ if t consists of a D-space in which each cell contains precisely one element of Σ.

We adopt the convention that a cell in which no symbol is written will be called empty; a cell containing a symbol will be called filled. It follows from the definition of tape that if t is a tape in some D-space then every cell in that D-space is filled.

In this paper the symbol B will be used exclusively to denote the blank. B is a legitimate possible symbol in any alphabet. Any

cell of any tape will be considered blank if and only if it contains B.

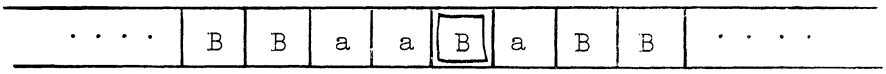
Def. 2.4 Any tape t will be a finite tape if and only if t contains a finite number of non-blank cells.

If t is a finite tape of dimension D, it is equivalent to say that the non-blank portion of t can be enclosed in a rectangular D-dimensional parallelepiped of finite dimensions. In this section we will limit consideration to arbitrarily large but finite tapes. Therefore whenever the term "tape" is used it will be understood that "finite tape" is implied.

Def. 2.5 The initial cell of any tape will be that cell located at the origin of that tape's coordinate system.

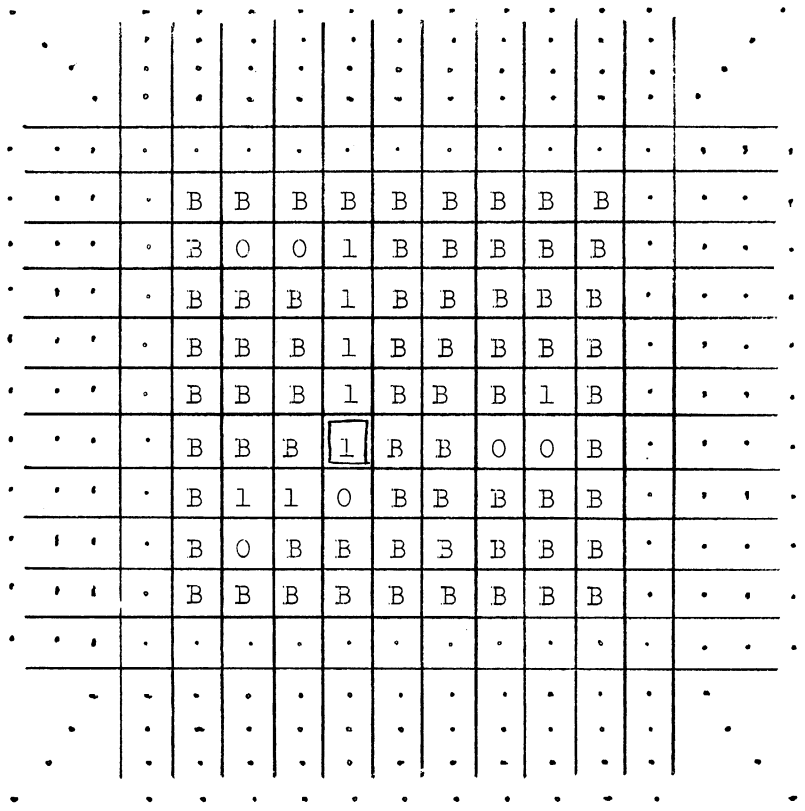
It will be convenient to omit explicit representation of the coordinate system of a tape; in such cases the initial cell of the tape will be indicated by a double boundary and the coordinate directions established by prior convention. In this paper for all 1-dim and 2-dim tapes the up, down, left, right directions will be respectively the coordinate directions +2, -2, -1, +1.

For example, Figure 75 gives an illustration of a 1-dim tape over $\Sigma_1 = \{B,a\}$ and Figure 76 gives an illustration of a 2-dim tape over $\Sigma_2 = \{B,0,1\}$.



Tape t_1

Figure 75



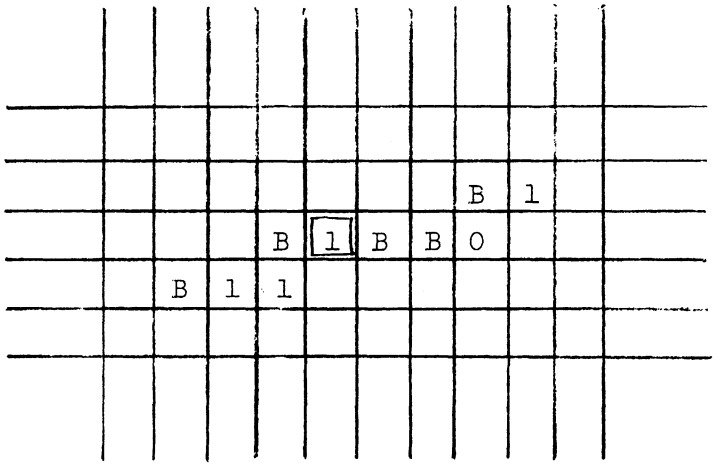
Tape t_2

Figure 76

Def. 2.6 Let Σ be an alphabet; t' is defined as a D-dim partial tape over Σ if t' consists of a D-space in which a finite number of cells contain precisely one element of Σ, all other cells being empty.

Def. 2.7 If t is a tape, t_s is defined as a subtape of t if and only if t_s is a partial tape of the same dimension as t and for each filled cell in t_s the corresponding cell of t contains the same symbol.

Thus, for example, t_{2s} given in Figure 77 is a subtape of t_2 (t_2 is given in Figure 78).



Subtape t_2'

Figure 77

Def. 2.8 If one is in any cell of a D-space with the coordinate axes identified from the 1-st to the D-th, to move d where d is some integer in the range $-D \leq d \leq D$ is defined as moving one cell in the $|d|$ direction, negative d meaning backward, positive d meaning forward and zero d meaning no move.

6.2.3 Machines

Def. 2.9 An n-head finite state machine (or just n-head machine) is a system $\mathcal{A} = < C, S, s^I, M >$ where

C: the characterization of the machine is a list of

a) the set of heads $H = \{h_i\}, i = 1,2, \ldots, n$

b) a partitioning of H into disjoint subsets $H_1, H_2, \ldots, H_m$ $(m \leq n)$; $\mathcal{A}$ works on m tapes, the heads of H_i reading tape t_i

c) two sets $\{\Sigma_i\}$ and $\{D_i\}$, $i = 1,2, \ldots, m$ where Σ_i is the alphabet that all the heads in H_i read in common and where D_i is the dimension of the space (tape) in which the heads of H_i move.

S: a finite non-empty set which together with the states "ACCEPT" (abbreviated A) and "REJECT" (abbreviated R) which are not in S make up the set of internal states of $\mathcal{A}$.

s^I: an element of S designated as the initial state of $\mathcal{A}$.

M: a mapping from

$$S \times \underbrace{\Sigma_1 \times \Sigma_1 \times \ldots \times \Sigma_1}_{\overline{\overline{H}}_1} \times \Sigma_2 \times \ldots\ldots \times \Sigma_{m-1} \times \underbrace{\Sigma_m \times \ldots \times \Sigma_m}_{\overline{\overline{H}}_m}$$

to

$$S \times \underbrace{D_1^{\pm} \times \ldots \times D_1^{\pm}}_{\overline{\overline{H}}_1} \times D_2^{\pm} \times \ldots \times D_{m-1}^{\pm} \times \underbrace{D_m^{\pm} \times \ldots \times D_m^{\pm}}_{\overline{\overline{H}}_m} \cup \{A,R\}$$

where $\overline{\overline{H}}_i$ = the number of elements in H_i

and $D_i^{\pm} = \{d \mid d \text{ is an integer in the range } -D_i \text{ to } +D_i\}$;

M constitutes the table of transitions of $\mathcal{A}$.

Def. 2.10 $\mathcal{A} = < C,S,s^I,M >$ accepts or rejects any m-tuple of tapes $t = (t_1, t_2, \ldots, t_m)$ in the following manner [it is understood that for $i = 1,2, \ldots,m$ t_i is a D_i-dim tape written over Σ_i in accordance with C of $\mathcal{A}$]:

1) $\mathcal{A}$ starts in state s^I with all the heads of each H_i resting on the initial cell of each t_i.

2) If $\mathcal{A}$ is in state s_k and the heads read the n-tuple of symbols σ

$$(\sigma \in \underbrace{\Sigma_1 \times \ldots \times \Sigma_1}_{\overline{\overline{H}}_1} \times \Sigma_2 \times \ldots \times \Sigma_{m-1} \times \underbrace{\Sigma_m \times \ldots \times \Sigma_m}_{\overline{\overline{H}}_m})$$

and M of $\mathcal{A}$ has the entry

$(s_k,\sigma) \rightarrow (s_\ell,d_1,d_2, \ldots,d_n)$

where $(d_1,d_2,\ldots, d_n) \in \underbrace{D_1 \text{ x}\ldots\text{x } D_1}_{H_1} \text{ x } D_2 \text{ x}\ldots\text{x } \underbrace{D_m \text{ x}\ldots\text{x } D_m}_{H_m}$

then $\mathcal{A}$ goes to state s_ℓ and each head h_i of $\mathcal{A}$ moves d_i.

3) $\mathcal{A}$ continues to repeat step 2 above; the heads of $\mathcal{A}$ move back and forth on their respective tapes and the machine passes through a sequence of internal states. If in a finite number of cycles $\mathcal{A}$ goes into the A(R) state then the machine stops and is said to <u>accept</u> (<u>strongly reject</u>) t. If $\mathcal{A}$ never goes into A or R then $\mathcal{A}$ is said to <u>weakly reject</u> t.

<u>Example 2.1</u> $\mathcal{A}_{2.1} = < C_1,S_1,s_1^I,M_1 >$ where

C_1 = $\mathcal{A}_{2.1}$ is a 2-head machine operating with both heads reading the same 1-dim tape written over $\Sigma_1 = \{B,0,1\}$

$S_1 = \{s_1,s_2\}$

$s_1^I = s_1$

M_1 =

s \ $\Sigma_1 x \Sigma_1$	BB	B0	B1	0B	00	01	1B	10	11
s_1	$s_1,0,0$	$s_2,-1,0$	$s_2,-1,0$		$s_1,1,0$	$s_1,1,0$		$s_1,1,0$	$s_1,1,0$
s_2	A			R	$s_2,-1,1$	R	R	R	$s_2,-1,1$

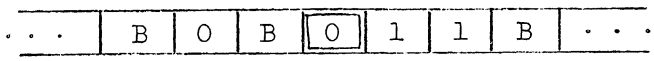
$\mathcal{A}_{2.1}$ accepts the tapes

...	B	**1**	0	0	1	0	0	1	B	B	...	

...	B	**1**	0	1	1	0	1	B	B	...		

..	B	B	0	1	B	**1**	0	1	B	B	1	B	B	..

while strongly rejecting



and weakly rejecting

Def. 2.11 Given any internal state s of machine $\mathcal{A}$ which works over m-tuples of tapes, s is said to be accessible (an accessible state) if and only if there is some input m-tuple that takes $\mathcal{A}$ from s^I to s.

Def. 2.12 Given any transition τ of $\mathcal{A}$ (a transition of $\mathcal{A}$ is an entry in the M table of $\mathcal{A}$) corresponding to reading the n-tuple of symbols σ while being in state s, τ is said to be accessible (an accessible transition) if and only if there is some input m-tuple that takes $\mathcal{A}$ from s^I to s and presents $\mathcal{A}$ with input σ.

Note 2.1 If τ is an inaccessible transition of machine $\mathcal{A}$ (as is, for example, the transition on s_1,0,B in $\mathcal{A}_{2.1}$ of example 2.1) then the destination state and the head movement of τ can be left unspecified without affecting the behavior of $\mathcal{A}$.

Def. 2.13 $\mathcal{A}$, an n-head machine, is called 1-way if an only if for each head h_i of $\mathcal{A}$ on all accessible transitions of $\mathcal{A}$ h_i moves a fixed direction d_i. If $\mathcal{A}$ is not 1-way it is 2-way.

Note 2.2 It is sufficient but not necessary that $\mathcal{A}$ be 1-way if all transitions specify the same head movements. Clearly inaccessible transitions can have any head movement at all and never affect the operation of $\mathcal{A}$.

Def. 2.14 The set of all m-tuples of tapes accepted by any n-head finite state machine $\mathcal{A}$ is denoted by $T(\mathcal{A})$.

Def. 2.15 If $\mathcal{A}$ is any n-head machine working on single tapes and t any tape in $T(\mathcal{A})$ then $g_{\mathcal{A}}(t)$, the generator of $\mathcal{A}$ in t, is defined as that subtape of t in which the filled cells are precisely those cells of t that $\mathcal{A}$ actually scans while accepting t. If $\mathcal{A}$ works on m-tuples of tapes and t is any m-tuple in $T(\mathcal{A})$ then $g_{\mathcal{A}}(t)$ is the m-tuple of subtapes derived by retaining as filled only the cells actually scanned in accepting t.

For example

t = | ... | B | B | 1 | 0 | B | 1 | 0 | 0 | 1 | 0 | 0 | 1 | B | B | .... |

is in $T(\mathcal{A}_{2.1})$ [see example 2.1] and

$g_{\mathcal{A}_{2.1}}(t)$ = | | B | 1 | 0 | 0 | 1 | 0 | 0 | 1 | B | | | |

= | B | 1 | 0 | 0 | 1 | 0 | 0 | 1 | B |

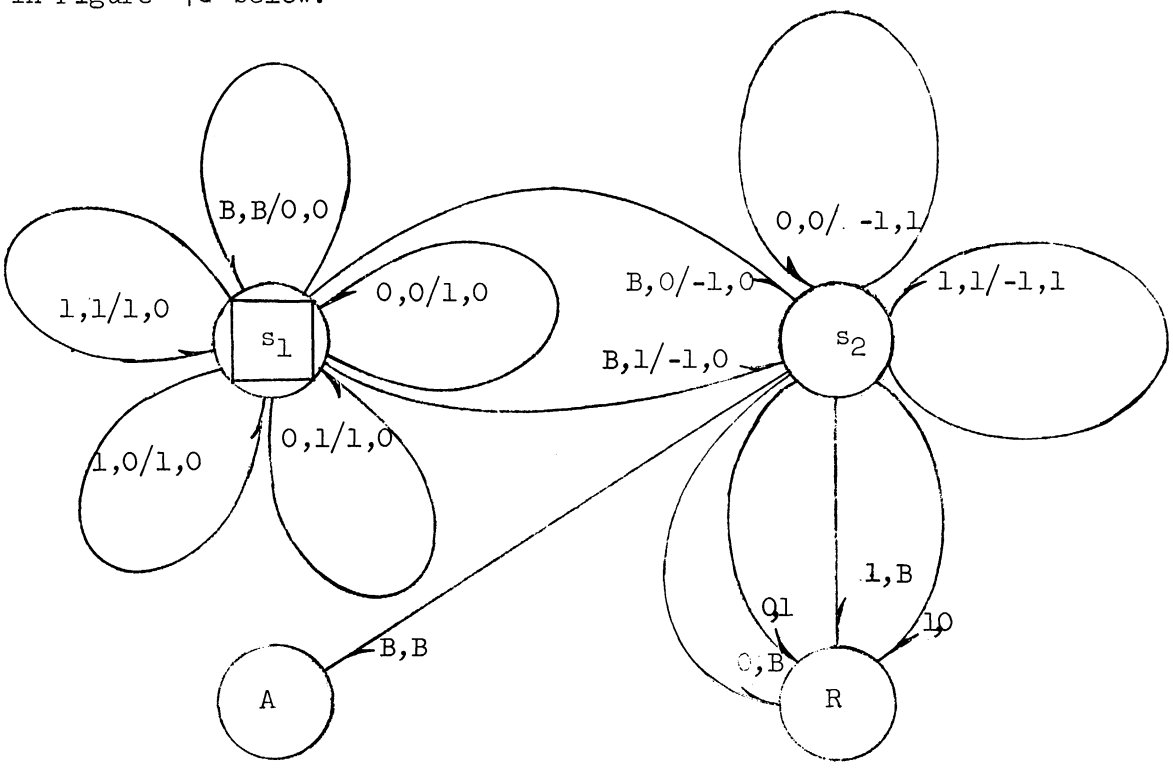
Def. 2.16 The set of all generators accepted by any n-head finite state machine $\mathcal{A}$ is denoted by $G(\mathcal{A})$. $G(\mathcal{A}) = \{g_{\mathcal{A}}(t) \mid t \in T(\mathcal{A})\}$.

6.2.4 State Graphs

As in example 2.1 any n-head finite state machine $\mathcal{A} = \langle C, S, s^I, M \rangle$ can be described by listing the set of states S, mentioning the initial state s^I, and by giving the table of moves M in tabular form. There is, however, a convenient graphical representation of any finite state machine known as the state graph. In it the set of internal states is represented as labelled circles.

The initial state is indicated by an inscribed square. Transitions are represented by labelled arrows such that if an arrow emanates from state s_k and impinges on state s_ℓ and is labelled with the symbol σ/d then $\mathcal{A}$ when in state s_k and reading n-tuple σ will fall into state s_ℓ with head movements according to n-tuple d.

For example, the state graph of machine $\mathcal{A}_{2.1}$ is given in Figure 78 below.



State Graph of $\mathcal{A}_{2.1}$

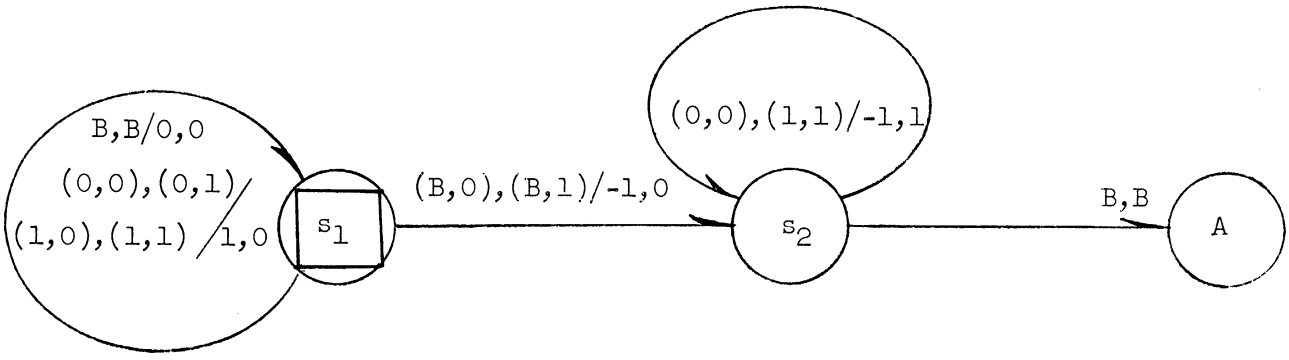
Figure 78

Note 2.3 If in any machine $\mathcal{A}$ several inputs $\sigma_1, \sigma_2, \ldots, \sigma_p$ all causes $\mathcal{A}$ to go from state s_k to state s_ℓ, with associated head movements $d_1, d_2, \ldots, d_p$ then only one arrow will be drawn from s_k

to s_ℓ in $\mathcal{A}$'s state graph and it will be labelled σ_1/d_1, σ_2/d_2,..., σ_p/d_p. If $d_1 = d_2 = \ldots = d_p = d$ one may further simplify the arrow label to $\sigma_1, \sigma_2, \ldots, \sigma_p/d$.

Note 2.4 In order to simplify the drawing of state graphs this paper will adopt the convention that the R state and all transition arrows to R will not be represented explicitly. One will understand that given any machine $\mathcal{A}$ in some state s_k and reading the input n-tuple of symbols σ, if no arrow with the input label σ leaves s_k then $\mathcal{A}$ will go to REJECT. This convention in no way alters the behavior of any machine for $T(\mathcal{A})$ and $G(\mathcal{A})$ remain unchanged as does the ability of $\mathcal{A}$ to strongly or weakly reject any tape.

Applying the conventions of Notes 2.3 and 2.4 to $\mathcal{A}_{2.1}$ yields the state graph given in Figure 79.



Simplified State Graph of $\mathcal{A}_{2.1}$

Figure 79

Note 2.5 Since this section is concerned only with finite tapes it follows that all tapes to be considered must contain the symbol B an infinite number of times. Because of this we will require all heads of all machines to include B in their alphabets.

Note 2.6 Observe that in the definition of any finite state machine $\sum_{i=1}^{m} \overline{\overline{H_i}} = n$.

Note 2.7 We will adopt the convention that if $H = \{h_1, h_2, \ldots, h_n\}$ then the first $\overline{\overline{H_1}}$ heads of H will constitute H_1, the next $\overline{\overline{H_2}}$ heads of H will constitute H_2, etc. ... One in no way limits the class of n-head machines by doing this since any machine can be put in this form by judicious labelling of the heads.

Note 2.8 In the definition of n-head machine it is required that each head begin on the initial cell of its respective tape. One may ask if the power of n-head machines is increased by allowing the heads to adopt some other fixed but not initial cell starting configuration. The answer is negative: if $\mathcal{M}$ is any n-head machine in which each head starts on some fixed but not necessarily initial cell then there exists an n-head machine $\mathcal{M}'$ which has all heads starting on initial cells and which is equivalent to $\mathcal{M}$ (i.e., $T(\mathcal{M}') = T(\mathcal{M})$). The construction of $\mathcal{M}'$ from $\mathcal{M}$ consists of adding a set of states $s'_0, s'_1, \ldots, s'_p$ to S the set of states of $\mathcal{M}$. s'_0 is the initial state of $\mathcal{M}'$. For all inputs $\mathcal{M}'$ has the transitions $s'_0 \to s'_1 \ldots \to s'_2 \to s^I$ p is made sufficiently large and appropriate movement n-tuples are associated with each transition such that after p + 1 cycles $\mathcal{M}'$ is in state s^I and the heads are in the desired starting position; from then on $\mathcal{M}'$ acts precisely like $\mathcal{M}$.

Note 2.9 In the definition of n-head machine it is required that each head movement be either a stand still or a unit jump along one of the coordinate axes. One may ask if the power of n-head machines is

increased by allowing each head movement to be a finite determined jump but not necessarily unit or along a coordinate direction. The answer is negative: if $\mathcal{A}$ is any n-head machine in which each head movement is a finite determined jump then there exists an n-head machine $\mathcal{A}'$ which has all head movements unit jumps along coordinate axes and which is equivalent to $\mathcal{A}$. The construction of $\mathcal{A}'$ from $\mathcal{A}$ consists of adding a number of states to $\mathcal{A}$ such that each non-unit jump is decomposed into a chain of unit jumps, each chain replacing a non-unit jump transition.

Note 2.10 Readers familiar with the work of Kleene, Rabin and Scott, McNaughton and Yamada, et al. may wonder at the relationship between the machines defined by Rabin and Scott (RS machines) and the n-head machines we have defined in this section. RS machines and n-head machines are both finite state deterministic machines; they do, however, differ in several essential ways:

1) An RS machine has one reading head. An n-head machine has n reading heads; each head may read a different alphabet and one or more heads may be placed on a tape.
2) An RS machine works only on 1-dim tapes. An n-head machine can, in general, work on tapes of finite but arbitrarily large dimension.
3) The method by which n-head machines accept or reject tapes differs from that of RS machines. One of the internal states of any n-head machine is the ACCEPT state; if the machine ever goes to

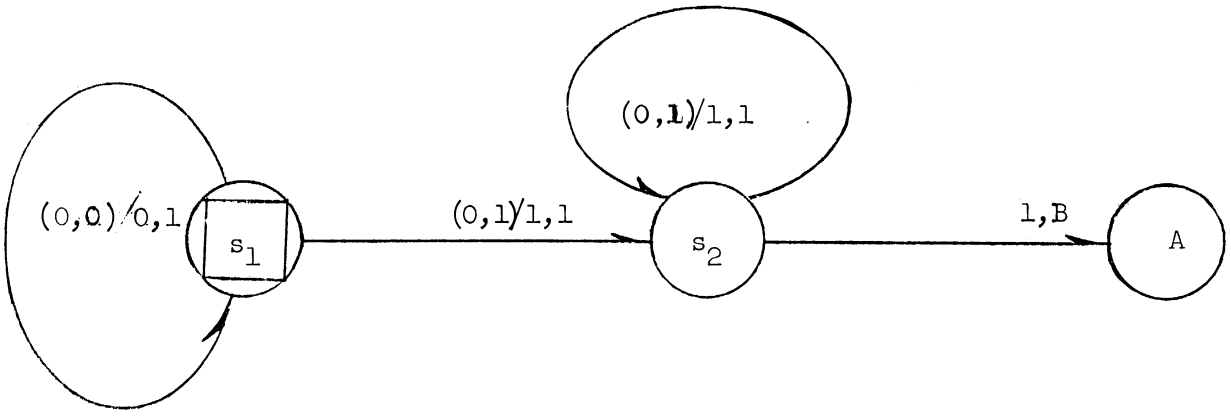
ACCEPT the machine stops and is said to accept the tape; the tape is rejected if the machine goes to the REJECT state. An RS machine, on the other hand, can only decide on accepting or rejecting a given tape precisely at the moment that the reading head leaves the filled portion of the tape and "steps off" the tape in some manner.

Note 2.11 In a real sense, given any n-head machine $\mathcal{M}$, $G(\mathcal{M})$ is a better parameter of the behavior of $\mathcal{M}$ then $T(\mathcal{M})$. For all $\mathcal{M}$, $\overline{\overline{T(\mathcal{M})}} = 0$ or ∞. This is clear since if $\mathcal{M}$ accepts no input then $\overline{\overline{T(\mathcal{M})}} = 0$; if, however, $\overline{\overline{T(\mathcal{M})}} \neq 0$ then there is at least one $t_1 \in T(\mathcal{M})$. Consider $g_{\mathcal{M}}(t_1)$; $g_{\mathcal{M}}(t_1)$ has an infinite number of empty cells; therefore by filling these cells of $g_{\mathcal{M}}(t_1)$ with elements of Σ, the alphabet of t, we can generate an infinite number of distinct tapes all in $T(\mathcal{M})$, so $\overline{\overline{T(\mathcal{M})}} = \infty$. $\overline{\overline{G(\mathcal{M})}}$ is not limited to 0 or ∞, but can be any integer value depending on $\mathcal{M}$.

Further, if g is a generator of $\mathcal{M}$ then any tape t containing g as a subtape is accepted by $\mathcal{M}$ whether t contains symbols out of the alphabets of $\mathcal{M}$ or not (in other words the empty cells of g are "don't care" cells whose contents do not affect the behavior of $\mathcal{M}$). Thus given $G(\mathcal{M})$ we know $T(\mathcal{M})$.

This section is concluded by an example, machine $\mathcal{M}_{2.2}$, which demonstrates that 2-head machines are more powerful than 1-head machines. $\mathcal{M}_{2.2}$ is a 2-head machine reading 1-dim tapes over the alphabet $\Sigma = \{B,0,1\}$. $\mathcal{M}_{2.2}$ will accept any tape which starting at the initial

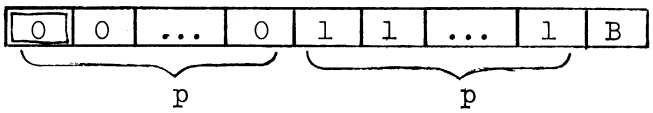
cell and moving right has p 0's followed by p 1's followed by B where p = 1,2,3, ... It is an established fact that such a set of tapes cannot be represented by a 1-head machine.(6)



Machine $\mathcal{A}_{2.2}$

Figure 80

For $\mathcal{A}_{2.2}$ observe that

$G(\mathcal{A}_{2.2}) = \{g \mid g =$  $p = 1,2,\ldots\}$

$T(\mathcal{A}_{2.2}) = \{t \mid$ t is 1-dim tape and $\exists$ g.϶. $g \in G(\mathcal{A}_{2.2})$ and g is a subtape of t. $\}$

6.3 THE LANGUAGE

6.3.1 Operations on Alphabets

Def. 3.1 If $\Sigma_1,\Sigma_2, \ldots, \Sigma_m$ are alphabets then the column alphabet of $\Sigma_1,\Sigma_2, \ldots, \Sigma_m$, denoted by

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix}$$

is defined as the alphabet consisting of all column m-tuples over the alphabets $\Sigma_1,\Sigma_2, \ldots, \Sigma_m$; i.e.,

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix} = \left\{ \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_m \end{bmatrix} \;\middle|\; \begin{array}{l} \sigma_1 \in \Sigma_1 \\ \sigma_2 \in \Sigma_2 \\ \vdots \\ \sigma_m \in \Sigma_m \end{array} \right\} .$$

For example, if $\Sigma_1 = \{B,O\}$ and $\Sigma_2 = \{a,b,c\}$ then

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \end{bmatrix} = \left\{ \begin{bmatrix} B \\ a \end{bmatrix}, \begin{bmatrix} B \\ b \end{bmatrix}, \begin{bmatrix} B \\ c \end{bmatrix}, \begin{bmatrix} O \\ a \end{bmatrix}, \begin{bmatrix} O \\ b \end{bmatrix}, \begin{bmatrix} O \\ c \end{bmatrix} \right\}.$$

Def. 3.2 If Σ is an alphabet and D some positive integer then Σ indexed by D, denoted by Σ/D, is defined as the alphabet consisting of all doubletons of the form σ/d where $\sigma\epsilon\Sigma$ and $d\epsilon D^{\pm}$; i.e.

$\Sigma/D = \{(\sigma/d) \mid \sigma\epsilon\Sigma,$ d is integer in the range -D to +D$\}$.

For example, if $\Sigma = \{0,1\}$ then

$\Sigma/2 = \{0/-2,\ 0/-1,\ 0/0,\ 0/1,\ 0/2,\ 1/-2,\ 1/-1,\ 1/0,\ 1/1,\ 1/2\}$.

6.3.2 Operations on Partial Tapes

Def. 3.3 If t is a partial tape existing in some D-space and written over the alphabet

$$\Sigma = \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix}$$

then t will be understood to have m channels where the i-th channel of t will be the tape existing in D-space and written over Σ_i and obtained from t by replacing every occurrence of an element

$$\begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_m \end{bmatrix} \quad \text{in} \quad \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix}$$

with the single element σ_i.

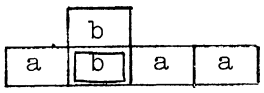
For example, if

t =

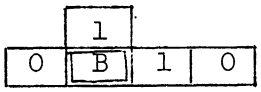
	b 1		
a 0	b B	a 1	a 0

is a 2-dim partial tape written over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \end{bmatrix}$

where $\Sigma_1 = \{a,b\}$ and $\Sigma_2 = \{B,0,1\}$ then the 1-st channel of t is



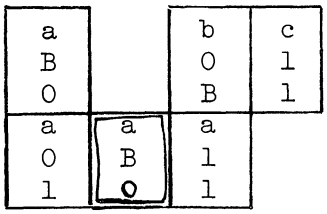
and the 2-nd channel of t is



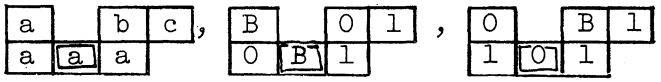
Def. 3.4 If t is a partial tape over $\Sigma = \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix}$ then the separation of t, denoted by t^{Ψ}, is defined as the m-tuple of partial tapes, $(t_1, t_2, \ldots, t_m)$ where t_i equals the i-th channel of t.

For example, if

$t =$



then $t^{\Psi} =$ (

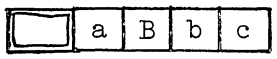


).

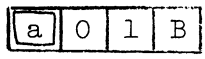
Note 3.1 If t is a tape over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_m \end{bmatrix}$ and m = 1 then $t^{\Psi} = t$.

Def. 3.5 t will be said to be an initial partial tape if and only if t is 1-dim and all the cells to the left of the initial cell are empty.

For example, $t_1 =$



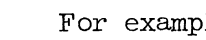
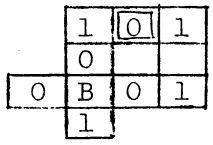
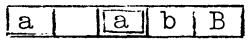
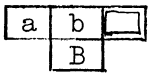
and $t_2 =$



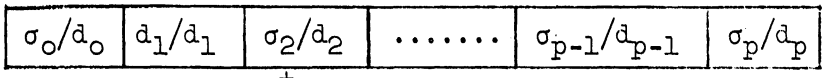
are initial partial tapes and $t_3 =$

is not.

Def. 3.6 t will be said to be a connected partial tape if and only if all cells of t are empty or if the initial cell of t is filled and for any two filled cells in t there exists a string of adjacent filled cells connecting the original two cells.

For example, t_1 =  and t_2 = 
are connected while t_3 = 
and t_4 =  are not.

Def. 3.7 If t is an initial connected partial tape over an alphabet of the form Σ/D then the fold of t (or t fold), denoted by t^f, is defined as the D-dim partial tape obtained from t in the following manner:

1) t = 
where $\sigma_i \epsilon \Sigma$ and $d_i \epsilon D^{\pm}$.

2) read t from left to right, one cell at a time, and simultaneously write out the following partial tape t' in an originally all empty D-space...

   a) let i = 0,

   b) write σ_o in the initial cell of the D-space and move d_o,

   c) augment i by 1,

   d) write σ_i in the cell under consideration and move d_i,

   e) repeat c,d until i = p at which time one writes σ_p in the cell under consideration and then stops.

The resulting partial tape t' will be finite (since t was finite) and each cell of t' will contain a finite number of elements of Σ.

3) Examine the cells of t' that contain more than one element of Σ. For each such cell

a) if the elements of Σ that it contains are identical, erase all but one of the elements; the resulting D-dim partial tape is t^f.

b) if any one of the cells of t' contains non-identical elements of Σ then there is no partial tape that equals t^f and t^f is defined as ϕ, the null set.

For example,

if t_1 =

B/0	B/0	B/-1	0/-1	1/1	0/1

then t_1' =

1	0,0	B,B,B

and t_1^f =

1	0	B

,

if t_2 =

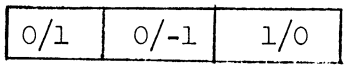
a/1	b/1	a/2	a/2	a/-1	b/-1	b/-2	b/-2	a/-2	c/-2	b/-1	a/2	b/1

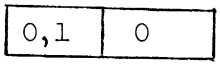
then t_2' =

	b	b	a
	b		a
	a,a	b	a
b	c		
a	b		

and t_2^f =

	b	b	a
	b		a
	a	b	a
b	c		
a	b		

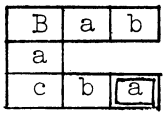
if t_3 = 

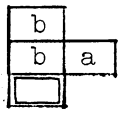
then t_3' = 

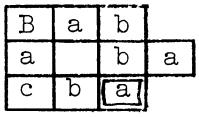
and $t_3^f = \emptyset$.

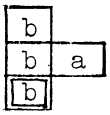
Note 3.1 If σ_p/d_p is the last symbol of some initial connected tape t, then t^f is independent of d_p. Therefore we can omit d_p if we wish and still define the fold operation without introducing any ambiguity.

Def. 3.8 If t_1 and t_2 are partial tapes of the same dimension then the cover of t_1 and t_2, denoted by $t_1 \textcircled{c} t_2$, is defined as the smallest partial tape that contains t_1 and t_2 as subtapes, if no such partial tape exists then $t_1 \textcircled{c} t_2 = \emptyset$.

For example, if t_1 = 

and t_2 = 

then $t_1 \textcircled{c} t_2$ = 

but if t_3 = 

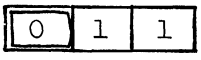
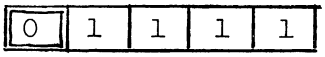
then $t_1 \textcircled{c} t_3 = \emptyset$

Note 3.2 $t_1 \textcircled{c} t_2$ can be defined operationally as follows:

1) let D be the dimension of t_1 and t_2,
2) start with an initially empty D-space; copy t_1 into it,
3) copy t_2 into the space; the result will be a finite partial tape t' each cell of which contains at most two symbols (one from t_1, one from t_2),
4) consider the cells of t' that contain two symbols; for each such cell if the symbols are identical, erase one of them ... the resulting partial tape is $t_1 \textcircled{c} t_2$; if any cell contains non-identical symbols then $t_1 \textcircled{c} t_2 = \emptyset$.

Note 3.3 If we define $\phi \odot t = t \odot \phi = \phi$ for all partial tapes t then the cover operation becomes commutative and associative, i.e., $t_1 \odot t_2 = t_2 \odot t_1$ and $t_1 \odot (t_2 \odot t_3) = (t_1 \odot t_2) \odot t_3$.

Def. 3.9 If t_1 and t_2 are initial connected tapes (therefore 1-dim) then t_1 concatenated by t_2, denoted by $\overbrace{t_1\ t_2}$ or just $t_1\ t_2$, is defined as the tape obtained by copying into the empty tail of t_1 (the empty cells of t_1 that most immediately follow, and perhaps include, the initial cell of t_1) the contents of t_2 beginning with the initial cell of t_2. The initial cell of $t_1\ t_2$ corresponds to the initial cell of t_1.

For example, if t_1 =  and t_2 =  then $t_1\ t_2$ =  .

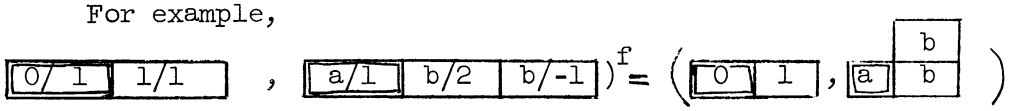
Note 3.4 The "null partial tape" (not to be confused with the null set) is that partial tape in which every cell is empty. The 1-dim null partial tape is denoted by Λ. Observe that Λ is an initial connected partial tape and that for any initial connected partial tape t, $t\Lambda = \Lambda t = t$.

Note 3.5 Observe that the concatenation operation is not commutative but is associative.

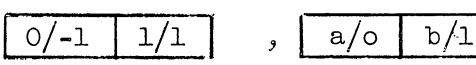
6.3.3 Operations on m-Tuples of Partial Tapes

Def. 3.10 If $t = (t_1, t_2, \ldots, t_m)$ is an m-tuple of partial tapes such that t_i^f is defined for i = 1,2, ..., m then the fold of t (or t fold), denoted by t^f, is defined as the m-tuple $(t_1^f, t_2^f, \ldots, t_m^f)$; if for some i = 1, 2, ...m $t_i^f = \phi$ then $t^f = \phi$.

For example,



and

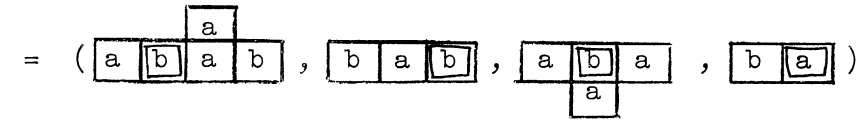


Def. 3.11 If $t = (t_1, t_2, \ldots, t_m)$ is an m-tuple of tapes and $\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix}$ an ℓ-tuple of non-zero positive integers whose sum equals $m, (\sum_{i=1}^{\ell} r_i = m)$, then the cover of t with respect to $r_1, r_2, \ldots r_\ell$, denoted by $t\textcircled{\cdot}\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix}$, is defined as the ℓ-tuple

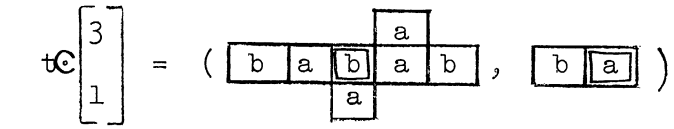
$$(t_1 \textcircled{\cdot} t_2 \textcircled{\cdot} \cdots \textcircled{\cdot} t_{r_1},\ t_{r_1+1} \textcircled{\cdot} t_{r_1+2} \textcircled{\cdot} \cdots \textcircled{\cdot} t_{r_1+r_2}, \ldots,\ t_{m-r_\ell+1} \textcircled{\cdot} t_{m-r_\ell+2} \textcircled{\cdot} \cdots \textcircled{\cdot} t_m)\ ;$$

if any element of $t\textcircled{\cdot}\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix}$ is ϕ then $t\textcircled{\cdot}\begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix} = \phi$.

For example, if



then



and $t\textcircled{\cdot}\begin{bmatrix} 2 \\ 2 \end{bmatrix} = \phi$.

6.3.4 Operations on Sets of m-Tuples of Partial Tapes

Def. 3.12 If T is a set of partial tapes (i.e., a set of 1-tuples of tapes) then the separation of T, denoted by $T^{\forall}$, is defined as the set of all m-tuples obtained by taking the separation of each element of T

(i.e., $T^{\Psi} = \{t^{\Psi} | t \epsilon T\}$).

Def. 3.13 If T is a set of m-tuples of partial tapes such that t^f is defined for all $t \epsilon T$ then the fold of T (or T fold), denoted by T^f, is defined as the set of all m-tuples obtained by taking the fold of each element of T (i.e., $T^f = \{t^f | t \epsilon T\}$)

Def. 3.14 If T is a set of m-tuples of partial tapes and an ℓ-tuple of non-zero positive integers such that $t \odot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix}$ is defined for all $t \epsilon T$ then the cover of T with respect to $r_1, r_2, \ldots, r_\ell$, denoted by $T \odot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix}$ is defined as the set of all ℓ-tuples obtained by taking the cover with respect to $r_1, r_2, \ldots, r_\ell$ of each element of T

$$\left(\text{i.e., } T \odot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix} = \left\{ t \odot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_\ell \end{bmatrix} \middle| \; t \epsilon T \right\} \right).$$

Def. 3.15 If T_1 and T_2 are sets of initial connected partial tapes then T_1 concatenated by T_2, denoted by $\widehat{T_1 T_2}$ (or just $T_1 T_2$), is defined as the set of initial connected tapes obtained by concatenating all elements of T_1 with all elements of T_2; (i.e., $T_1 T_2 = \{t_1 t_2 | t_1 \epsilon T_1, t_2 \epsilon T_2\}$).

Def. 3.16 If T is a set of initial connected partial tapes then T star, denoted by T^*, is defined as the set $\Lambda \cup T \cup T^2 \cup T^3 \cup \ldots\ldots$

where $T^i = \underbrace{T\ T\ \ldots\ T}$

concatenated i times

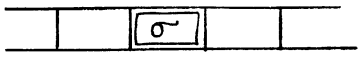
and $\cup$ denotes the conventional union of sets.

Note 3.6 T* is the smallest set that contains T and is closed under concatenation.

6.3.5 Regular Expressions

A regular expression (RE) is a symbolic means of representing certain sets of initial connected 1-dim partial tapes. The union and intersection of sets of 1-dim partial tapes will be indicated by $\cup$ and $\cap$ respectively. If T is a set of 1-dim partial tapes written over Σ then the complement of T, denoted by $\sim$ T, will consist of all 1-dim partial tapes written over Σ and not in T.

Def. 3.17 If Σ is an alphabet then

1) all elements of Σ are simple terms and all simples terms are RE's over Σ; if $\sigma \epsilon \Sigma$ then σ denotes the partial tape 

2) Λ and $\emptyset$ are RE's over Σ,

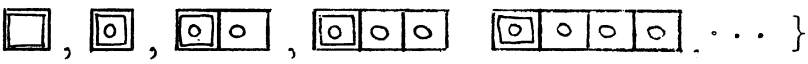
3) if α is an RE over Σ then $\sim\alpha$ and α^* are RE's over Σ,

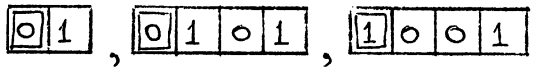
4) if α and β are RE's over Σ then $\alpha \cup \beta$, $\alpha \cap \beta$ and $\alpha\beta$ are RE's over Σ,

5) no expression is a RE over Σ unless it is obtainable by 1) to 4) above.

For example,

$0 = \{ \cdots$  $\cdots \}$

$0^* = \{$  $\cdots \}$

$(0\,1 \cup 1\,0)^*\,0\,1 = \{$  $\cdots\cdots \}$

Note 3.7 In any partial tape represented by a regular expression the leftmost symbol of the partial tape is in the initial cell of the tape and all filled cells are connected.

Note 3.8 Any finite set of 1-dim initial connected partial tapes can be represented by a regular expression simply by taking the finite union of the enumerated tapes. Not all infinite sets of partial tapes can be represented by RE's; for example the sets $0^n 1\, 0^n$ (or $0^n 1^n$) cannot be represented by a RE.(6)

6.4 EQUIVALENCE THEOREMS

6.4.1 1-Way 1-Dim 1-Head Machines

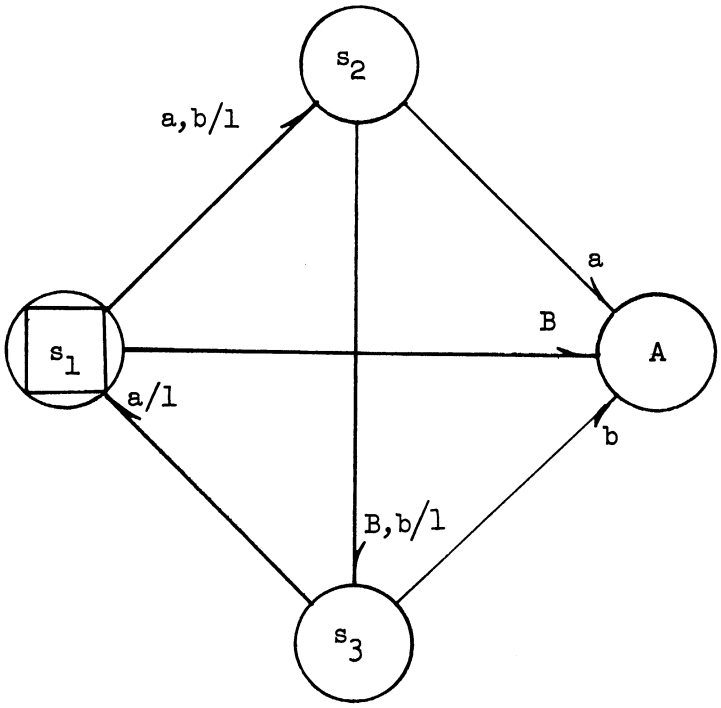
Theorem 4.1 If $\mathcal{OL}$ is a 1-way 1-dim 1-head machine working on tapes written over Σ then $G(\mathcal{OL}) = \beta$ where β is an RE over Σ.

Proof: An effective procedure exists to determine if any $\mathcal{OL}$ is 1-way (see Section 6.5). Without any loss of generality we can assume $\mathcal{OL}$ to be 1-way in the +1 direction in which event all the accessible transitions of $\mathcal{OL}$ will carry labels of the form $\sigma/1$ where $\sigma\epsilon\Sigma$. Since one can remove all inaccessible transitions from the state graph of $\mathcal{OL}$ without alterning $G(\mathcal{OL})$ one finds that the state graph of $\mathcal{OL}$ is precisely the state graph of a "one-input, one-output automaton" as described by McNaughton and Yamada,[3] $\mathcal{OL}$ having a single output state, namely the ACCEPT state. Therefore, using the procedure given in Part II of the McNaughton-Yamada paper one can construct β the RE over Σ that represents all 1-dim partial tapes taking $\mathcal{OL}$ from s^I to A; i.e., $\beta = G(\mathcal{OL})$.

QED

Note 4.1 If $\mathcal{OL}$ is a 1-way D-dim 1-head machine then $G(\mathcal{OL})$ consists of a set of partial tapes, each consisting of a D-space empty except for a finite line of symbols along one of the D-coordinates. This line of symbols can be represented as a RE over Σ, the alphabet of $\mathcal{OL}$.

Example 4.1 Figure 81 gives $\mathcal{OL}_{4.1}$ a 1-way 1-dim 1-head machine working on tapes over $\Sigma = \{a,b,B\}$; find $G(\mathcal{OL}_{4.1})$.



Machine $\mathcal{A}_{4.1}$

Figure 81

Using the technique of McNaughton and Yamada one finds that

$$G(\mathcal{A}) = B \cup (a \cup b)a \cup (a \cup b)(B \cup b)[a(a \cup b)(B \cup b)]^*[b \cup aB \cup a(a \cup b)a].$$

Theorem 4.2 If β is a RE over Σ then there exists a 1-way 1-dim 1-head machine $\mathcal{A}$ working over Σ such that $G(\mathcal{A}) = \beta$.

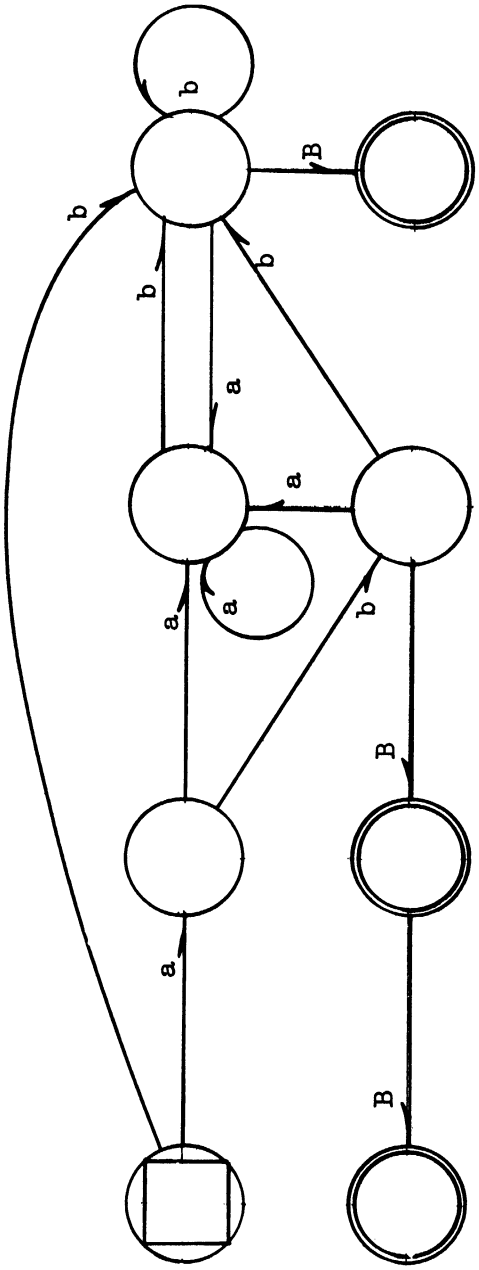
Proof: Construct, via Part III of the McNaughton and Yamada paper, the state graph of $\mathcal{A}'$ the "one-input, one-output automaton" that represents β. $\mathcal{A}'$ will in general have more than one terminal state (output = one); merge all terminal states of $\mathcal{A}'$ into one state labelled ACCEPT and delete all transitions from this state; call the new machine thus obtained $\mathcal{A}$. If t is a tape accepted by $\mathcal{A}$ then t must have a subtape that takes $\mathcal{A}'$ from s^I to a terminal state, i.e., t has a subtape in β; conversely if t has a subtape in β then t will be accepted by $\mathcal{A}$. Thus $G(\mathcal{A}) = \beta$.

QED

Example 4.2 Let $\beta = (a \cup b)^*bB \cup abBB$ be a RE over $\Sigma = \{B,a,b\}$. Find a 1-way 1-dim 1-head machine $\mathcal{A}_{4.2}$ such that $G(\mathcal{A}_{4.2}) = \beta$.

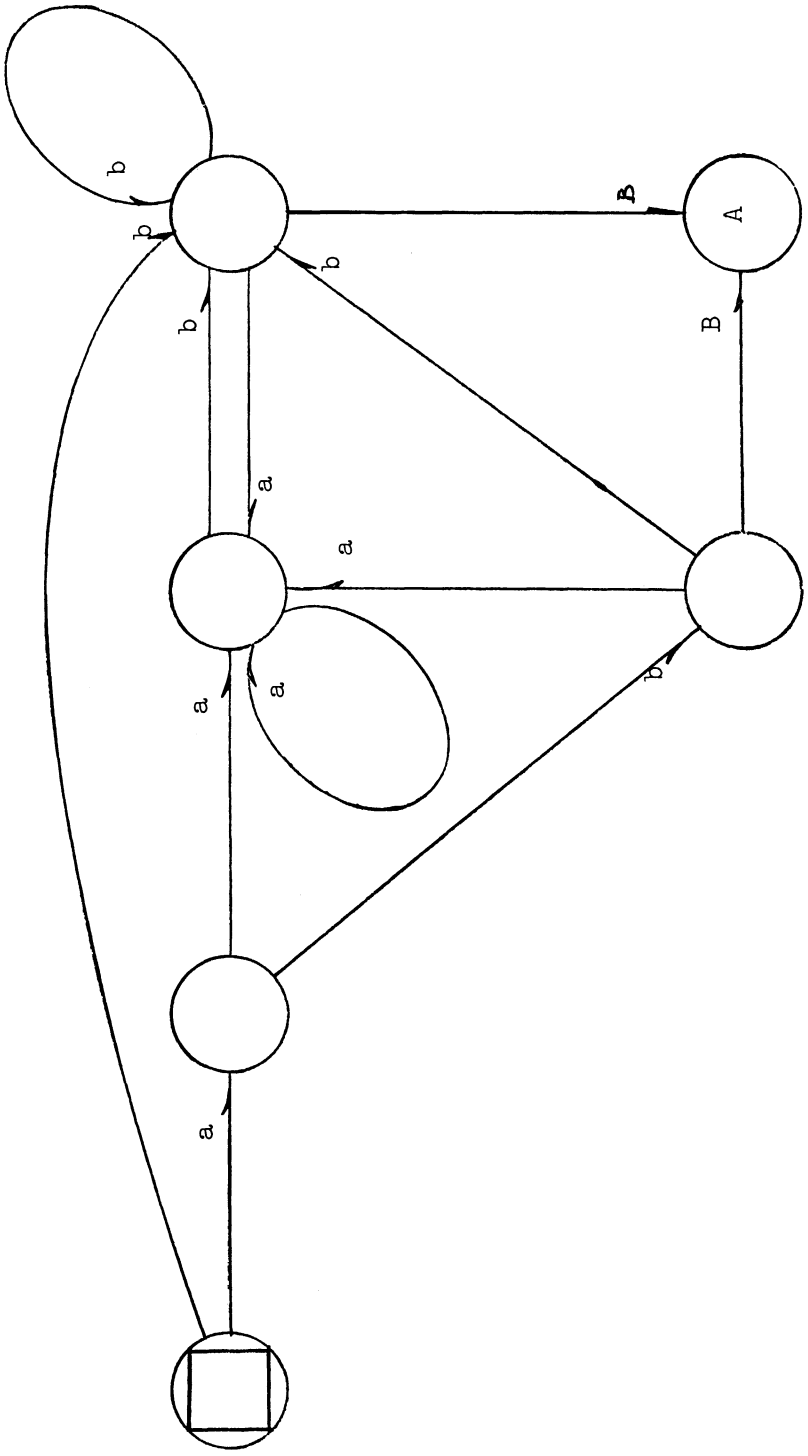
Using the McNaughton and Yamada technique one first constructs $\mathcal{A}'_{4.2}$ (Figure 82) the one-input one-output machine that represents β [terminal states of $\mathcal{A}'_{4.2}$ are represented by double circles].

By merging the terminal states of $\mathcal{A}'_{4.2}$ into a single ACCEPT state and by deleting all transitions from ACCEPT one obtains the desired machine $\mathcal{A}_{4.2}$, (Figure 83). The head movements for all transitions in $\mathcal{A}_{4.2}$ are understood to be +1.



Machine $\mathcal{A}'_{4.2}$

Figure 82



Machine $\mathcal{M}_{4.2}$

Figure 83

6.4.2 1-Way 1-Dim n-Head n-Tape Machines

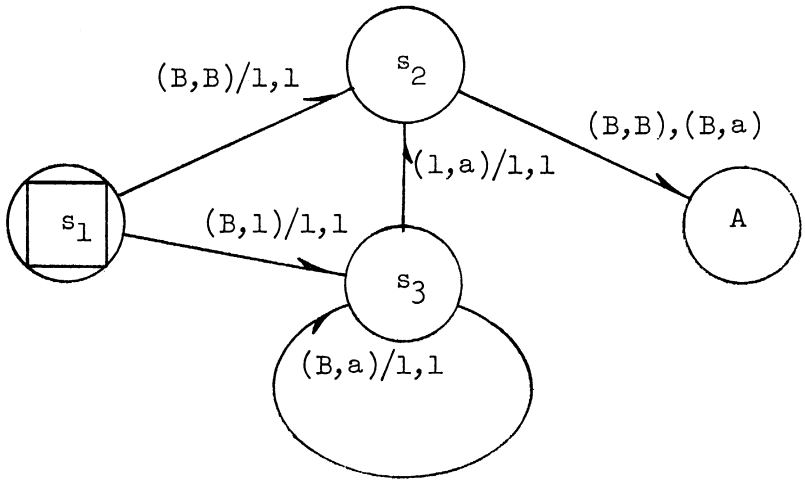
Theorem 4.3 If $\mathcal{A}$ is a 1-way 1-dim n-head machine operating such that each head h_i works on a distinct tape written over Σ_i ($i = 1,2, \ldots, n$) then $G(\mathcal{A}) = \beta^{\psi}$ where β is a RE over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_n \end{bmatrix}$

Proof: An effective procedure exists to determine if any $\mathcal{A}$ is 1-way (see Section 6.5). Without any loss of generality we can assume $\mathcal{A}$ to be 1-way in the +1 direction for all heads, in which event all the accessible transitions of $\mathcal{A}$ will carry labels of the form $\sigma_1,\sigma_2, \ldots, \sigma_n/1,1, \ldots, 1$ where $\sigma_i \epsilon \Sigma_i$. One can remove all inaccessible transitions from $\mathcal{A}$ without alterning $G(\mathcal{A})$. Since the heads of $\mathcal{A}$ move in synchronism, one can imagine the input to $\mathcal{A}$ to be either a set of n single channel tapes or a single n-channel tape (or more precisely the separation of a single n-channel tape). If one adopts the latter point of view then the n reading heads $h_1,h_2, \ldots, h_n$ reading over $\Sigma_1,\Sigma_2, \ldots, \Sigma_n$ respectively can be considered as one reading head reading over the alphabet $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_n \end{bmatrix}$.

Therefore, via Theorem 4.1, β, the set of single n-channel generators accepted by the 1-head machine reading over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_n \end{bmatrix}$ would be expressible as a RE over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_n \end{bmatrix}$. Taking the separation of β one gets $G(\mathcal{A})$. i.e., $G(\mathcal{A}) = \beta^{\psi}$.

QED

Example 4.3 Let $\mathcal{M}_{4.3}$ be the 1-way 1-dim 2-head machine given in Figure 84. Head h_1 works on tapes written over $\Sigma_1 = \{B,0,1\}$ and h_2 works on tapes written over $\Sigma_2 = \{B,a,1\}$ Find $G(\mathcal{M}_{4.3})$.



Machine $\mathcal{M}_{4.3}$

Figure 84

Considering $\mathcal{M}_{4.3}$ to be 1-head reading over $\begin{bmatrix}\Sigma_1\\ \Sigma_2\end{bmatrix}$ one finds via theorem 4.1 that

$$\beta = \left\{\begin{bmatrix}B\\B\end{bmatrix} \cup \begin{bmatrix}B\\1\end{bmatrix} \begin{bmatrix}B\\a\end{bmatrix} * \begin{bmatrix}1\\a\end{bmatrix}\right\}\left\{\begin{bmatrix}B\\B\end{bmatrix} \cup \begin{bmatrix}B\\a\end{bmatrix}\right\}$$

and that

$$G(\mathcal{M}_{4.3}) = \beta^{\Psi} = \left[\left\{\begin{bmatrix}B\\B\end{bmatrix} \cup \begin{bmatrix}B\\1\end{bmatrix} \begin{bmatrix}B\\a\end{bmatrix} * \begin{bmatrix}1\\a\end{bmatrix}\right\}\left\{\begin{bmatrix}B\\B\end{bmatrix} \cup \begin{bmatrix}B\\a\end{bmatrix}\right\}\right]^{\Psi}$$

Theorem 4.4 If β is a RE over $\begin{bmatrix}\Sigma_1\\ \Sigma_2\\ \vdots\\ \Sigma_n\end{bmatrix}$ then there exists a 1-way 1-dim

n-head machine $\mathcal{A}$ in which each head h_i reads over Σ_i and for which $G(\mathcal{A}) =. \beta^{\psi}$.

Proof: Via the method of Theorem 4.2 construct $\mathcal{A}''$ a 1-way 1-dim 1-head machine reading over $\begin{bmatrix}\Sigma_1\\ \Sigma_2\\ \vdots\\ \Sigma_n\end{bmatrix}$ which has $G(\mathcal{A}'') = \beta$. Each transition of $\mathcal{A}''$ will be labelled $\begin{bmatrix}\sigma_1\\ \sigma_2\\ \vdots\\ \sigma_n\end{bmatrix}\Big/1$. Convert $\mathcal{A}''$ to a 1-way 1-dim n-head machine by changing each transition label of $\mathcal{A}''$ as follows:

$$\begin{bmatrix}\sigma_1\\ \sigma_2\\ \vdots\\ \sigma_n\end{bmatrix}\Big/1 \longrightarrow (\sigma_1,\sigma_2, \ldots, \sigma_n)/(1,1, \ldots, 1)$$

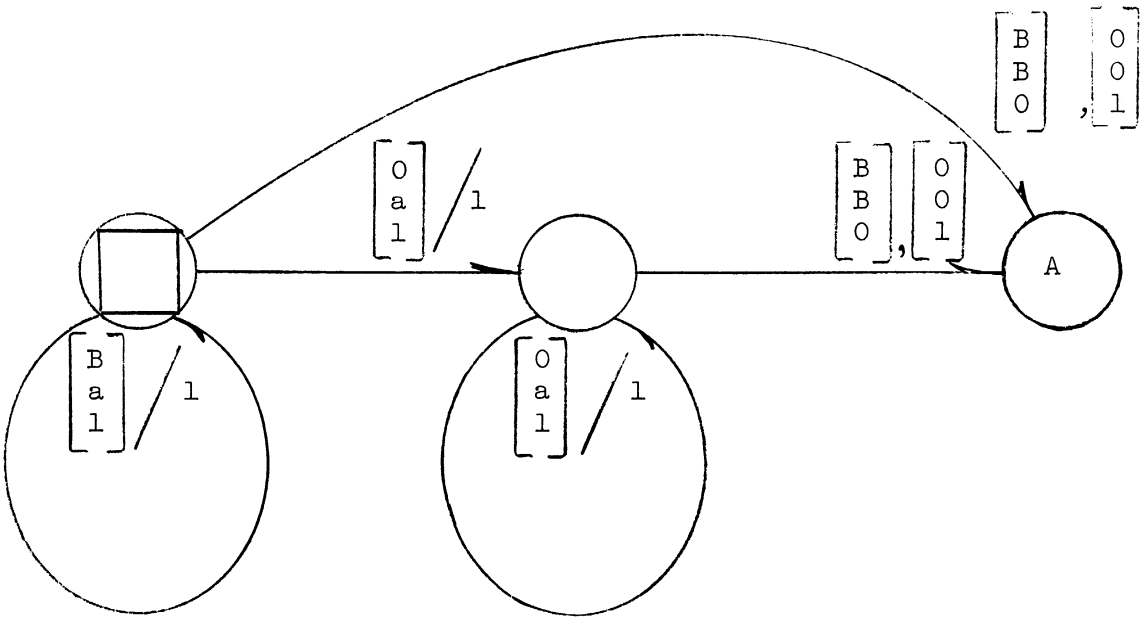
The resulting machine $\mathcal{A}$ has as generators precisely the separation of $G(\mathcal{A}'')$; i.e., $G(\mathcal{A}) = G(\mathcal{A}'')^{\psi} = \beta^{\psi}$.

QED

Example 4.4 Given

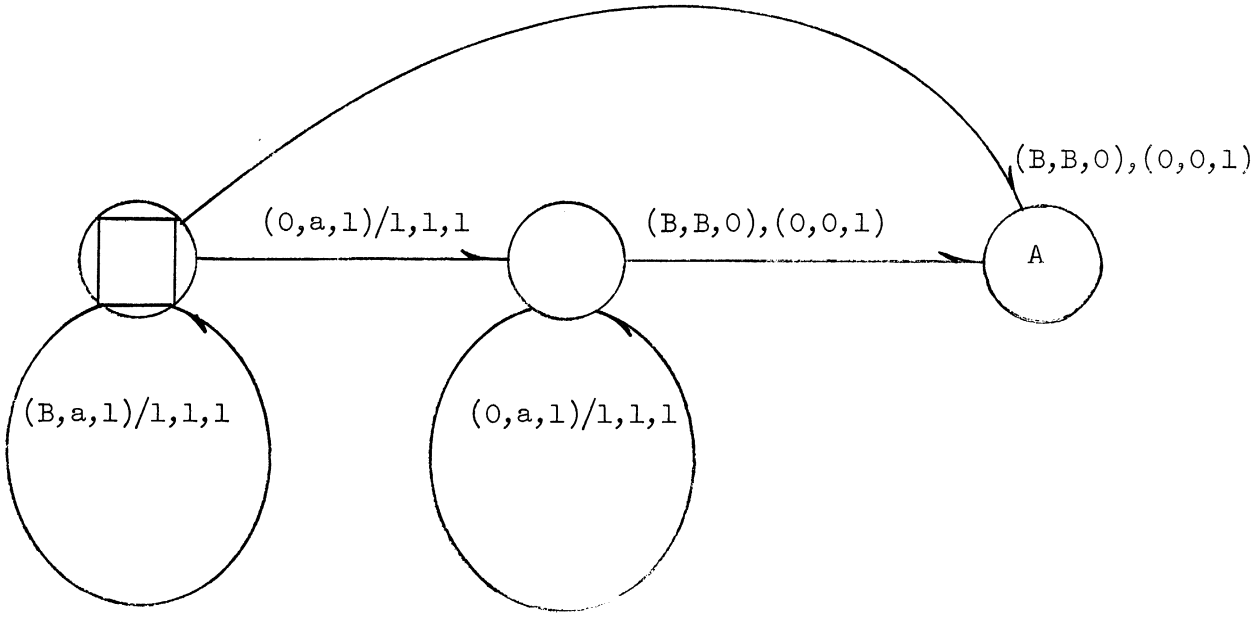
$$\beta = \begin{bmatrix}B\\ a\\ 1\end{bmatrix} * \begin{bmatrix}0\\ a\\ 1\end{bmatrix} * \left\{\begin{bmatrix}B\\ B\\ 0\end{bmatrix} \cup \begin{bmatrix}0\\ 0\\ 1\end{bmatrix}\right\}$$

construct machine $\mathcal{A}_{4.4}$ such that $G(\mathcal{A}_{4.4}) = \beta^{\psi}$. Using the method presented in Theorem 4.4 one first derives the machine $\mathcal{A}''_{4.4}$ (Figure 85) Applying the mapping $\begin{bmatrix}\sigma_1\\ \sigma_2\\ \sigma_3\end{bmatrix}\Big/1 \longrightarrow (\sigma_1,\sigma_2,\sigma_3)/1,1,1$ to the transition labels of $\mathcal{A}''_{4.4}$ one gets the desired machine $\mathcal{A}_{4.4}$ (Figure 86).



Machine $\mathcal{A}''_{4.4}$

Figure 85



Machine $\mathcal{A}_{4.4}$

Figure 86

Note 4.2 If $\mathcal{A}$ is a 1-way 1-dim n-head machine working on m tapes ($m \leq n$).then some tapes can have more than one head per tape. If any two heads h_i and h_j are on the same tape and move in the same direction then their positions will always coincide and they can be replaced by a single head; if such is the case for all heads on each tape then $\mathcal{A}$ can be replaced by a 1-way 1-dim m-head machine $\mathcal{A}''$ that is equivalent to $\mathcal{A}$ (i.e. $G(\mathcal{A}) = G(\mathcal{A}'') = \beta^{\Psi}$ where β is a RE with m channels).

6.4.3 2-Way 1-Dim 1-Head Machines

Def. 4.1 Let β be a RE over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$; β will be said to be realizable if and only if β or any of its equivalent RE's has no well-formed part of the form

$$\begin{bmatrix} \sigma_{\alpha_1}/d_{\delta_1} \\ \sigma_{\alpha_2}/d_{\delta_2} \\ \vdots \\ \sigma_{\alpha_n}/d_{\delta_n} \end{bmatrix} \quad A_1 \quad \bigcup \quad \begin{bmatrix} \sigma_{\alpha_1}/d_{\gamma_1} \\ \sigma_{\alpha_2}/d_{\gamma_2} \\ \vdots \\ \sigma_{\alpha_n}/d_{\gamma_2} \end{bmatrix} \quad A_2$$

where for some $i = 1,2, \ldots, n$ $\quad d_{\delta_i} \neq d_{\gamma_i}$ (A_1 and A_2 are sets of partial tapes over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$).

Theorem 4.5 If $\mathcal{A}$ is a 2-way 1-dim 1-head machine working on tapes written over Σ then $G(\mathcal{A}) = \beta^f$ where β is a realizable RE over $\Sigma/1$.

Proof: Let $\mathcal{A}'$ be derived from $\mathcal{A}$ by considering $\mathcal{A}$ to be a 1-way 1-dim 1-head machine reading over $\Sigma/1$ with the head movement of +1 for all transitions of $\mathcal{A}'$ understood. β is the RE over $\Sigma/1$ representing $G(\mathcal{A}')$. From the fact that for a given state in $\mathcal{A}'$ and for each $\sigma\epsilon\Sigma$ there is only one transition in $\mathcal{A}'$ one deduces that β is realizable; the assumption that β is not realizable would imply that $\mathcal{A}$ has a state with two transitions for the same input -- this is not allowed.

Let $t\epsilon T(\mathcal{A})$. The behavior of $\mathcal{A}$ on $g_{\mathcal{A}}(t)$ can be described by the sequence

$$\rho = s^I,\ \sigma_o/d_o;\ \ s_1,\sigma_1/d_1;\ \ldots\ldots;\ s_{p-1},\sigma_{p-1}/d_{p-1};\ s_p,\sigma_p.$$

where $\mathcal{A}$ starts in state s^I, reads σ_o (in cell o) moves its head d_o and goes to state s_1;..... $\mathcal{A}$ in state s_i during the i-th cycle reads σ_i (not necessarily in cell i) moves its head d_i and goes to state s_{i+1};.... $\mathcal{A}$ in state s_p during the p-th cycle reads σ_p and goes to A ($\mathcal{A}$ accepts $g_{\mathcal{A}}(t)$). Consider the partial tape

$$t' = \boxed{\sigma_o/d_o}\ \boxed{\sigma_1/d_1}\ \boxed{\ldots\ldots\ldots}\ \boxed{\sigma_p}$$

extracted from ρ. Since t' derived from the functioning of $\mathcal{A}$ on t it follows that σ_o/d_o is an initial symbol of β, σ_p a final symbol of β and $(\sigma_i/d_i,\ \sigma_{i+1}/d_{i+1})$ a transition of β for i = 1,2, ..., p-1. Thus $t'\epsilon\beta$. The definition of the fold operator exactly parallels the head movement of $\mathcal{A}$ so that $t'^f = g_{\mathcal{A}}(t)$. But $t'\epsilon\beta \rightarrow t'^f\epsilon\beta^f$ so that $g_{\mathcal{A}}(t) = t'^f\epsilon\beta^f$. $\therefore$ one has the partial proof $g\epsilon\ G(\mathcal{A}) \rightarrow g\epsilon\beta^f$.

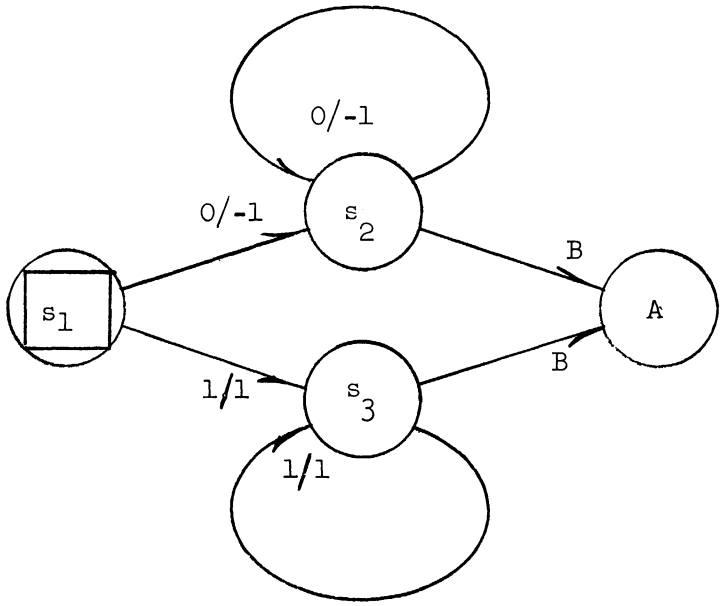
To complete the proof one must show that $g \epsilon \beta^f \rightarrow g \epsilon G(\mathcal{A})$. Take any g in β^f. Therefore there is some t' in β such that $t'^f = g$. t' is in β therefore $t' \epsilon G(\mathcal{A}')$. Write the sequence

$$\rho = s^I, \sigma_o/d_o; \quad s_1, \sigma_1/d_1; \ldots; s_p, \sigma_p.$$

that describes the behavior of $\mathcal{A}'$ on t' = $\boxed{\sigma_o/d_o}\boxed{\sigma_1/d_1}\boxed{\cdots}\boxed{\sigma_p}$; $\mathcal{A}'$ starts in s^I, reads σ_o/d_o of t', goes to state s_1, reads σ_1/d_1, goes to s_2, ..., goes to s_p, reads σ_p, goes to A. But if $\mathcal{A}'$ accepts t' then $\mathcal{A}$ accepts $t'^f = g$ since the fold operation parallels the head movement of $\mathcal{A}$. Thus $g \epsilon \beta^f$, which completes the proof.

QED

Example 4.5 Let $\mathcal{A}_{4.5}$, the 2-way 1-dim 1-head machine working on tapes written over $\Sigma = \{B,0,1\}$, be shown in Figure 87. Find $G(\mathcal{A})$.



Machine $\mathcal{A}_{4.5}$

Figure 87

From $\mathcal{A}_{4.5}'$ one gets $\beta = (0/-1)(0/-1)^*B \cup (1/1)(1/1)^*B$ or that $G(\mathcal{A}_{4.5}) = [(0/-1)(0/-1)^*B \cup (1/1)(1/1)^*B]^f$. Observe that β is realizable.

Theorem 4.6 If β is a realizable RE over $\Sigma/1$ then there exists a 2-way 1-dim 1-head machine $\mathcal{A}$ such that $G(\mathcal{A}) = \beta^f$.

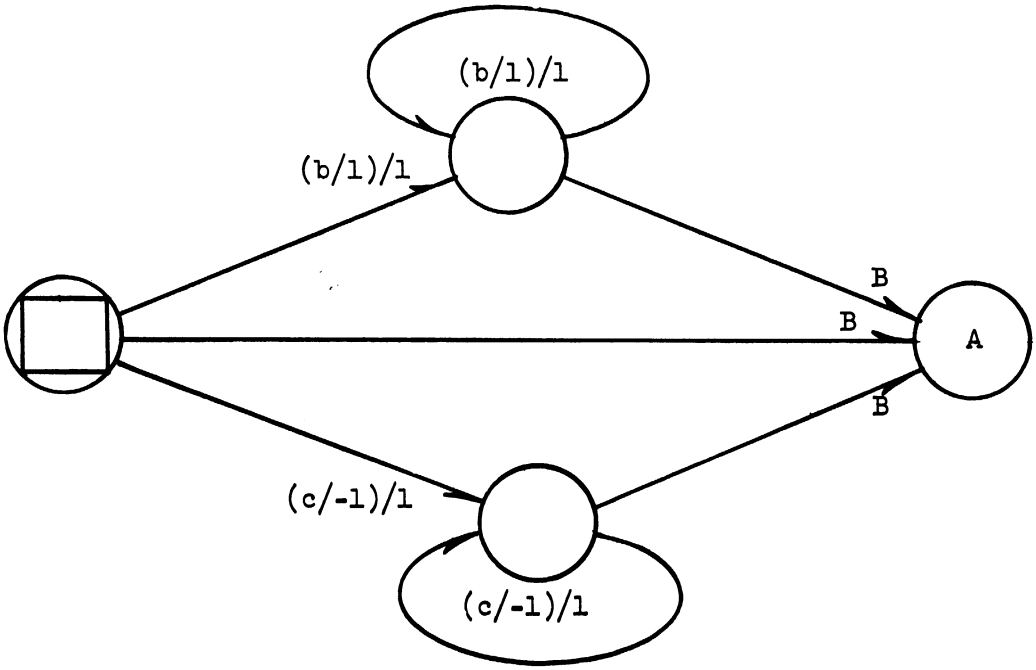
Proof: An effective method exists to determine if β is realizable (see Chapter V). Construct via Theorem 4.2 the machine $\mathcal{A}'$ that reads over $\Sigma/1$ and has $G(\mathcal{A}') = \beta$. Since β is realizable we are assured that for each $\sigma\epsilon\Sigma$ and each state of $\mathcal{A}'$, $\mathcal{A}'$ will have just one transition. Thus if we convert $\mathcal{A}'$ to a 2-way 1-dim 1-head machine $\mathcal{A}$ reading over Σ by applying to the transition labels of $\mathcal{A}'$ the mapping $(\sigma/d)/1 \Rightarrow \sigma/d$ we are assured that $\mathcal{A}$ is in legitimate form (i.e. only one transition leaving each state for each input). The proof follows by reversing the arguments of the proof of Theorem 4.5.

QED

Example 4.6 Let β equal the realizable RE $(b/1)^*B \cup (c/-1)^*B$. Find a 2-way 1-dim 1-head machine $\mathcal{A}_{4.6}$ such that $G(\mathcal{A}_{4.6}) = \beta^f$.

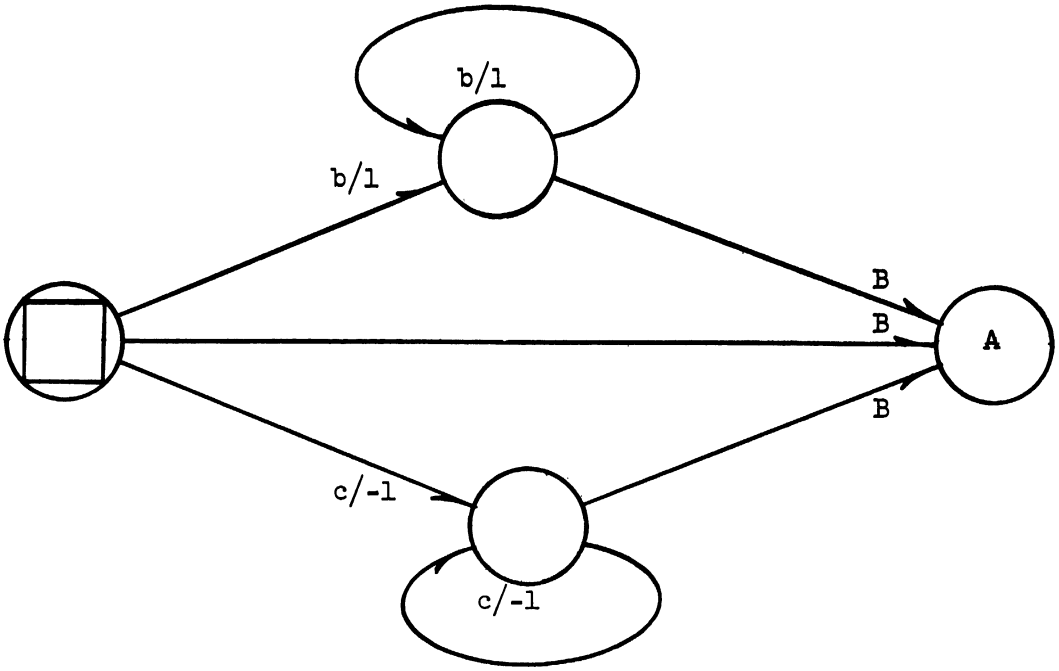
$\mathcal{A}_{4.6}'$, the 1-way 1-dim 1-head machine reading over $\{B,b,c\}/1$ that satisfies $G(\mathcal{A}_{4.6}') = \beta$ is computed via Theorem 4.2 and is given in Figure 88.

The machine $\mathcal{A}_{4.6}$ which satisfies $G(\mathcal{A}_{4.6}) = \beta^f$ is obtained from $\mathcal{A}_{4.6}'$ by applying the mapping $(\sigma/d)/1 \Rightarrow \sigma/d$ to all transition labels in $\mathcal{A}_{4.6}'$. $\mathcal{A}_{4.6}$ is given in Figure 89.



Machine $\mathcal{M}'_{4.6}$

Figure 88



Machine $\mathcal{M}_{4.6}$

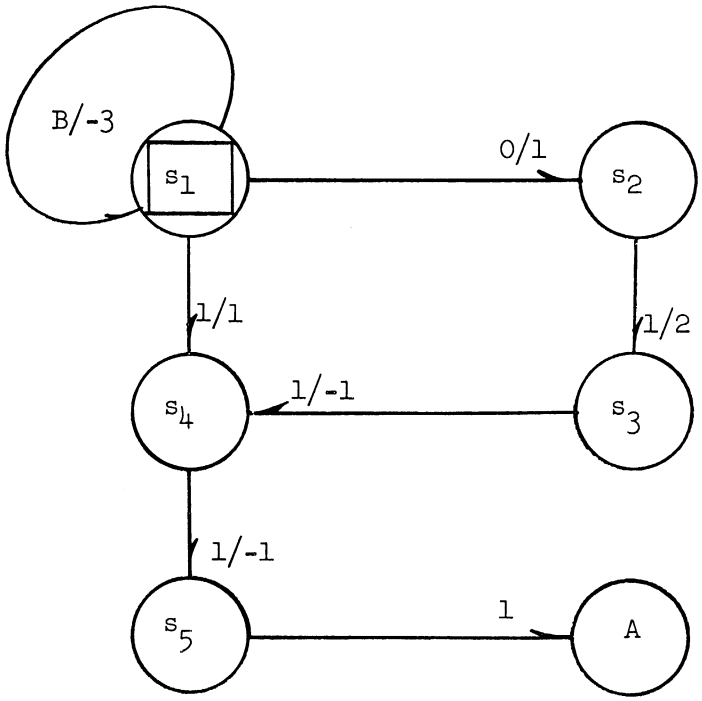
Figure 89

6.4.4 2-Way D-Dim 1-Head Machines

Theorems 4.5 and 4.6 can be immediately extended to 1-head machines working over D-dim tapes; the proofs are essentially the same as in Theorems 4.5 and 4.6, differing only in those places where the head movement goes to D-dimensions. The D-dim theorems are given below without proofs but with examples.

Theorem 4.7 If $\mathcal{M}$ is a 2-way D-dim 1-head machine working on tapes written over Σ then $G(\mathcal{M}) = \beta^f$ where β is a realizable RE over Σ/D.

Example 4.7 $\mathcal{M}_{4.7}$ shown in Figure 90 is a 2-way 3-dim 1-head machine working on tapes written over $\Sigma = \{B, 0, 1\}$. $G(\mathcal{M}_{4.7})$ is derived to be $\{(B/-3)*[(0/1)(1/2)(1/-1)\cup(1/1)](1/-1)1\}^f$.



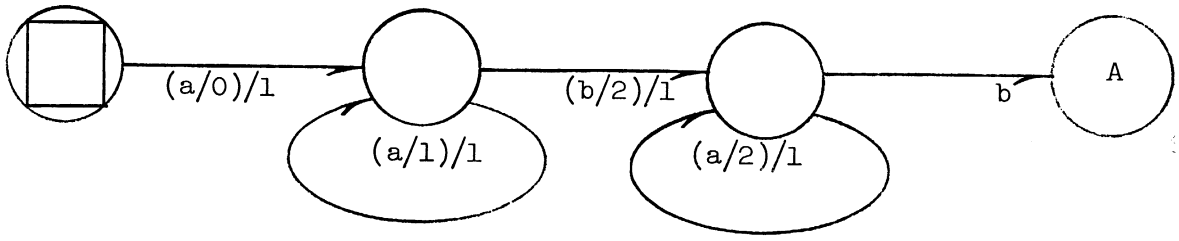
Machine $\mathcal{M}_{4.7}$

Figure 90

Theorem 4.8 If β is a realizable RE over Σ/D then there exists a 2-way D-dim 1-head machine $\mathcal{M}$ such that $G(\mathcal{M}) = \beta^f$.

Example 4.8 Construct a 2-way 2-dim 1-head machine $\mathcal{M}_{4.8}$ such that $G(\mathcal{M}_{4.8}) = \beta^f$ when $\beta = (a/0)(a/1)^*(b/2)(a/2)^*b$.

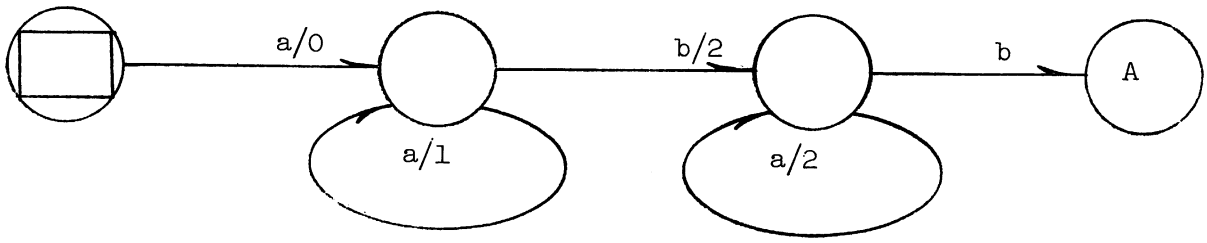
$\mathcal{M}'_{4.8}$, the 1-way 1-dim 1-head machine that reads over $\{a,b\}/2$ and for which $G(\mathcal{M}'_{4.8}) = \beta$, is computed via Theorem 4.2 and is given in Figure 91.



Machine $\mathcal{M}'_{4.8}$

Figure 91

The machine $\mathcal{M}_{4.8}$ which satisfies $G(\mathcal{M}_{4.8}) = \beta^f$ is obtained from $\mathcal{M}'_{4.8}$ by applying the mapping $(\sigma/d)/1 \Rightarrow \sigma/d$ to all transition labels in $\mathcal{M}'_{4.8}$. $\mathcal{M}_{4.8}$ is given in Figure 92.



Machine $\mathcal{M}_{4.8}$

Figure 92

6.4.5 2-Way D-Dim n-Head n-Tape Machines

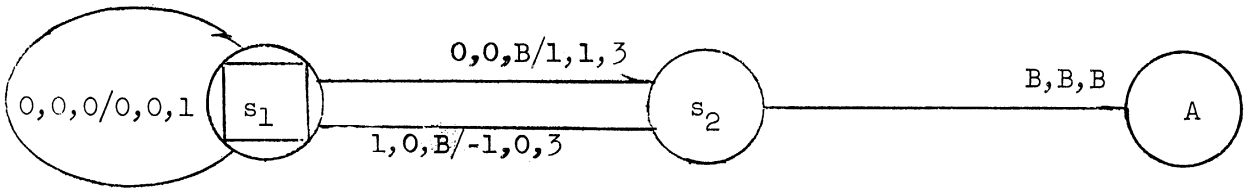
Theorem 4.9 If $\mathcal{M}$ is a 2-way n-head machine with each head $h_i (i = 1,2,\ldots,n)$ working on a distinct tape of dimension D_i and written over Σ_i then $G(\mathcal{M}) = \beta^{\psi f}$ where β is a realizable RE over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$.

Proof: The RE β is obtained by applying the mapping $(\sigma_1,\sigma_2,\ldots,\sigma_n)(d_1,d_2,\ldots,d_n) \Rightarrow (\sigma_1/d_1, \sigma_2/d_2, \ldots, \sigma_n/d_n)/1,1,\ldots,1$ to each transition label of $\mathcal{M}$ thereby obtaining a 1-way n-head machine $\mathcal{M}'$ whose heads read respectively over $\Sigma_1/D_1, \Sigma_2/D_2, \ldots, \Sigma_n/D_n$; let $\beta^{\psi} = G(\mathcal{M}')$. Arguing as in the proof of Theorem 4.5 if t' is an n-tuple in β^{ψ} then $\mathcal{M}'$ accepts t';and if $t'^{f} \neq \emptyset$ then $\mathcal{M}$ when working on t'^{f} will go through the same sequence of states as $\mathcal{M}'$ and therefore t'^{f} is accepted by $\mathcal{M}$ (or $t'^{f} \in G(\mathcal{M})$). Thus $\beta^{\psi f} \subseteq G(\mathcal{M})$. Conversely if t is some input in $G(\mathcal{M})$ then by examining the behavior of $\mathcal{M}$ in accepting t we can deduce the sequence $t' \in \beta$ such that $t'^{f} = t$. Thus $\beta^{\psi f} \supseteq G(\mathcal{M})$. The conclusion then is that $G(\mathcal{M}) = \beta^{\psi f}$.

That β is realizable follows from the observation that if β were not then one could show $\mathcal{M}$ must have a state with two transitions leaving it for the same input n-tuple; this is not allowed. Therefore β must be realizable.

QED

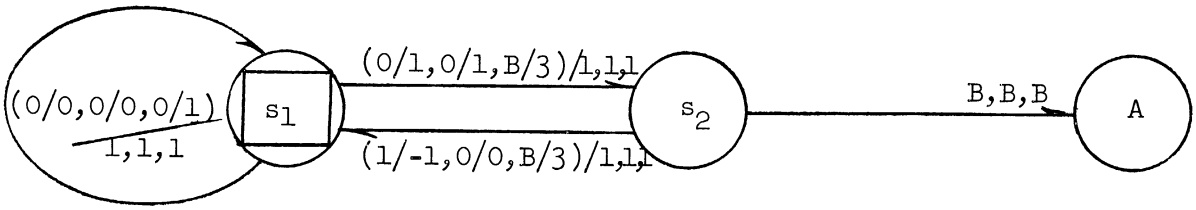
Example 4.9 Let $\mathcal{M}_{4.9}$ be the 2-way 3-head machine shown in Figure 93. Each head of $\mathcal{M}_{4.9}$ works on a distinct tape with $D_1 = 1$, $D_2 = 2$, $D_3 = 3$ and $\Sigma_1 = \Sigma_2 = \Sigma_3 = \{B,0,1\}$. Find $G(\mathcal{M}_{4.9})$.



Machine $\mathcal{A}_{4.9}$

Figure 93

Applying the mapping of Theorem 4.9 one obtains the 1-way n-head machine $\mathcal{A}'_{4.9}$ shown in Figure 94.



Machine $\mathcal{A}'_{4.9}$

Figure 94

Theorem 4.3 applied to $\mathcal{A}'_{4.9}$ yields $G(\mathcal{A}'_{4.9}) = \beta^{\psi}$ where

$$\beta = \left\{ \begin{bmatrix} 0/1 \\ 0/1 \\ B/3 \end{bmatrix} \begin{bmatrix} 1/-1 \\ 0/0 \\ B/3 \end{bmatrix} \cup \begin{bmatrix} 0/0 \\ 0/0 \\ 0/1 \end{bmatrix} \right\} * \begin{bmatrix} 0/1 \\ 0/1 \\ B/3 \end{bmatrix} \begin{bmatrix} B \\ B \\ B \end{bmatrix}$$

Thus

$$G(\mathcal{A}_{4.9}) = \beta^{\psi f} = \left[\left\{ \begin{bmatrix} 0/1 \\ 0/1 \\ B/3 \end{bmatrix} \begin{bmatrix} 1/-1 \\ 0/0 \\ B/3 \end{bmatrix} \cup \begin{bmatrix} 0/0 \\ 0/0 \\ 0/1 \end{bmatrix} \right\} * \begin{bmatrix} 0/1 \\ 0/1 \\ B/3 \end{bmatrix} \begin{bmatrix} B \\ B \\ B \end{bmatrix} \right]^{\psi f}$$

Theorem 4.10 If β is a realizable RE over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$

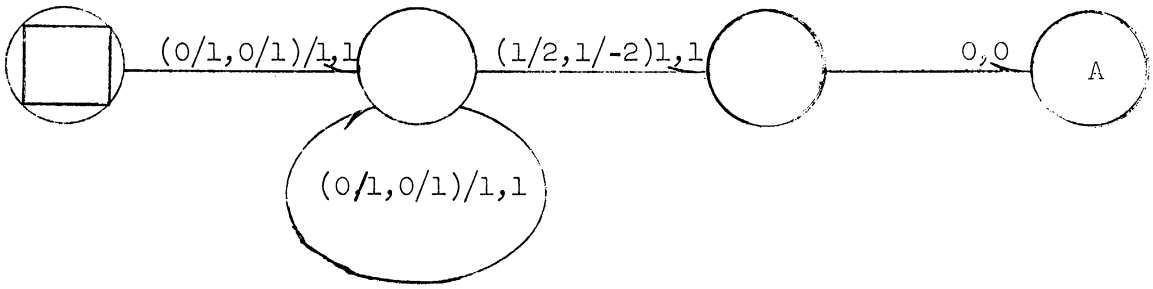
then there exists a 2-way n-head machine $\mathcal{A}$ such that $G(\mathcal{A}) = \beta^{\psi f}$

Proof: Construct via Theorem 4.2 the machine $\mathcal{A}'$ that reads over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$ and has $G(\mathcal{A}') = \beta^{\psi}$. Obtain $\mathcal{A}$ from $\mathcal{A}'$ by applying the mapping $(\sigma_1/d_1, \sigma_2/d_2, \ldots, \sigma_n/d_n)/1,1, \ldots, 1 \rightarrow (\sigma_1,\sigma_2, \ldots, \sigma_n)/d_1,d_2,\ldots d$ to all transition labels of $\mathcal{A}'$. Since β is realizable we are assured that will have only one transition leaving each state for each input n-tuple. The proof follows by reversing the arguments of the proof of Theorem 4.9.

QED

Example 4.10 Construct a 2-way 2-head machine $\mathcal{A}_{4.10}$ such that $G(\mathcal{A}_{4.10}) = \beta^{\psi f}$ where $\beta = \begin{bmatrix} 0/1 \\ 0/1 \end{bmatrix} \begin{bmatrix} 0/1 \\ 0/1 \end{bmatrix} * \begin{bmatrix} 1/2 \\ 1/-2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

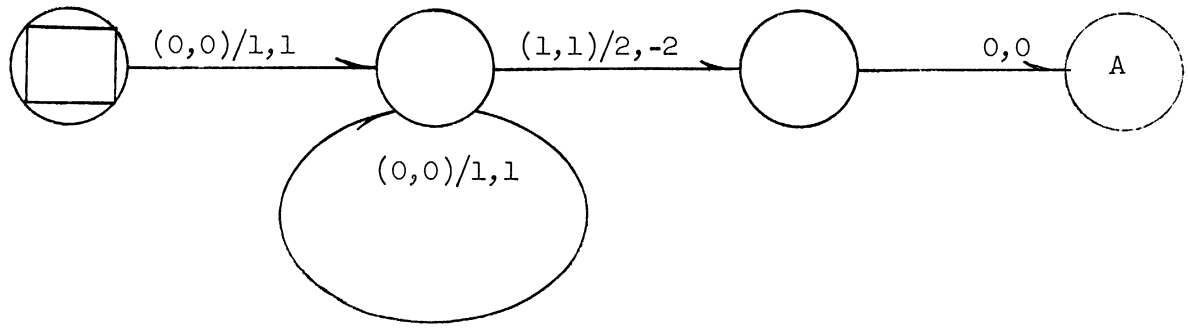
$\mathcal{A}'_{4.10}$, the machine with $G(\mathcal{A}'_{4.10}) = \beta^{\psi}$, is shown in Figure 95.



Machine $\mathcal{A}'_{4.10}$

Figure 95

Applying the mapping of Theorem 4.10 to $\mathcal{A}'_{4.10}$ one obtains $\mathcal{A}_{4.10}$ shown in Figure 96.



Machine $\mathcal{A}_{4.10}$

Figure 96

6.4.6 2-Way D-Dim n-Head m-Tape Machines

Theorem 4.11 If $\mathcal{A}$ is a n-head machine operating on m tapes ($m \leq n$) such that the first n_1 heads work on tape t_1 written over Σ_1 in D_1 dimensions, the next n_2 heads work on tape t_2 written over Σ_2 in D_2 dimensions, ..., the last n_m heads work on tape t_m written over Σ_m in D_m dimensions ($n \geq 1; i=1,2,\ldots, m$) then

$$G(\mathcal{A}) = \beta^{\Psi f_{\mathcal{C}}} \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix}$$ where β is a realizable

RE over $\left[\begin{matrix}\Sigma_1/D_1 \\ \Sigma_1/D_1 \\ \vdots \\ \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_{m-1}/D_{m-1} \\ \Sigma_m/D_m \\ \vdots \\ \Sigma_m/D_m\end{matrix}\right.$ $\left.\begin{matrix} \\ \\ n_1 \\ \\ \\ \\ \\ \\ n_m \\ \\ \end{matrix}\right]$

Proof: Let $\mathcal{A}'$ be the same machine as $\mathcal{A}$ but with each head on a distinct tape; then via Theorem 4.9 let β be the realizable RE over

$$\left[\begin{matrix}\Sigma_1/D_1 \\ \vdots \\ \Sigma_1/D_1 \\ \vdots \\ \Sigma_n/D_m \\ \vdots \\ \Sigma_m/D_m\end{matrix}\right.\quad\left.\begin{matrix} \\ n_1 \\ \\ \\ \\ n_m \\ \\ \end{matrix}\right]$$

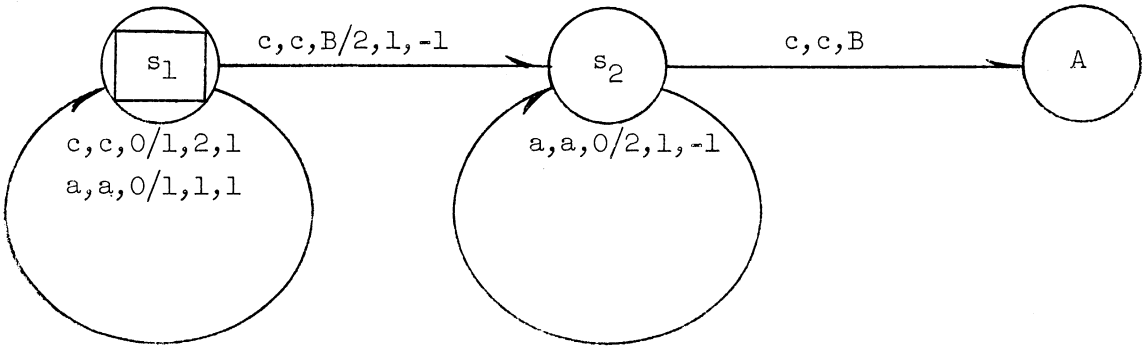
such that $G(\mathcal{A}') = \beta^{\Psi f}$. If $t' = (t'_1, t'_2, \ldots, t'_n) \in \beta^{\Psi f}$ and if $t = t' \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} \neq \emptyset$ then $\mathcal{A}$ when working on t will go through the same state sequence of states as $\mathcal{A}'$ working on t'. Since every filled cell of t is scanned by $\mathcal{A}$ (since every filled cell of t' is scanned by $\mathcal{A}'$) and t is accepted by $\mathcal{A}$, $t \in G(\mathcal{A})$ or in other words $\beta^{\Psi f} \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} \subseteq G(\mathcal{A})$. Conversely if $t = (t_1, t_2, \ldots, t_m) \in G(\mathcal{A})$ then there is an n-tuple $t' = (t'_1, t'_2, \ldots, t'_n)$ where t'_i is the generator scanned by

by the i-th head of $\mathcal{M}$. t' must exist in $\beta^{\psi f}$ and $t' \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix}$ must equal t. Thus $\beta^{\psi f} \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} \supseteq G(\mathcal{M})$.

Concluding then, $\beta^{\psi f} \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix} = G(\mathcal{M})$.

QED

Example 4.11 Let $\mathcal{M}_{4.11}$ be the 3-head machine shown in Figure 97 Heads h_1 and h_2 work on the same tape of dimension 2 written over $\Sigma_1 = \{B,a,c\}$; head h_3 works on a tape of dimension 1 written over $\Sigma_2 = \{B,0\}$. Find $G(\mathcal{M}_{4.11})$.



Machine $\mathcal{M}_{4.11}$

Figure 97

Applying Theorem 4.9 one obtains $\beta = \left\{ \begin{bmatrix} c/1 \\ c/2 \\ 0/1 \end{bmatrix} \cup \begin{bmatrix} a/1 \\ a/1 \\ 0/1 \end{bmatrix} \right\}^* \begin{bmatrix} c/2 \\ c/1 \\ B/-1 \end{bmatrix} \begin{bmatrix} a/2 \\ a/1 \\ 0/-1 \end{bmatrix} \begin{bmatrix} c \\ c \\ B \end{bmatrix}$

and thus $G(\mathcal{M}_{4.11}) = \beta^{\psi f} \odot \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} = \left\{ \begin{bmatrix} c/1 \\ c/2 \\ 0/1 \end{bmatrix} \cup \begin{bmatrix} a/1 \\ a/1 \\ 0/1 \end{bmatrix} \right\}^* \begin{bmatrix} c/2 \\ c/1 \\ B/-1 \end{bmatrix} \begin{bmatrix} a/2 \\ a/1 \\ 0/-1 \end{bmatrix} \begin{bmatrix} c \\ c \\ B \end{bmatrix}^{\psi f} \odot \begin{bmatrix} 2 \\ 1 \end{bmatrix}$

Theorem 4.12 If β is a realizable RE over $\begin{bmatrix} \Sigma_1/D_1 \\ \vdots \\ \Sigma_1/D_1 \\ \vdots \\ \Sigma_m/D_m \\ \vdots \\ \Sigma_m/D_m \end{bmatrix} \begin{matrix} \} n_1 \\ \\ \\ \\ \} n_m \end{matrix}$

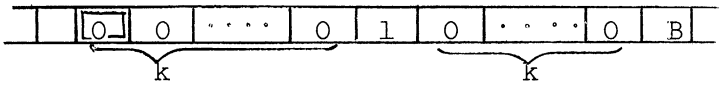
Then there exists a 2-way n-head machine $\mathcal{A}$, $(n = \sum_{i=1}^{m} n_i)$, such that

$$G(\mathcal{A}) = \beta^{\psi f} \odot \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_m \end{bmatrix}$$

Proof: Let $\mathcal{A}$ be the machine obtained by applying Theorem 4.10 to β (i.e. $G(\mathcal{A}) = \beta^{\psi f}$). Instead of letting $\mathcal{A}$ operate with one head per tape alter $\mathcal{A}$ such that the first n_1 heads of $\mathcal{A}$ operate on a single tape t_1, the next n_2 heads operate on a single tape t_2, ..., the last n_m heads operate on a single tape t_m. The proof is completed by reversing the arguments of Theorem 4.11.

QED

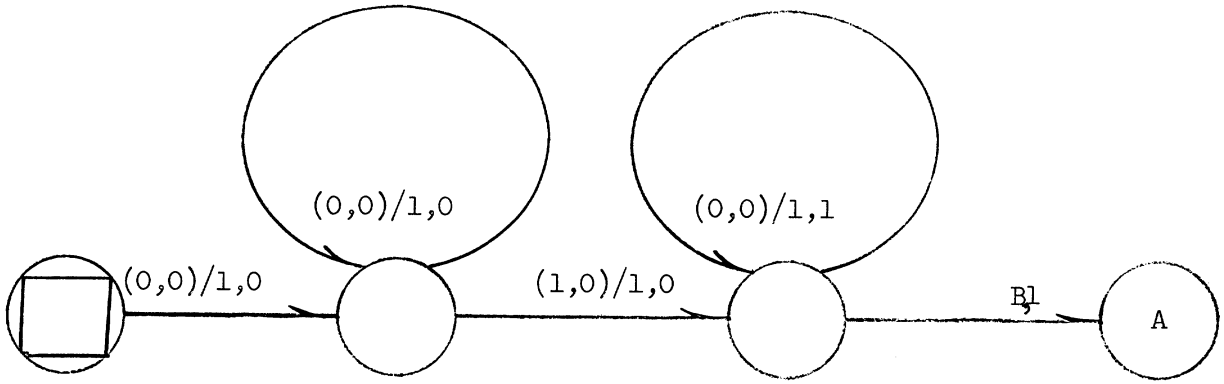
Example 4.12 Construct a machine $\mathcal{A}_{4.12}$ that works on 1-dim tapes over $\Sigma = \{B,0,1\}$ and such that

$G(\mathcal{A}) = \{t \mid t =$ 

, k = 1,2,.

One can show that $G(\mathcal{A})_{4.12} = \begin{bmatrix} 0/1 \\ 0/0 \end{bmatrix} \begin{bmatrix} 0/1 \\ 0/0 \end{bmatrix} * \begin{bmatrix} 1/1 \\ 0/0 \end{bmatrix} \begin{bmatrix} 0/1 \\ 0/1 \end{bmatrix} * \begin{bmatrix} B \\ 1 \end{bmatrix} \odot [2]$

Thus $\mathcal{A}_{4.12}$ is the 2-head machine shown in Figure 98 with both heads working on the same tape.



Machine $\mathfrak{A}_{4.12}$

Figure 98

6.5 ASSORTED ALGORITHMS AND THEOREMS DEALING WITH THE DECISION PROBLEMS AND SPEED OF OPERATION OF n-HEAD MACHINES

6.5.1 Algorithm for Deciding 1-Wayness of Machines

The algorithm will be given below assuming that the machine $\mathcal{M}$ under consideration is n-head working on n-tapes (i.e., one head per tape); the remarks following the presentation of the algorithm indicate how the method may be extended to include machines with more than one head per tape.

Algorithm 5.1 Let $\mathcal{M}$ be an n-head machine working on n-tapes (one-head per tape) and let the state set of $\mathcal{M}$ be $S \cup \{A,R\}$ with $s^I \in S$. The transitions of $\mathcal{M}$ going to A or R will be assumed to cause no head motion of $\mathcal{M}$.

1) Let $i = 0$ and $\mathcal{S}(0) = \{s^I\}$.

2) Pick any transition leaving s^I and not going to A or R; let the head motion associated with this transition be the n-tuple $d = (d_1, d_2, \ldots, d_n)$. If no such transition exists $\mathcal{M}$ is trivially 1-way (i.e. $\mathcal{M}$ never moves since all transitions from s^I go to A or R).

3) Consider all transitions leaving states in $\mathcal{S}(i)$, all these transitions must either go to A or R or must have head movement n-tuples equal to d. If this criterion is not met $\mathcal{M}$ is not 1-way. If it is met let

$$\mathcal{S}(i+1) = \mathcal{S}(i) \cup \{\text{all destination states of transitions leaving states in } \mathcal{S}(i)\}.$$

4) If $\mathcal{S}(i+1) = \mathcal{S}(i)$ halt; $\mathcal{A}$ is 1-way; if $\mathcal{S}(i+1) \supset \mathcal{S}(i)$ augment i by 1 and go to step 3).

Proof: First of all, transitions leaving s^I are accessible since any symbol can be put in the initial cell of each tape. If $\mathcal{A}$ is to be 1-way all transitions leaving s^I therefore must go either to A or R or else have the same movement n-tuple d. If in the application of the algorithm $\mathcal{A}$ has not been disqualified as a 1-way machine after i repetitions of step 3) then we know for all inputs to $\mathcal{A}$, $\mathcal{A}$ either accepts or rejects the input or else has moved to some state $s_j (\neq A,R)$ the heads of $\mathcal{A}$ always moving d each machine cycle and thus after i cycles each head of $\mathcal{A}$ is scanning a previously unscanned cell and so all transitions leaving $\mathcal{S}(i)$ are accessible and therefore must go to A or R or also have head movement d. If a transition from $\mathcal{S}(i)$ has a head movement not equal to d there is an input to $\mathcal{A}$ for which $\mathcal{A}$ is not 1-way. The algorithm halts in at most $\overline{\overline{S}} + 2$ repetitions of step 3) since $S \cup \{A,R\} \supseteq \mathcal{S}(i+1) \supseteq \mathcal{S}(i)$.

QED

Note 5.1 Let $\mathcal{S} = \mathcal{S}(i)$ where $\mathcal{S}(i) = \mathcal{S}(i+1)$ in algorithm 5.1; $\mathcal{S}$ is then the set of accessible states of $\mathcal{A}$. $T(\mathcal{A}) = \emptyset$ if and only if $A \notin \mathcal{S}$.

Note 5.2 One can extend the algorithm to the case of many heads per tape by implementing the following step:

If two (or more heads), h_1 and h_2, of $\mathcal{M}$ work on the same tape then during the first machine cycle of $\mathcal{M}$ we only need to consider those transitions from s^I in which h_1 and h_2 read the same symbols; if the d associated with any one of these transitions indicates that h_1 and h_2 move in the same direction then in applying the algorithm one observes that transitions leaving $\mathcal{S}(i)$ are accessible if and only if h_1 and h_2 read the same symbols (assuming $\mathcal{M}$ is 1-way) - therefore transitions leaving $\mathcal{S}(i)$ and in which h_1 and h_2 read different symbols can be considered inaccessible and can be ignored in applying the algorithm. If the d associated with the transitions leaving s^I indicate that h_1 and h_2 move in different directions then for all states in $\mathcal{S}(i) - \{s^I\}$ transitions for which h_1 and h_2 read different symbols must be considered accessible -- furthermore if for some $\mathcal{S}(i)$ there is a transition leaving a state in $\mathcal{S}(i)$ and returning to s^I then all transitions leaving s^I and for which h_1 and h_2 read different symbols must be considered as now being accessible.

6.5.2 Algorithm for Deciding the Realizability of Regular Expressions

Algorithm 5.2 Let β be a RE over $\begin{bmatrix} \Sigma_1/D_1 \\ \Sigma_2/D_2 \\ \vdots \\ \Sigma_n/D_n \end{bmatrix}$

To check if β is realizable, attempt to construct via Theorem 4.10 an n-head machine $\mathcal{M}$ such that $G(\mathcal{M}) = \beta^{\psi f}$. When the proposed $\mathcal{M}$ is obtained check each state of $\mathcal{M}$ to see that only one transition per state is labelled with a given input. If the check is unsatisfactory then it follows that β is not realizable. Further, if β was not realizable $\mathcal{M}$ would not pass the check. Therefore β is realizable if and only if $\mathcal{M}$ has one transition per state for each input.

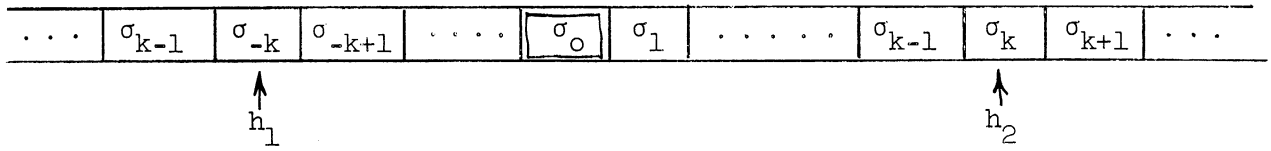
6.5.3 1-Way 2-Head Equivalents of 2-Way 1-Dim 1-Head Machines

Shepherdson[7] has shown that for any 2-way 1-dim 1-head machine $\mathcal{M}$ if one restricts the inputs of $\mathcal{M}$ to those 1-dim tapes for which $\mathcal{M}$ never scans cells to the left of cell 0 then there is a 1-way 1-dim 1-head machine which is equivalent to $\mathcal{M}$. It is impossible in general to construct a 1-way 1-dim 1-head machine equivalent to $\mathcal{M}$ for all inputs. One can construct, however, a 2-head machine that is 1-way and equivalent to $\mathcal{M}$.

Theorem 5.1 If $\mathcal{M}$ is any 2-way 1-dim 1-head machine then there exists a 1-way 1-dim 2-head machine, constructable from $\mathcal{M}$ and denoted by $\mathcal{J}(\mathcal{M})$, such that $G(\mathcal{M}) = G(\mathcal{J}(\mathcal{M}))$.

Proof: $\mathcal{J}(\mathcal{M})$ will have two heads h_1 and h_2. Initially h_1 and h_2 will both be placed on the initial cell of the tape to be examined. Once $\mathcal{J}(\mathcal{M})$ is operating h_1 will move one cell per machine cycle in the -1 direction and h_2 will move one cell per machine cycle in the +1 direction; therefore $\mathcal{J}(\mathcal{M})$ will be a 1-way 1-dim 2-head machine.

For all 1-dim tapes the head positions of $\mathcal{J}(\mathcal{A})$ after k machine cycles will be



and the input to $\mathcal{J}(\mathcal{A})$ will be (σ_{-k}, σ_k).

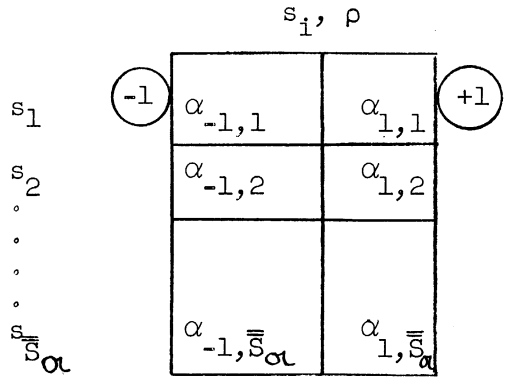
For any tape t let t_k be the subtape of t consisting of the cells -k, -k+1, ..., 0, ... k-1, k. The crux of the construction of $\mathcal{J}(\mathcal{A})$ depends on the observation that given $\mathcal{A} = < C, S_{\mathcal{A}}, s^I, M >$ working on tapes over Σ then for any 1-dim tape t and any integer k, t_k can be put into one of $2 + 2\bar{\bar{S}}_{\mathcal{A}}(2\bar{\bar{S}}_{\mathcal{A}} + 2)^{2\bar{\bar{S}}_{\mathcal{A}}}$ equivalence classes depending on the behavior of $\mathcal{A}$ on t_k. Furthermore, if $[t_k]$ is the equivalence class of t_k and σ_{-k-1} and σ_{k+1} the contents of cells (-k-1) and (k+1) of t then $[t_{k+1}]$ is uniquely determined by $[t_k]$ and $(\sigma_{-k-1}, \sigma_{k+1})$.

The state set of $\mathcal{J}(\mathcal{A})$ is made up precisely of these equivalence classes $[t_k]$, and the transitions of $\mathcal{J}(\mathcal{A})$ on inputs $(\sigma_{-k-1}, \sigma_{k+1}) \in \Sigma \times \Sigma$ are determined as follows:

1) If on reading t_k, $\mathcal{A}$ goes to A(R) then $[t_k] = A(R)$; in the event $\mathcal{A}$ weakly rejects t_k without ever leaving t_k then $[t_k] = R$; thus we have identified two of the equivalence classes, A and R.

2) If on reading t_k, $\mathcal{A}$ does not accept or reject t_k then $\mathcal{A}$ must step off t_k at either the left (-1) or right (+1) end in some state $s_i \in S_{\mathcal{A}}$ If one knew the behavior of $\mathcal{A}$ on t_k if $\mathcal{A}$ started on cell -k and

again on cell k beginning in each state of $\mathcal{O}\ell$ then one could find $[t_{k+\ell}]$ for all $\ell \geq 0$ without knowing precisely what t_k was, i.e., one only need know $[t_k]$. Thus for any t_k, $[t_k]$ can be A, R or a behavior label of the form



where $\rho = \pm 1$ and s_i,ρ denotes that when working on t_k $\mathcal{O}\ell$ steps of the ρ-th end of t_k in state s_i and where $\alpha_{x,y}$ denotes the behavior of $\mathcal{O}\ell$ on t_k if started on the x-th end of t_k in state s_y. $\alpha_{x,y} = A(R)$ if $\mathcal{O}\ell$ moves to A(R) without leaving t_k, $\alpha_{x,y} = R$ if $\mathcal{O}\ell$ weakly rejects t_k without leaving t_k, $\alpha_{x,y} = s_j,\theta$ if $\mathcal{O}\ell$ leaves t_k on the θ-end of t_k in state s_j.

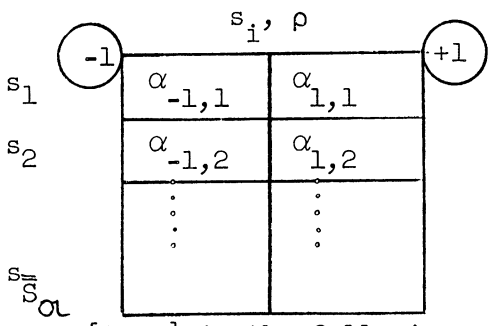
Since for every t_k and a given $\mathcal{O}\ell$ one can put t_k in precisely one of the above mentioned equivalence classes one gets that the number of equivalence classes is

$$\underbrace{2}_{A,R} + \underbrace{2\bar{\bar{S}}_{\mathcal{O}\ell}\,(2 + 2\bar{\bar{S}}_{\mathcal{O}\ell}\,)^{2\bar{\bar{S}}_{\mathcal{O}\ell}}}_{\text{number of behavior labels}}$$

Given $[t_k]$ and $(\sigma_{-k-1}, \sigma_{k+1})$ one can determine $[t_{k+1}]$ as follows:

1) If $[t_k] = A(R)$ then for all $\ell \geq 0$, $[t_{k+\ell}] = A(R)$. This means that if $\mathcal{A}$ accepts (rejects) t_k without reading σ_{-k-1} or σ_{k+1} then $\sigma_{-k-1-\ell}$ and $\sigma_{k+1+\ell}$ can be anything without affecting the behavior of $\mathcal{A}$ or $\mathcal{J}(\mathcal{A})$ on t.

2) If $[t_k] =$



then one determines $[t_{k+1}]$ in the following manner: (assume $\rho = -1$, if $\rho = +1$ just alter the following presentation accordingly).

a) if $\mathcal{A}$ moves to A(R) on reading σ_{-k-1} in state s_i then $[t_x] \xrightarrow{(\sigma_{-k-1}\cdot\omega)} A(R)$; ω indicates that σ_{k+1} can be anything, even a symbol not in Σ, since $\mathcal{A}$ would never read σ_{k+1}; (i.e. cell k+1 is not a filled cell in this particular generator of $\mathcal{A}$ with respect to t).

b) if $\mathcal{A}$ on reading σ_{-k-1} moves back onto t_k in state s_x then consult $\alpha_{-1,x}$ of $[t_k]$ to see what $\mathcal{A}$ would do on t_k:

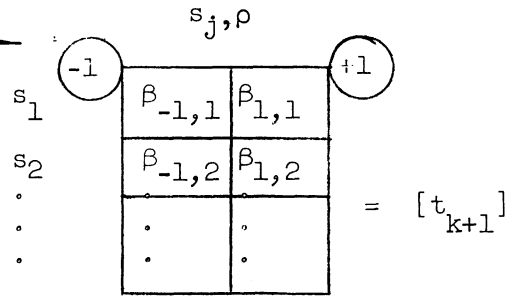
if $\alpha_{-1,x} = A(R)$ then $[t_k] \xrightarrow{(\sigma_{-k-1},\omega)} A(R)$,

if $\alpha_{-1,x} = s_w,-1$ then one knows $\mathcal{A}$ will return to read σ_{-k-1} in s_w without scanning σ_{k+1}; so examine what $\mathcal{A}$ would do in s_w reading σ_{-k-1} and re-apply b).

if $\alpha_{-1,x} = s_w,+1$ then one knows $\mathcal{A}$ will step off t_k on the right to read σ_{k+1} in state s_w, examine what $\mathcal{A}$ would do and re-apply b) or apply c) getting $[t_x] \xrightarrow{(\sigma_{-k-1},\sigma_{k+1})} [t_{k+1}]$ (σ_{k+1} is used in place of ω since if σ_{k+1} is scanned by $\mathcal{A}$ then $[t_{k+1}]$ is not independent of σ_{k+1}).

c) if in applying b) one discovers $\mathcal{A}$ would move -1 to scan σ_{-k-2} or move +1 to scan σ_{k+2} then

$[t_k] \xrightarrow{(\sigma_{k-1},\sigma_{k+1})}$



$= [t_{k+1}]$

where $\rho = -1$ if $\mathcal{A}$ scans σ_{-k-2} or +1 if $\mathcal{A}$ scans σ_{k+2}, s_j being the state $\mathcal{A}$ is in when moving to scan σ_{-k-2} or σ_{k+2}; β_{xy} is also determined from $\mathcal{A}$ and $[t_k]$ by using a) b) c) but by starting $\mathcal{A}$ in state s_y on σ_{-k-1} if $x = -1$ and on σ_{k+1} if $x = +1$.

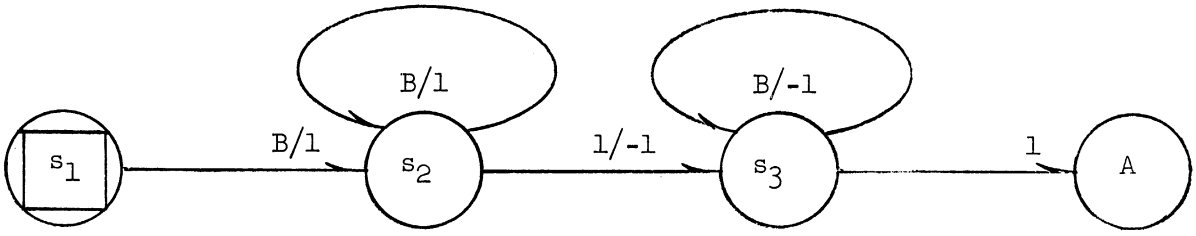
An efficient way of constructing $\mathcal{J}(\mathcal{A})$ is to begin with an initial state, I, and let $\mathcal{J}(\mathcal{A})$ start with both heads on the initial cell of t. Thus the only transitions from I that can occur are transitions on inputs of the form (σ,σ) since h_1 and h_2 read the same symbol when in I. By applying a) b) c) to I of $\mathcal{J}(\mathcal{A})$ one finds all the states of $\mathcal{J}(\mathcal{A})$ immediately accessible from I. To these states one applies all inputs from $\Sigma \times \Sigma$ (all inputs are possible since $\mathcal{J}(\mathcal{A})$ is 1-way) and finds the second rank of accessible states of

$\mathcal{J}(\mathcal{M})$; one continues in this manner until a closed machine $\mathcal{J}(\mathcal{M})$ is formed.

The manner of constructing $\mathcal{J}(\mathcal{M})$ assures one that $G(\mathcal{M}) = G(\mathcal{J}(\mathcal{M}))$. Furthermore, $\mathcal{J}(\mathcal{M})$ may strongly reject some tapes only weakly rejected by $\mathcal{M}$; if one desires $\mathcal{J}(\mathcal{M})$ to also reject (weakly reject) a tape if and only if $\mathcal{M}$ does it then this can be accomplished by adding a weak reject state, WR to $\mathcal{J}(\mathcal{M})$ (WR=a state that loops on itself for all inputs) and when in constructing $\mathcal{J}(\mathcal{M})$ a weak reject by $\mathcal{M}$ is uncovered do not send $\mathcal{J}(\mathcal{M})$ to R but rather to WR.

QED

Example 5.1 Let $\mathcal{M}_{5.1}$ be the 2-way 1-dim 1-head machine shown in Figure 99 that reads tapes written over $\Sigma = \{B,1\}$ and accepts input tape t if and only if t has a blank initial cell and a 1 to the right and left of the initial cell. Find $\mathcal{J}(\mathcal{M}_{5.1})$.



Machine $\mathcal{M}_{5.1}$

Figure 99

Let I be the initial state of $\mathfrak{J}(\mathfrak{A}_{5.1})$. When $\mathfrak{J}(\mathfrak{A}_{5.1})$ is in I the only possible inputs to $\mathfrak{J}(\mathfrak{A}_{5.1})$ are (B,B) and (1,1). So considering $t_k =$ [B] and $t_k =$ [1] one finds that

I —(B,B)→ s_2, 1

s_2,1	s_2,1
s_2,1	s_2,1
s_3,-1	s_3,-1

I —(1,1)→ R

[to be read as:"state I on input (B,B) goes to state s_2, 1

s_2,1	s_2,1
s_2,	s_2,1
s_3,-1	s_3,-1

, etc..."]

Continuing one gets

s_2, 1 —(B,B)→ s_2,1

s_2,1	s_2,1
s_2,1	s_2,1
s_3,-1	s_3,-1

s_2,1 —(B,B)→ s_2,1

s_2,1	s_2,1
s_2,1	s_2,1
s_3,-1	s_3,-1

s_2,1 —(B,1)→ s_3,-1

s_3,-1	R
s_3,-1	s_3,-1
s_3,-1	A

s_2,1 —(1,B)→ s_2,1

R	s_2,1
s_3,-1	s_2,1
A	A

s_2,1 —(1,1)→ A

s_3,-1 —(B,ω)→ s_3,-1

s_3,-1	R
s_3,-1	s_3,-1
s_3,-1	A

s_3,-1 —(B,ω)→ s_3,-1

s_3,-1	
s_3,-1	
s_3,-1	

s_3,-1 —(1,ω)→ A .

s_2,1 —(ω,B)→ s_2,1

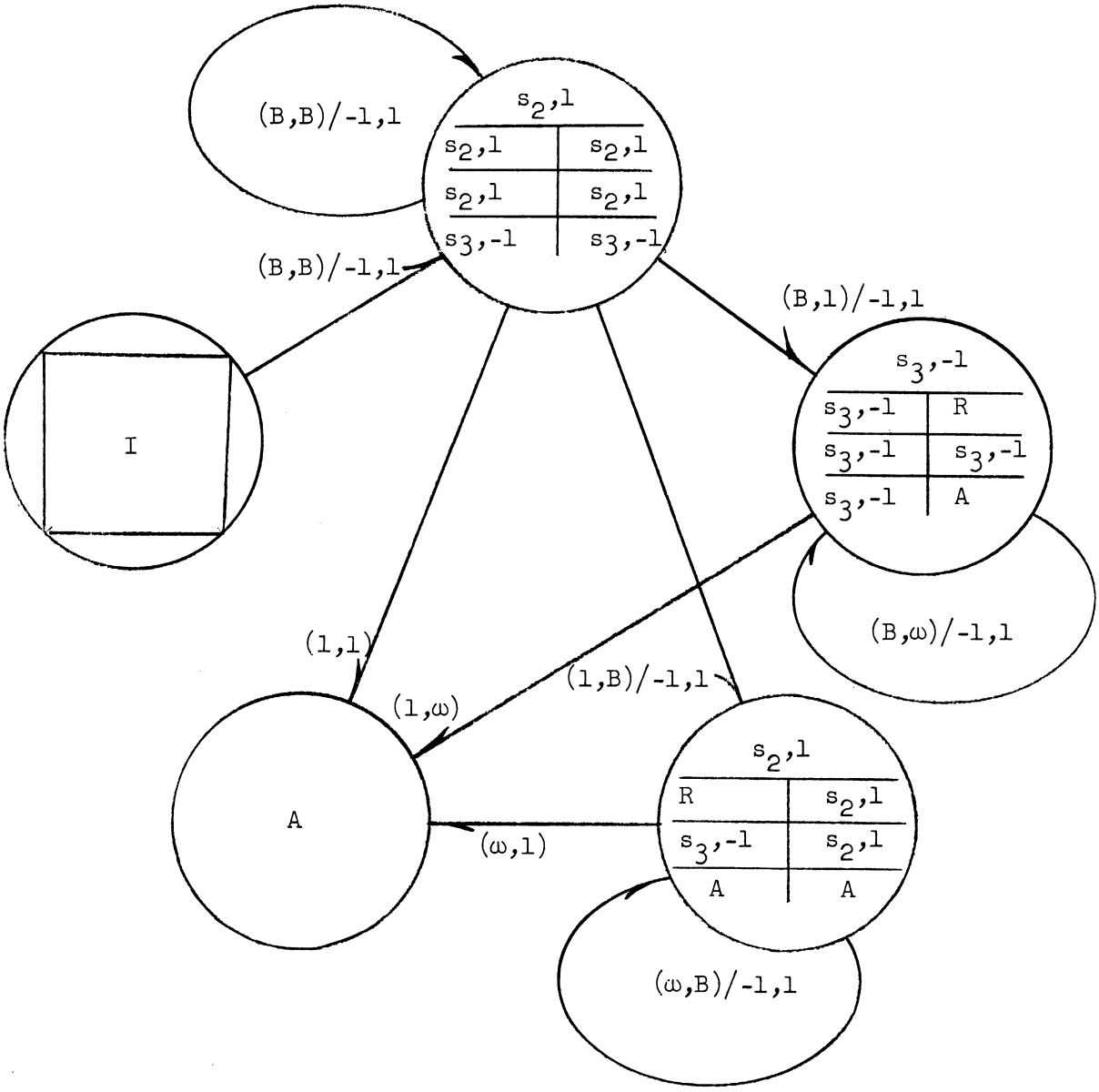
R	s_2,1
s_3,-1	s_2,1
A	A

s_2,1 —(ω,B)→ s_2,1

	s_2,1
	s_2,1
	A

s_2,1 —(ω,1)→ A

Thus a suitable state graph for $\mathfrak{J}(\mathfrak{A}_{5.1})$ is given in Figure 100.



Machine $\mathfrak{J}(\mathfrak{a}_{5.1})$

Figure 100

6.5.4 The "Particular Input" Decision Problem

Def. 5.1 Let $\mathcal{A}$ be any n-head machine and t any input to $\mathcal{A}$ (t is in general an m-tuple of tapes) then $\tau_{\mathcal{A}}(t)$ is defined if and only if $\mathcal{A}$ accepts or strongly rejects t and in that event $\tau_{\mathcal{A}}(t)$ equals the number of machine cycles it takes for $\mathcal{A}$ to accept or reject t.

Theorem 5.2 If $\mathcal{A} = < C,S,s^I,M >$ is any 1-head machine working on D-dim tapes and if t is a D-dim tape for which the initial cell and all non-blank cells can be enclosed in a D-dim rectangular parallelepiped of dimensions $\ell_1 \times \ell_2 \times \ldots \times \ell_D$ and if $\tau_{\mathcal{A}}(t)$ is defined (if $\mathcal{A}$ accepts or strongly rejects t) then

$$\tau_{\mathcal{A}}(t) < \overline{\overline{S}} \prod_{i=1}^{D} (\ell_i + 2\overline{\overline{S}})$$

Proof: Let P_1 be the rectangular parallelepiped of dimensions $\ell_1 \times \ell_2 \times \ldots \times \ell_D$ that encloses the initial cell and the non-blank cells of t. Enclose P_1 with a larger rectangular parallelepiped P_2 such that the corresponding sides of P_1 and P_2 are $\overline{\overline{S}}$ cells apart. P_2 therefore has dimensions $(\ell_1 + 2\overline{\overline{S}}) \times (\ell_2 + 2\overline{\overline{S}}) \times \ldots \times (\ell_D + 2\overline{\overline{S}})$. Let $\mathcal{A}$ work on t, its head starting on the initial cell inside P_1. After $\cdot \overline{\overline{S}} \prod_{i=1}^{D} (\ell + 2\overline{\overline{S}}) = \tau$ machine cycles one of three possibilities must have occurred:

Possibility 1) $\mathcal{A}$ accepts or strongly rejects t; in which event the theorem holds.

Possibility 2) $\mathcal{A}$ neither accepts or strongly rejects t and the head of $\mathcal{A}$ never left P_2. But τ equals the total possible combinations of head position in P_2 and state of $\mathcal{A}$; if after τ machine cycles $\mathcal{A}$ never left P_2 nor accepted or rejected t than $\mathcal{A}$ must be in a loop and therefore never will accept or reject t. Thus the theorem holds.

Possibility 3) $\mathcal{A}$ neither accepted nor rejected t and the head of $\mathcal{A}$ left P_2. Let h (the head of $\mathcal{A}$) have left P_2 for the first time during the i-th machine cycle. By the construction of P_2 and P_1 one knows that h has read B for the last $\overline{\overline{S}}$ machine cycles preceeding the i-th. Since in reading these $\overline{\overline{S}}$ B's $\mathcal{A}$ neither accepted nor strongly rejected t but instead moved away from P_1 we are assured that $\mathcal{A}$ will continue to read blanks and move further away from P_1, never accepting or strongly rejecting t. Thus the theorem holds.

QED

Note 5.3 In the trivial case of $\mathcal{A}$ being a 0-head machine the acceptance or rejectance of all tapes is a function only of S and M of $\mathcal{A}$. If $\tau_{\mathcal{A}}(t)$ is defined in this case then for all t, $\tau_{\mathcal{A}}(t) \leq \overline{\overline{S}}$.

Note 5.4 Minsky[45] has shown no procedure exists for determining if a general 2-head 2-tape machine accepts or strongly rejects a particular input t. His results in no way require the heads of the machine to work on separate tapes and so one can conclude that: if $n \geq 2$ no procedure exists to determine if a general n-head machine accepts or strongly rejects a particular input t.

In contrast with theorem 5.2 it is a direct consequence of Minsky's result that there is no function $f(\mathcal{A},t)$ of $\mathcal{A}$ and t such that if $\mathcal{A}$ is a general n-head machine and t an input, $\tau_{\mathcal{A}}(t) \leq f(\mathcal{A},t)$ if $\tau_{\mathcal{A}}(t)$ exists. If such a function existed then there would indeed be a procedure to decide if any general n-head machine accepted a particular input t.

6.5.5 The Emptiness Decision Problem

Of the several decision problems one can propose dealing with n-head machines there are three which can be shown to be equivalent. These decision problems are:

1) The emptiness decision problem: given any n-head machine $\mathcal{M}$ does $\mathcal{M}$ accept any input whatsoever? (i.e, does $T(\mathcal{M}) = \phi$).
2) The state accessibility problem: given any n-head machine $\mathcal{M}$ and any internal state s of $\mathcal{M}$ is s accessible?
3) The transition accessibility problem: given any n-head machine $\mathcal{M}$ and any transition τ of $\mathcal{M}$ is τ accessible?

Theorem 5.3 The emptiness decision problem (1), the state accessibility problem(2), and the transition accessibility problem (3) are equivalent in the sense that one can devise a general procedure to answer one of the problems for all n-head machines if and only if one can devise a general procedure to answer all of the problems for all n-head machines.

Proof: One can present the proof by showing that a general procedure to solve (3) $\Longrightarrow$ a general procedure to solve (2) $\Longrightarrow$ a general procedure to solve (1) $\Longrightarrow$ a general procedure to solve (3) [or in short notation $gp(3) \Longrightarrow gp(2) \Longrightarrow gp(1) \Longrightarrow gp(3)$].

a) $gp(3) \Longrightarrow gp(2)$: let $\mathcal{M}$ be any n-head machine and s any state of $\mathcal{M}$. gp(3) assures us we can determine if any transition of $\mathcal{M}$ is accessible. Consider each of the transitions entering s and determine if each is accessible. s is accessible if and only if one or more of the transitions entering s is accessible. Thus $gp(3) \Longrightarrow gp(2)$.

b) gp(2) $\Rightarrow$ gp(1): let $\mathcal{A}$ be any n-head machine with ACCEPT state A. $T(\mathcal{A}) \neq \emptyset$ if and only if A is an accessible state of $\mathcal{A}$. But gp(2) assures us we can determine if A is accessible. Thus gp(2) $\Rightarrow$ gp(1).

c) gp(1) $\Rightarrow$ gp(3): let $\mathcal{A}$ be any n-head machine and τ any transition of $\mathcal{A}$. Alter $\mathcal{A}$ by letting all inputs to A go to R and by changing the destination of τ to A (if τ goes to A originally then leave it). Call this new machine $\mathcal{A}'$. $T(\mathcal{A}') \neq \emptyset \Rightarrow \tau$ accessible in $\mathcal{A}$ and gp(1) assures us we can determine if $T(\mathcal{A}') = \emptyset$. Thus gp(1) $\Rightarrow$ gp(3).

QED

Theorem 5.4 Given any 1-dim 1-head machine $\mathcal{A}$ there is a general procedure for determining if $T(\mathcal{A}) = \emptyset$.

Proof: If $\mathcal{A}$ is 1-way then one can apply the result of Theorem 7 of Rabin and Scott[6] to $\mathcal{A}$ and thereby decide if $T(\mathcal{A}) = \emptyset$. If $\mathcal{A}$ is 2-way then Theorem 7 of Rabin and Scott can be applied to $\mathfrak{J}(\mathcal{A})$, the 2-head 1-way equivalent of $\mathcal{A}$; $T(\mathcal{A}) = \emptyset$ if and only if $T(\mathfrak{J}(\mathcal{A})) = \emptyset$.

QED

Note 5.5 If $\mathcal{A}$ is a 1-dim 1-way machine then Theorem 9 of Rabin and Scott can be applied to $\mathcal{A}$ to determine if $G(\mathcal{A})$ is infinite. If $\mathcal{A}$ is 1-dim 2-way then Theorem 9 of Rabin and Scott can be applied to $\mathfrak{J}(\mathcal{A})$ to determine if $G(\mathcal{A})$ is infinite.

Note 5.6 Since every 1-way n-head machine reading over $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ is isomorphic to a 1-way 1-head machine reading over $\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \vdots \\ \Sigma_n \end{bmatrix}$ it is evident via

Theorem 5.4 that a general procedure exists to determine if $T(\mathcal{A}) = \phi$ if $\mathcal{A}$ is a 1-way n-head machine.

Theorem 5.5 There is no effective procedure for deciding if $T(\mathcal{A}) = \phi$ for any general n-head machine $\mathcal{A}$, if $n \geq 2$.

Proof: This result is proved by Rabin and Scott in their Theorem 19.

QED

Theorem 5.6 There is no effective procedure for deciding if $T(\mathcal{A}) = \phi$ for any general 1-head machine $\mathcal{A}$ if $\mathcal{A}$ works on tapes of dimension $D \geq 2$.

Proof: Consider the set $\mathfrak{B}$ of all 2-way 1-dim 2-tape 2-head machines such that the state set S of each machine in $\mathfrak{B}$ is partitioned into two sub-sets S_1 and S_2 and such that on all transitions from states in S_1 only head h_1 will move and on all transitions from states in S_2 only head h_2 will move. The set $\mathfrak{B}$ is precisely the set of "two-way two-tape automata" described by Rabin and Scott.

The input to any machine $\mathfrak{B}$ in $\mathfrak{B}$ will be restricted to pairs of 1-dim partial tapes of the form (ht_1h, ht_2h) where the initial cell of each tape corresponds to the first cell in t_1 and t_2 respectively and where Σ, the alphabet of t_1 and t_2 does not contain h. h is an endmark and in operation $\mathfrak{B}$ confines its head movements strictly to the cells filled by ht_1h and ht_2h.

Rabin and Scott have shown in their Theorem 19 that in general no effective procedure exists to determine if $T(\mathfrak{B}) = \phi$.

One can show that for any $\mathfrak{B}$ in $\mathfrak{B}$ there is a 1-head machine $\mathcal{A}$ working on 2-dim tapes such that $T(\mathfrak{B}) = \phi$ if and only if $T(\mathcal{A}) = \phi$

and therefore no effective method exists to determine if $T(\mathcal{A}) = \emptyset$ since if a method did exist we could determine (contra Rabin and Scott) if $T(\mathcal{B}) = \emptyset$ for all $\mathcal{B}$ in $\mathfrak{B}$.

Let $\mathcal{B} \in \mathfrak{B}$. Let t_1 and t_2 be any 1-dim partial tapes over Σ, the alphabet of $\mathcal{B}$. One defines $(ht_1h) \times (ht_2h)$ as follows:

> $(ht_1h) \times (ht_2h)$ will be a 2-dim partial tape written over $(\Sigma \cup \{h\})^2$ such that cell (0,0) will be the initial cell of $(ht_1h) \times (ht_2h)$ and such that if σ_i is in the i-th cell of ht_1h and τ_j is in the j-th cell of ht_2h then cell (i,j) of $(ht_1h) \times (ht_2h)$ will contain (σ_i, τ_j). If $\ell g(t_x)$ is the number of filled cells in t_x then the contents of cell (i,j) in $(ht_1h) \times (ht_2h)$ is defined only for
>
> $$-1 \leq i \leq \ell g(t_1)$$
>
> and
>
> $$-1 \leq j \leq \ell g(t_2)$$

From $\mathcal{B}$ one constructs $\mathcal{A}_2$ such that $\mathcal{A}_2$ has the same transition structure as $\mathcal{B}$; however $\mathcal{A}_2$ is 1-head and reads over inputs in $(\Sigma \cup \{h\})^2$ whereas $\mathcal{B}$ is 2-head and each head reads over $\Sigma \cup \{h\}$. Thus the <u>input</u> labels to transitions in $\mathcal{A}_2$ and $\mathcal{B}$ will identical. As for the head movements of $\mathcal{A}_2$, if a particular transition of $\mathcal{B}$ had head movement

a) (1,0) then $\mathcal{A}_2$ moves its head + 1,

b) (-1,0) then $\mathcal{A}_2$ moves its head - 1,

c) (0,1) then $\mathcal{A}_2$ moves its head + 2,

d) (0,-1) then $\mathcal{A}_2$ moves its head - 2.

By the construction of $\mathcal{B}$ all head movements of $\mathcal{B}$ must be one of the four listed above; thus $\mathcal{A}_2$ is well defined for each $\mathcal{B}$.

It follows directly from the manner in which $\mathcal{A}_2$ was constructed that

$$(ht_1h,\ ht_2h) \in T(\mathcal{B}) \iff (ht_1h) \times (ht_2h) \in T(\mathcal{A}_2).$$

One can construct a 1-head machine $\mathcal{A}_1$ that accepts any 2-dim tape t if and only if t has a subtape of the form $(ht_1h) \times (ht_2h)$. Furthermore $\mathcal{A}_1$ can be built such that it will halt on the initial cell of t if t is accepted.

If one merges and identifies the A state of $\mathcal{A}_1$ with the initial state of $\mathcal{A}_2$ one obtains a composite machine $\mathcal{A}$ such that

$$\begin{aligned}
& T(\mathcal{A}) \stackrel{?}{=} \emptyset \\
\iff\ & T(\mathcal{A}_1) \cap T(\mathcal{A}_2) \stackrel{?}{=} \emptyset \\
\iff\ & \text{does there exist a } t_1 \text{ and } t_2 \text{ such that} \\
& (ht_1h) \times (ht_2h) \in T(\mathcal{A}_2) \\
\iff\ & T(\mathcal{B}) \stackrel{?}{=} \emptyset .
\end{aligned}$$

Since $T(\mathcal{B}) \stackrel{?}{=} \emptyset$ is not effectively decidable one concludes that $T(\mathcal{A}) \stackrel{?}{=} \emptyset$ is not effectively decidable.

QED

Note 5.7 In a manner similar to the proof of Theorem 5.6 one can show that no effective procedure exists to decide if any general 2-dim 1-head machine strongly rejects any tape.

6.5.6 Boolean Properties of n-Head Machines

Theorem 5.7 If $\mathcal{A}_1$ and $\mathcal{A}_2$ are n_1-head and n_2-head machines respectively, then there exist machines $\mathcal{B}_1$ and $\mathcal{B}_2$, each with at most n_1+n_2 heads such that

a) $T(\mathfrak{B}_1) = T(\mathfrak{A}_1) \cap T(\mathfrak{A}_2)$

and

b) $T(\mathfrak{B}_2) = T(\mathfrak{A}_1) \cup T(\mathfrak{A}_2)$.

Proof: a) Let $\mathfrak{B}_1$ have n_1+n_2 heads with the first n_1 heads placed on tapes in the manner of $\mathfrak{A}_1$ and the second n_2 heads placed on tapes in the manner of $\mathfrak{A}_2$. Let the states of $\mathfrak{B}_1$ be doubletons of the form (s_{i_1}, s_{i_2}) where $s_{i_1} \in S_{\mathfrak{A}_1} \cup \{A,R\}$ and $s_{i_2} \in S_{\mathfrak{A}_2} \cup \{A,R\}$. Let $(s^I_{\mathfrak{A}_1}, s^I_{\mathfrak{A}_2})$ be the initial state of $\mathfrak{A}_1$.

If $\mathfrak{B}_1$ is in state (s_{i_1}, s_{i_2}) and reads input $(\sigma_1, \sigma_2, \ldots, \sigma_{n_1+n_2})$ then $\mathfrak{B}_1$ goes to state (s_{j_1}, s_{j_2}) with head movements $(d_1, d_2, \ldots, d_{n_1+n_2})$ where s_{j_1}, s_{j_2} and $d_1, d_2, \ldots, d_{n_1+n_2}$ are determined from the transition tables of $\mathfrak{A}_1$ and $\mathfrak{A}_2$ as follows:

$M_{\mathfrak{A}_1}$: $(s_{i_1}, \sigma_1, \sigma_2, \ldots, \sigma_{n_1}) \longrightarrow (s_{j_1}, d_1, d_2, \ldots, d_{n_1})$

$M_{\mathfrak{A}_2}$: $(s_{i_2}, \sigma_{n_1+1}, \ldots, \sigma_{n_1+n_2}) \longrightarrow (s_{j_2}, d_{n_1+1}, \ldots d_{n_1+n_2})$.

[it is understood that on all inputs $\mathfrak{A}_1$ and $\mathfrak{A}_2$ go from A to A].

The ACCEPT state in $\mathfrak{B}_1$ is (A, A).

As constructed $\mathfrak{B}_1$ will accept an m-tuple of tapes t if and only if t is accepted by both $\mathfrak{A}_1$ and $\mathfrak{A}_2$; thus $T(\mathfrak{B}_1) = T(\mathfrak{A}_1) \cap T(\mathfrak{A}_2)$.

b) Construct $\mathfrak{B}_2$ exactly as $\mathfrak{B}_1$ above and then merge all states of the form (A, A), (A, s_{i_2}), (s_{i_1}, A) into a single accept state. $\mathfrak{B}_2$ will accept an m-tuple of tapes t if and only if $\mathfrak{A}_1$ or $\mathfrak{A}_2$ or both accept t; thus $T(\mathfrak{B}_2) = T(\mathfrak{A}_1) \cup T(\mathfrak{A}_2)$.

QED

Theorem 5.8 If $\mathcal{A}$ is any n-head machine that strongly represents $T(\mathcal{A})$ (i.e., any input to $\mathcal{A}$ is either accepted or strongly rejected) then there is an n-head machine $\mathcal{B}$ that strongly represents $\sim T(\mathcal{B})$.

Proof: Interchange the labels of the A and R states of $\mathcal{A}$. One obtains an n-head machine $\mathcal{B}$ that strongly represents $\sim T(\mathcal{A})$ since if t takes $\mathcal{A}$ to A then it takes $\mathcal{B}$ to R and if t takes $\mathcal{A}$ to R it takes $\mathcal{B}$ to A.

QED

Note 5.8 One is obliged to restrict the hypothesis of Theorem 5.8 to machines that strongly represent their sets. The reason for this is that there are some sets which can be weakly represented at best and thus the construction of Theorem 5.8 would not be possible. A case in point: let T be the set of all 1-dim tapes over {B, 0} such that at least one cell to the right of the initial cell contains 0. T can be weakly represented by 1-way 1-dim 1-head machine $\mathcal{A}_{5.2}$ shown in Figure 101.

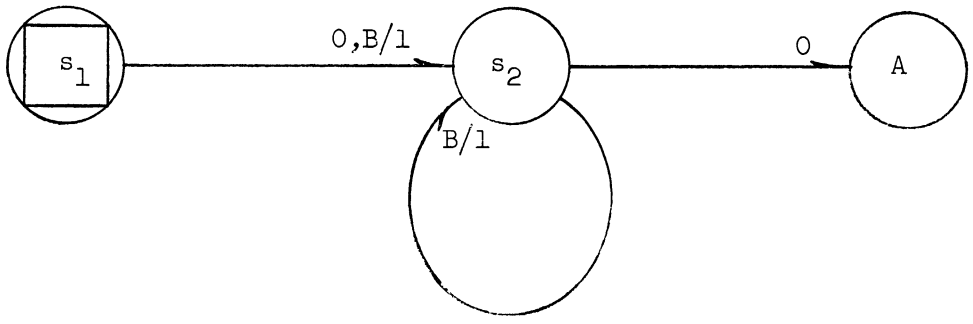


Figure 101. Machine $\mathcal{A}_{5.2}$.

Any tape t' with all blanks to the right of the initial cell is weakly rejected by $\mathcal{A}_{5.2}$ therefore any machine $\mathcal{B}$ that purports to represent $\sim T(\mathcal{A}_{5.2})$ must accept t'. But no such $\mathcal{B}$ can exist for we would have to require that $\mathcal{B}$ check all cells to the right of the initial cell for blanks - thereby implying that $\mathcal{B}$ must go through an infinite number of cycles before accepting t'. But via Theorem 5.2 one deduces that $\mathcal{B}$ must accept t' in a finite number of cycles. Therefore by contradiction $\mathcal{B}$ can not exist.

6.5.7 Speed Theorems

Theorem 5.9 If $\mathcal{A}$ is any 1-dim 1-head machine and t any tape for which $\tau_{\mathcal{A}}(t)$ is defined then

$$\tau_{\mathcal{A}}(t) \geq \tau_{\mathcal{J}(\mathcal{A})}(t).$$

Furthermore if in accepting or strongly rejecting t, $\mathcal{A}$ stands still or reverses direction then

$$\tau_{\mathcal{A}}(t) > \tau_{\mathcal{J}(\mathcal{A})}(t).$$

Proof: For any 1-dim tape t let t_k be the subtape of t consisting of the cells -k, -k+1, ..., -1,0,1, ..., k. If t is accepted or strongly rejected by $\mathcal{A}$ then there exists a smallest k such that $\mathcal{A}$ accepts or strongly rejects t_k and never leaves t_k. Since k is the smallest such number it follows that $\mathcal{A}$ must read cell -k or +k of t. Therefore $\tau_{\mathcal{A}}(t) \geq k$. But by construction $\mathcal{J}(\mathcal{A})$ is 1-way; thus one deduces that $\tau_{\mathcal{J}(\mathcal{A})} = k$. Therefore $\tau_{\mathcal{A}}(t) \geq \tau_{\mathcal{J}(\mathcal{A})}(t)$.

If in addition one knows that $\mathcal{A}$ stands still or reverses direction in accepting or strongly rejecting t then

$$\tau_{\mathcal{A}}(t) > k = \tau_{\mathcal{J}(\mathcal{A})}(t)$$

QED

Theorem 5.10 If $\mathcal{A}$ is any 1-dim 1-head machine and t any tape for which $\tau_{\mathcal{A}}(t)$ is defined then there is no n-head machine $\mathcal{B}$ for any n such that $G(\mathcal{B}) = G(\mathcal{A})$ and such that $\tau_{\mathcal{B}}(t) < \tau_{\mathfrak{J}(\mathcal{A})}(t)$.

Proof: If $\mathcal{B}$ is any such n-head machine then as in the proof of Theorem 5.9, $\mathcal{B}$ must scan cell -k or +k of t_k in order to accept or strongly reject t. Thus $\tau_{\mathcal{B}}(t) \geq k = \tau_{\mathfrak{J}(\mathcal{A})}(t)$. Thus it is not possible that $\tau_{\mathcal{B}}(t) < \tau_{\mathfrak{J}(\mathcal{A})}(t)$.

QED

Theorem 5.11 There exists an infinite collection of sets of 1-dim tapes $\mathcal{A} = \{A_j\}$, each set A_j representable by 1-head machines such that if $\mathcal{A}_j$ is any 1-head machine representing A_j (i.e., $T(\mathcal{A}_j) = A_j$) then for any tape t for which $\tau_{\mathcal{A}_j}(t)$ is defined

$$\tau_{\mathcal{A}_j}(t) > \tau_{\mathfrak{J}(\mathcal{A}_j)}(t).$$

Proof: Let A_j be the set of 1-dim tapes written over $\Sigma = \{B,a\}$ such that $A_j = \{t|$ there are at least j a's to the right and left of the initial cell$\}$. A_j can be represented by a 1-head machine $\mathcal{A}'_j$ which operates as follows:

1) $\mathcal{A}'_j$ reads the initial cell; if B go to 2), if a go to 3)
2) move right counting the a's but not the B's; after j a's reverse and count left for 2 j a's. On the 2j-th a moving left accept t.
3) move right counting the a's but not the B's; after j a's reverse and count left for (2j + 1) a's. On the (2j+1)-th a moving left accept t.

Thus there is at least one 1-head machine that represents A_j.

Let $\mathcal{A}_j$ be <u>any</u> 1-head machine that represents A_j. $\mathcal{A}_j$ cannot strongly reject any tape since if t' is strongly rejected by $\mathcal{A}_j$ then t' must have less than j a's either to the left or right of the initial cell and no $\mathcal{A}_j$ could check this in a finite number of cycles. Thus for any $\mathcal{A}_j$ $\tau_{\mathcal{A}_j}(t)$ is defined if and only if $t \in A_j$. But if $t \in A_j$ then $\mathcal{A}_j$ must reverse direction in accepting t since the definition of A_j requires that $\mathcal{A}_j$ check both to the left and the right of the initial cell. Thus via Theorem 5.9 $\tau_{\mathcal{A}_j}(t) > \tau_{\mathfrak{J}(\mathcal{A}_j)}(t)$ for all t such that $\tau_{\mathcal{A}_j}(t)$ is defined.

<u>QED</u>

<u>Note 5.9</u> The final paragraph of the proof of Theorem 5.1 assures one that $\tau_{\mathcal{A}}(t)$ defined $\Rightarrow$ $\tau_{\mathfrak{J}(\mathcal{A})}(t)$ defined for any 1-dim 1-head machine $\mathcal{A}$ and and any tape t. Furthermore if $\mathfrak{J}(\mathcal{A})$ is constructed such that $\mathfrak{J}(\mathcal{A})$ weakly rejects t if and only if $\mathcal{A}$ weakly rejects t then $\tau_{\mathcal{A}}(t)$ defined $\Leftrightarrow$ $\tau_{\mathfrak{J}(\mathcal{A})}(t)$ defined.

<u>Theorem 5.12</u> For any integer $k>0$ there exists an infinite number of sets of 1-dim tapes all representable by 1-dim 1-head machines and such that if A is any such set and $\mathcal{A}$ any 1-head machine representing A then

a) for all t in A $\quad \tau_{\mathcal{A}}(t) \geq \tau_{\mathfrak{J}(\mathcal{A})}(t) + 2k$

b) for all t in A', A' being an infinite subset of A,

$$\tau_{\mathcal{A}}(t) > k\, \tau_{\mathfrak{J}(\mathcal{A})}(t) .$$

<u>Proof</u>: Part b) of the theorem will be proved first. For convenience and without loss of generality one can limit k to the even integers.

Let

$$\Sigma = \{B, \sigma_1, \sigma_2, \ldots, \sigma_{k+3}\} .$$

Let the set A_k be defined as all 1-dim tapes over Σ of the form shown in Figure 102 where $\sigma_{\alpha_i} \neq \sigma_{\alpha_j}$ and $\sigma_{\alpha_i} \neq B$ for all $i, j, i \neq j$.

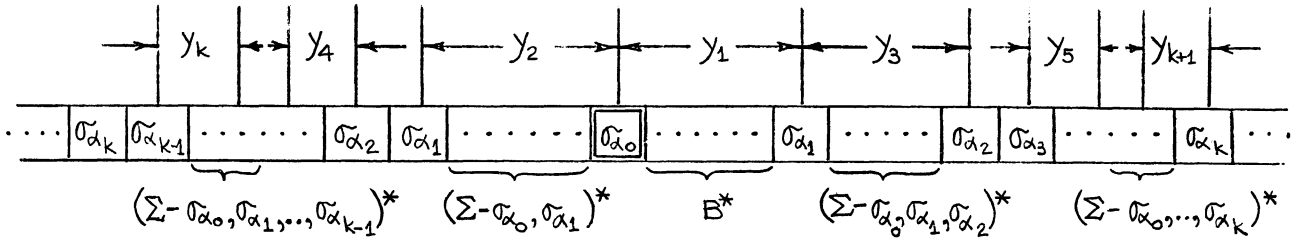


Figure 102. Form of Tapes in A_k.

There exists at least one 1-head machine $\mathcal{A}$ that represents A_k. $\mathcal{A}$ works as follows:

1) Read initial cell and remember σ_{α_0}; if $\sigma_{\alpha_0} = B$ reject t.
2) Move right to first non-blank cell. This contains σ_{α_1}. Check $\sigma_{\alpha_1} \neq \sigma_{\alpha_0}$ and $\sigma_{\alpha_1} \neq B$ and remember σ_{α_1}.
3) Move left past σ_{α_0} to first occurrence of σ_{α_1}. Check that σ_{α_0} has not occurred more than once. Move left to read σ_{α_2}. Check σ_{α_2}. Check $\sigma_{\alpha_2} \neq B$ or $\sigma_{\alpha_2} \neq \sigma_{\alpha_1}$ or σ_{α_0}.
4) Move right past σ_{α_1}, ... σ_{α_0}, ...σ_{α_1}, ... to σ_{α_2},.... etc..

.
.
.

$\mathcal{A}$ will finally move left to read $\sigma_{\alpha_{k-1}}$, will check for proper occurrences of σ_{α_0}, σ_{α_1}, ... and will move left to read σ_{α_k}. Check

that $\sigma_{\alpha_k} \neq B$, σ_{α_o}, σ_{α_1}, ..., $\sigma_{\alpha_{k-1}}$. Then move right passing $\sigma_{\alpha_{k-1}}$, $\sigma_{\alpha_{k-2}}$, ..., σ_{α_o}, σ_{α_1}, ..., $\sigma_{\alpha_{k-1}}$, ... σ_{α_k} and stop. Accept t.

The complete process described above requires only a finite memory and therefore can be done by a finite state machine.

Referring to the tape form in Figure 102 let the distance from the initial cell to σ_{α_i} on the left and on the right be x_{iL} and x_{iR} respectively. Any tape in A_k is governed by the relations

$$
\begin{aligned}
y_i &\geq 1 \quad \text{for all } i \\
x_{1R} &= y_1 \\
x_{2R} &= x_{1R} + y_3 \\
x_{3R} &= x_{2R} + 1 \\
x_{4R} &= x_{3R} + y_5 \\
&\vdots \\
x_{(k-1)R} &= x_{(k-2)R} + 1 \\
x_{kR} &= x_{(k-1)R} + y_{k+1} \\
\\
x_{1L} &= y_2 \\
x_{2L} &= x_{1L} + 1 \\
x_{3L} &= x_{2L} + y_4 \\
&\vdots \\
x_{kL} &= x_{(k-1)L} + 1
\end{aligned}
$$

Consider <u>any</u> 1-head machine $\mathcal{M}'$ that represents A_k. Consider also the set of tapes in A_k such that $y_i > \overline{\overline{S}}_{\mathcal{M}'}$ for all i. Call this

subset of A_k by the name A_k''.

$$(\forall t)_{t\epsilon A_k''} \quad \tau_{\mathcal{A}'}(t) \geq \tau_{\mathcal{A}}(t)$$

since $\mathcal{A}'$ if it represents A_k can go at most a distance $\leq \overline{\overline{S}}_{\mathcal{A}}$, past each σ_{α_i} before reversing direction and discovering the value of $\sigma_{\alpha_{i+1}}$.

Consider A_k' the subset of A_k'' that contains all tapes of A_k'' such that $y_1 > \frac{rk^2 + k(k-2)}{2}$

and

$$y_2, y_3, \ldots, y_{k+1} = r$$

where

$$r = \overline{\overline{S}}_{\mathcal{A}'} + 1.$$

A_k' is an infinite subset of A_k and for any $t\epsilon A_k'$, $\tau_{\mathcal{A}'}(t) \geq \tau_{\mathcal{A}}(t)$ since $A_k' \subseteq A_k''$.

But
$$\begin{aligned}\tau_{\mathcal{A}}(t) = {} & 2y_1 + 2(y_2+1) + 2(y_1+y_3+1) \\ & + 2(y_2+1+y_4+1) + \ldots\ldots \\ & \ldots\ldots + (y_1+y_3+1+y_5+1+ \ldots\ldots +1+y_{k+1}) \\ & > ky_1 + y_1.\end{aligned}$$

Thus

$$\tau_{\mathcal{A}'}(t) > ky_1 + y_1 \qquad \text{for all } t\epsilon A_k'.$$

Now

$$\tau_{\mathcal{J}(\mathcal{A})}(t) = \max [x_{kR}, x_{kL}] .$$

But for all $t\epsilon A_k'$ $\quad x_{kR} = x_{kL} + y_1 > x_{kL}$.

Thus

$$\begin{aligned}\tau_{\mathcal{J}(\mathcal{A})}(t) &= y_1 + y_3 + 1 + \ldots\ldots + 1 + y_{k+1} \\ &= y_1 + \frac{rk+k-2}{2}\end{aligned}$$

for all $t\epsilon A_k'$.

So for all $t \epsilon A_k'$

$$\tau_{\mathcal{A}'}(t) - k\,\tau_{\mathcal{J}(\mathcal{A})} > ky_1 + y_1 - ky_1 - \frac{rk^2 + k(k-2)}{2}$$

$$> y_1 - \frac{rk^2 + k(k-2)}{2}$$

But if $t \epsilon A_k'$ then

$$y_1 > \frac{rk^2 + k(k-2)}{2}$$

and so

$$\tau_{\mathcal{A}'}(t) - k\tau_{\mathcal{J}(\mathcal{A})} > 0$$

or

$$\tau_{\mathcal{A}'}(t) > k\tau_{\mathcal{J}(\mathcal{A})}$$

which proves the first part of the theorem. If A_k satisfies the theorem then $A_{k+2\ell}$ for all $\ell \geq 0$ also satisfies the theorem. Therefore the number of sets of tapes satisfying the theorem for any particular k is infinite.

To deduce part a) of the theorem one can argue that if $\mathcal{A}'$ is any 1-head machine that represents A_k then $\mathcal{A}'$ must at least go out to read σ_k on one end and then reverse and read out to σ_k on the other end. Thus for any $t \epsilon A_k$

$$\tau_{\mathcal{A}'}(t) \geq \min [2x_{kL} + x_{kR},\ 2x_{kR} + x_{kL}].$$

But for any $t \epsilon A_k$

$$\tau_{\mathcal{J}(\mathcal{A})}(t) = \max [x_{kL}, x_{kR}].$$

Thus for all $t \epsilon A_k$

$$\tau_{\mathcal{A}'}(t) - \tau_{\mathcal{J}(\mathcal{A})}(t) = \min [x_{kL} + x_{kR},\ 2x_{kL},\ 2x_{kR}]$$

$$\geq 2k$$

or

$$\tau_{\mathcal{A}'}(t) \geq \tau_{\mathcal{J}(\mathcal{A})}(t) + 2k.$$

QED

Theorem 5.13 Let $\mathcal{G}_\mathcal{L} = \{\mathcal{A}_i\}$ be the set of all 1-head machines recognizing some set of 1-dim tapes A. Let $\mathcal{J}(\mathcal{G}_\mathcal{L})$ be the 1-way 2-head equivalent of any machine in $\mathcal{G}_\mathcal{L}$. Then for any particular tape t_o in A there is a machine $\mathcal{A}$ in $\mathcal{G}_\mathcal{L}$ such that

$$\tau_{\mathcal{A}}(t_o) \leq 3\tau_{\mathcal{J}(\mathcal{G}_\mathcal{L})}(t_o).$$

Proof: $\mathcal{J}(\mathcal{G}_\mathcal{L})$ is independent of which machine in $\mathcal{G}_\mathcal{L}$ was used as its basis since all machines in $\mathcal{G}_\mathcal{L}$ have the same set of generators.

Let $t_o \in A$ and let $\tau_{\mathcal{J}(\mathcal{G}_\mathcal{L})}(t_o) = x$. $\mathcal{A}$ can be constructed to first check any tape t by reading left x cells and then right 2x cells - this gives $\mathcal{A}$ enough information to decide if t and t_o have the same generator. If t has the same generator as t_o then $\mathcal{A}$ accepts t; if not $\mathcal{A}$ moves x cells left (which returns its head to the initial cell) and then proceeds to examine t according to the procedure of any machine $\mathcal{A}_i$ in $\mathcal{G}_\mathcal{L}$.

By the construction of $\mathcal{A}$ it is necessary that $\mathcal{A} \in \mathcal{G}_\mathcal{L}$ and that

$$\tau_{\mathcal{A}}(t_o) \leq 3x = 3\tau_{\mathcal{J}(\mathcal{G}_\mathcal{L})}(t_o).$$

QED

6.6 TOPICS FOR FURTHER STUDY

6.6.1 Reduction Problems

Among the possible criteria one can use as a measure of the complexity of n-head machines are three that arise naturally from the structure of n-head machines; namely, the number of heads, the number of states, and the speed in accepting or strongly rejecting inputs. Relative to these criteria three problems can be formulated:

1) Head Reduction Problem: given a set of tapes T produce a machine with as few heads as possible that represents T.
2) State Reduction Problem: given a set of tapes T produce a machine with as few states as possible that represents T.
3) Speed Reduction Problem: given a set of tapes T produce a machine that represents T and that accepts or rejects inputs as quickly as possible.

The above three problems, both in their most general form and in many special forms, constitute an area of almost totally unexplored questions. A collection of remarks and observations on these reduction problems follows below.

6.6.1.1 Head Reduction

Two heads, h_i and h_j, of any machine $\mathcal{M}$ will be said to be bound if and only if h_i and h_j are on the same tape and if for all inputs to $\mathcal{M}$ there is a finite upper bound on the distance that ever exists between h_i and h_j. The bound property determines an equivalence relation on the set

of heads H of $\mathcal{A}$ in that heads are in the same equivalence class if and only if they are bound to each other. It is a consequence of the bound property that if H is divided into p such equivalence classes then $\mathcal{A}$ can be shown to be computationally equivalent to a machine with p heads (one head per equivalence class of H). However, no general method is known to determine if two heads are bound and further there is no guarantee that the p-head machine is indeed the minimum head machine equivalent to $\mathcal{A}$.

One might try to show that for each i = 1,2,... there is a set of inputs C_i such that C_i can be represented by a machine with i-heads but no fewer. This is indeed the case if C_i equals some non-trivial set of i-tuples; thus to represent C_i any machine must have at least one head per tape or at least i-heads. In order to render the question more significant one might re-ask the question but restrict C_i to be a set of 1-dim tapes. It is the author's conjecture that the set C_i defined as the set of 1-dim tapes written over $\Sigma = \{B,0,1\}$ and having generators of the form $0^{x_1}1\,0^{x_2}1\,0^{x_3}\ldots\ldots1\,0^{x_{i-1}}1\,0^{x_1x_2\cdots x_{i-1}}$ can be represented with no machine having fewer than i-heads. Certainly C_i can be represented by an i-head machine.

It is an interesting application of Minsky's paper that if the initial cell of every tape submitted to a machine is uniquely distinguishable then every set of m-tuples definable by a Turing machine is representable by an n-head machine with at most m+2 heads. This result follows from letting two heads in conjunction with the uniquely distinguishable initial cells of their tapes represent the total state transition of the Turing machine via Minsky and letting the remaining m heads be placed one head per tape and read and move according to the inputs and the state of the Turing machine.

6.6.1.2 State Reduction

If one confines one's interest to 1-way machines then the classical reduction methods as introduced by Moore[42] suffice to yield the minimum state equivalent of any machine. The general problem for 2-way machines is, however, unsolved. Namely, given a representable set of inputs, no method is known for securing a minimum state machine to represent the set.

Some remarks can be made about reducing the number of states in a given machine. All inaccessible states can be eliminated from any machine. All inaccessible transitions can be made "don't care" transitions. Further, given a machine $\mathcal{M}$ possibly with some don't care transitions one can ignore the head movement associated with each transition and apply a conventional state reduction procedure to $\mathcal{M}$ thus partioning the state set of $\mathcal{M}$ into equivalence classes of mergable states; given any two states in the same equivalence class one proceeds to merge them if and only if for any input the transitions leaving each state on that input have identical head movements. The above technique of state reduction never alters the number of heads in a given machine.

In general the head reduction and state reduction problems are not independent - consider the machines $\mathcal{M}_{6.1}$ and $\mathcal{M}_{6.2}$ shown in Figures 103 and 105 respectively: $\mathcal{M}_{6.1}$ has 1-head and four states while $\mathcal{M}_{6.2}$ has four heads and one state (A and R are not counted here). $\mathcal{M}_{6.1}$ is a 1-way 1-head machine in reduced form but $\mathcal{M}_{6.2}$ is a 2-way 4-head machine with fewer states than $\mathcal{M}_{6.1}$. Careful inspection will show that $G(\mathcal{M}_{6.1}) = G(\mathcal{M}_{6.2}) = aa^*bb^*cc^*B$.

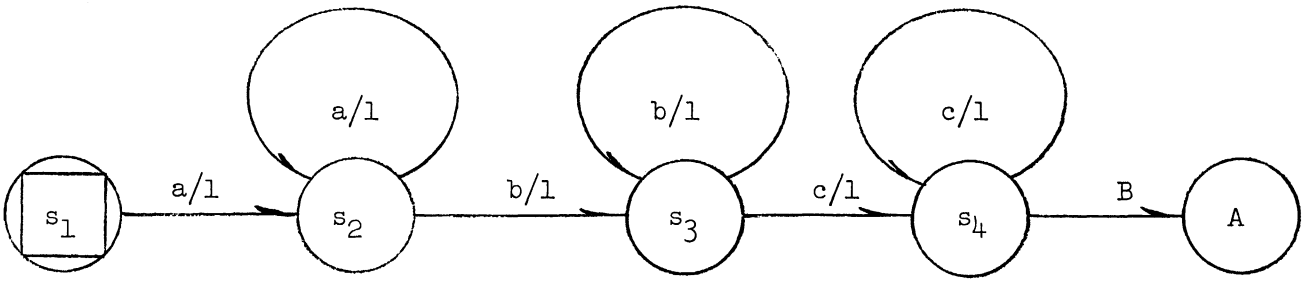


Figure 103. Machine $\mathcal{M}_{6.1}$.

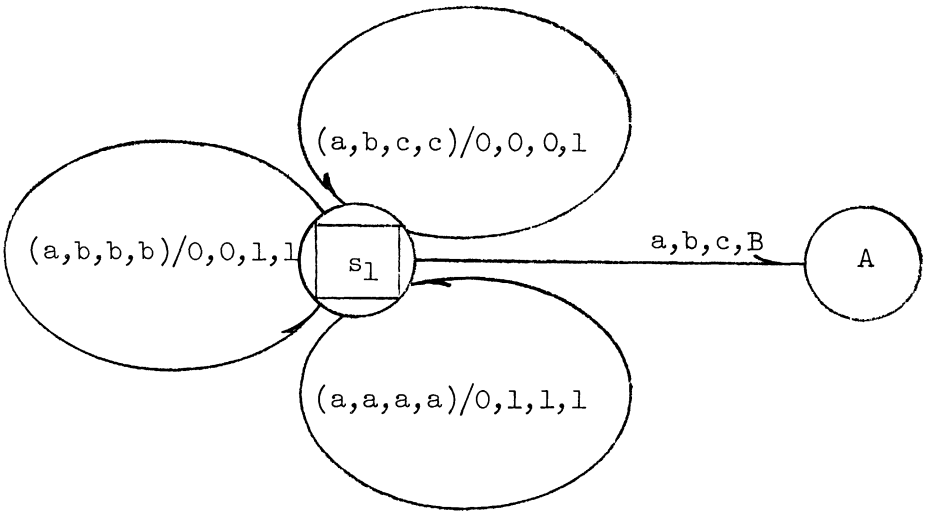


Figure 104. Machine $\mathcal{M}_{6.2}$.

6.6.1.3 Speed Reduction

If A_1 is a set of tapes representable by some 1-dim 1-head machine $\mathcal{M}_1$ then via theorem 5.10 one knows that $\mathfrak{J}(\mathcal{M}_1)$ is the fastest (or one of a set of the fastest) machine that recognizes any tape in A_1. If A_2 is a set of D-dim tapes representable by some n-head machine $\mathcal{M}_2$ then if $G(\mathcal{M}_2)$ is finite one can construct a machine $\mathcal{M}_2'$ such that $G(\mathcal{M}_2) = G(\mathcal{M}_2')$ and such that no machine is faster than $\mathcal{M}_2'$. [$\mathcal{M}_2'$ will be provided with a

suitably large number of heads that will fan out from the initial cell of the tape such that after each machine cycle an increasing region of tape will have been scanned; if g is any generator in $G(\mathcal{M}_2)$ and if md(g) is the Manhattan distance to the cell of g farthest away from the initial cell then $\mathcal{M}_2'$ will recognize g in md(g) machine cycles - no machine could do it faster. If A_3 is a set of D-dim tapes representable by some n-head machine $\mathcal{M}_3$ and such that $G(\mathcal{M}_3)$ is infinite then in general it appears that there is no single machine equivalent to $\mathcal{M}_3$ and which detects all $g \in G(\mathcal{M}_3)$ faster than any other machine; rather it seems that for any machine $\mathcal{M}_3'$ computationally equivalent to $\mathcal{M}_3$ there is another machine $\mathcal{M}_3''$ such that for all inputs $\mathcal{M}_3''$ is just as rapid as $\mathcal{M}_3'$ and for some inputs $\mathcal{M}_3''$ is more rapid.

One might also expect that the state reduction and head reduction problems are not independent of the speed reduction problem.

6.6.2 Representability Problems

In the synthesis theorems of Section 6.4 one was required to begin with a realizable RE; failure to do so resulted in an "improper" machine, i.e., a machine in which some of the states had several transitions leaving it on the same input, each transition having a different associated head movement. In general it appears that non-realizable RE's cannot be used as a basis for machine synthesis; however, some techniques can be tried in an effort to procure "proper" machines to represent sets of inputs based on non-realizable RE's. For example:

If $\mathcal{M}$ is the improper machine derived in an attempt to represent a set of inputs based on β, a non-realizable RE,

1) if one of the offending transitions goes to A then the remaining offending transitions that leave the same state as the transition going to A can be deleted from $\mathcal{A}$ without affecting $T(\mathcal{A})$;

2) if any offending transition can be shown to be inaccessible then that transition can be deleted from $\mathcal{A}$ without affecting $T(\mathcal{A})$;

3) if the number of times the machine will pass through a state s from which offending transitions emanate is finite for all inputs then by expanding the number of heads and states of the machine one can construct a new machine $\mathcal{A}'$ that is proper in regard to all transitions leaving s and equivalent to $\mathcal{A}$ [$\mathcal{A}'$ operates by dividing part of its head set every time it embarks on the offending transitions; a part of the set follows each transition; since $\mathcal{A}$ passes through s a finite number of times $\mathcal{A}'$ will have to split its head set at most a finite number of times];

4) if $\beta^{\Psi f}$ is finite then one can always construct a realizable RE β' such that $\beta'^{\Psi f} = \beta^{\Psi f}$, thus the machine $\mathcal{A}'$ based on β' will be equivalent to $\mathcal{A}$; if $\beta^{\Psi f}$ is not finite one can still search for a realizable RE β' such that $\beta'^{\Psi f} = \beta^{\Psi f}$ in which case a machine derived from β' will be equivalent to the machine derived from β.

6.7 SUMMARY

Section 6 attempts to treat the problems associated with multiple head finite state machines. It begins, in Section 6.2, by (1) defining n-head machines, (2) defining the form of their inputs, and (3) prescribing the manner in which these machines accept and reject inputs. As defined in this section n-head machines are the same as classical single head automata, as understood by say McNaughton and Yamada, with the restrictions and additions that (1) there can be only two final states, namely ACCEPT and REJECT, (2) if the machine enters one of these final states it halts operation immediately, (3) each transition in these machines is specified by the present state of the machine and by the n-tuple of input symbols scanned by the heads, (4) each transition is accompanied by an n-tuple of head movements which need not be identical for all transitions in a given machine, and (5) the inputs are multi-dimensional tapes that in general can extend in all directions from the initial (or starting) cell of each tape.

Resulting from these machines' ability to accept and reject inputs is the notion of using them to define sets of inputs depending on whether an input set is accepted or rejected by a particular machine. Section 6.2 develops the concept of generators as it applies to sets of defined inputs and shows that for each machine its generator set is equivalent to its set of defined inputs.

It is evident from the examples included in Section 6.2 that n-head machines are more powerful than single head machines. It is further demonstrated that even with the restrictions that (1) n-head machines always

start with their heads on the initial cells of their input tapes and (2) all movements are one-cell-at-a-time-in-a-coordinate-direction, nevertheless the computational power of the machines is just as great as with machines that do not start on initial tape cells and whose head movements may not all be unit moves.

Section 6.3 introduces a language which is later shown to be equivalent to n-head machines in its ability to define sets of tapes. The language presented includes the already well known language of regular expressions which has been augmented to include the newly defined operations of column alphabets, indexed alphabets, and the separation, fold and cover of tapes. These newly defined operations correspond in a natural manner to the structure of n-head machines - i.e.,column alphabets correspond to multiple heads, indexed alphabets correspond to the movements associated with each head, separation corresponds to several distinct heads working simultaneously, fold corresponds to 2-way D-dim head movements and cover corresponds to several distinct heads scanning the same tape.

In Section 6.4 anequivalence is developed in the form of twelve theorems between the input generators defined by n-head machines and particular expressions in the language of Section 6.3. The theorems constitute six analysis-synthesis pairs which treat n-head machines of various complexities beginning with 1-way 1-dim 1-head machines and concluding with 2-way D-dim n-head m-tape machines. Aside from their academic value these theorems are useful in that given a desired set of generators if one can represent them by a suitable expression in the language then the synthesis theorems allow direct implementation of a machine possessing the given generators.

Section 6.5 deals with a number of questions relating to n-head machines. It begins by presenting two algorithms - one to decide if a given n-head machine is 1-way, the other to decide if a given regular expression is realizable; both of these algorithms are necessary for execution of some of the theorems in Section 6.4. Section 6.5 develops a 1-way 2-head equivalent of every 2-way 1-dim 1-head machine. Note that under the assumptions of this paper a 2-way automaton is allowed to scan both sides of the initial cell; under this condition the fifteenth theorem of Rabin and Scott becomes invalid and is replaced by Theorem 5.1 of this section.

The work of Rabin and Scott is extended in Section 6.5 to include all n-head machines. The results of Theorems 5.3 to 5.6 can be summarized as follows:

> The existence or non-existence of effective procedures to answer certain decision questions partitions the class of n-head machines into three categories as shown in Table 6.

TABLE 6

THE EXISTENCE OF EFFECTIVE PROCEDURES FOR DECISION PROBLEMS

Type of Machine / Decision Problem	1-Dim 1-Head	D-Dim 1-Head $D \geq 2$	General n-Head
Particular Input Problem	Yes	Yes	No
Emptiness, State Accessibility and Transition Accessibility Problems	Yes	No	No

Section 6.5 continues by presenting a number of theorems treating the Boolean properties of n-head machines and concludes with a number of theorems treating the relative speeds of computationally equivalent machines. The speed theorems are developed within the milieu of 1-dim machines. Some but not all of the speed theorem results can be extrapolated to multi-dimensional machines. The speed theorems can be paraphrased as follows:

> For each 1-head machine $\mathcal{M}$ working over 1-dim tapes there is a 2-head 1-way machine $\mathcal{J}(\mathcal{M})$ which is computationally equivalent to $\mathcal{M}$. $\mathcal{J}(\mathcal{M})$ is always as fast as $\mathcal{M}$ and is faster than $\mathcal{M}$ if and only if $\mathcal{M}$ reverses or halts its head movement during examination of an input. There are sets of 1-dim tapes $A_1, A_2, \ldots A_j, \ldots$ such that if $\mathcal{M}_j$ is any 1-head machine defining A_j then $\mathcal{J}(\mathcal{M}_j)$ is faster than $\mathcal{M}_j$ for all inputs. Furthermore, the A_j can be defined such that for all inputs in A_j $\mathcal{J}(\mathcal{M}_j)$ is faster than $\mathcal{M}_j$ by an arbitrarily large difference and for all inputs in some infinite subset of A_j $\mathcal{J}(\mathcal{M}_j)$ is faster than $\mathcal{M}_j$ by an arbitrarily large factor. For any set A of 1-dim tapes definable by 1-head machines and for any particular tape t_o in A there is a 1-head machine $\mathcal{M}_o$ that defines A and has the property that no machine that defines A is more than three times faster than $\mathcal{M}_o$ in recognizing t_o.

Section 6.6 contains some suggestions for further study. These suggestions lie in the areas of (1) head-state-speed reduction and (2) representability problems. A number of partial results are included with each suggestion. Some of the partial results are:

1) It is evident that the number of heads and states a machine has and the speed with which it recognizes inputs are not independent quantities. The work of previous authors on these reduction problems has been confined to 1-way 1-dim 1-head machines; expansion of the field of inquiry to 2-way n-head machines seems reasonable and re-opens many questions considered answered for the 1-way case.

2) Given any set of inputs one can ask if an n-head machine exists that defines the set. Using the work of Minsky for direction one can conclude that if the initial cells of all tapes are uniquely distinguishable by machines - as they must be by us - then all sets of m-tuples of tapes definable by Turing machines are definable by finite state machines with at most m+2 heads. If, however, as this paper has assumed, the initial cell is not uniquely distinguishable by the machines then it is an open question in general as to whether one can decide given a set of inputs if an n-head machine exists that defines the set.

6.8 SOME PRACTICAL PROBLEMS WITH n-HEAD MACHINES

The purpose of this section is to treat in an informal manner some of the practical considerations that came to T. F. Piatkowski's attention while he was conducting gedanken experiments on multiple head automata. The work of this section in no way summarizes nor replaces material presented in Sections 6.1 through 6.7, but rather complements the prior work and can most probably be best understood after a reading of those sections.

The topics to be considered are six in number and in the order of presentation are:

1) The finite nature of real-life problems
2) Non-implication of Section 6.4
3) The advantage of end-marks on tapes
4) "Time" as a tape dimension
5) A consequence of touched heads
6) Application of n-head machines (two heads are better than one ... sometimes!)

6.8.1 The Finite Nature of Real-Life Problems

The abstract development of the theory of multiple-head machines as presented in Sections 6.1 through 6.7, presumes that, in general:

1) input tapes can be and, in fact, are infinite in extent, and
2) the reading heads of n-head machines can maneuver themselves arbitrarily far apart.

In real life both of these presumptions are false and the negation of either one of them leads to a precise statement of the minimum number of heads needed

to recognize any real-life set of tapes. Consider the following arguments:

a) While theoretical problems submitted to n-head machines may be arbitrarily large, real-life problems are finite; i.e., due to the finite life of machines and their operators, we are interested only in sets of tapes in which the size of the input tapes and the deviation of machine computation will be confined to ranges known a priori. Thus, any "real-life" set of n-tuples of tapes will have a finite number of generators and can be shown to be recognized with machine carrying n-heads or less (i.e., at most one head per tape).

b) Since each head of any n-head machine is required to keep within communication distance of the central control mechanism of the machine and since practical limitations place an upper limit on such a distance (i.e., telephone wires can only be so long, or radio signals sent so far in a reasonable time, etc.), it follows that in all real life n-head machines, heads working on the same tapes are "bound" and that, consequently (via the results in Section 6.6), any "real-life" n-head machine can be replaced with an equivalent machine that requires at most one head per input tape.

The above arguments indicate that the finite nature of real-life problems dictates that any "practical" problem, if solvable by any n-head machine, is solvable by some n-head machine with at most one head per tape. However, such arguments are true in theory only for real life also imposes a priori bounds on the number of internal states any practical n-head machine can

possess. This imposes a further restriction on the number of sets of tapes recognizable by practical n-head machines. We are therefore confronted with the result that of all sets of inputs theoretically recognizable by n-head machines, only an infinitesimal subset are recognizable by real-life machines. Furthermore, constraints on the allowable separation between heads and the total number of integral states together determine the minimum number of heads one can employ to recognize a given set of inputs (in general, for computationally equivalent machines as the number of heads goes up the number of states goes down and vice versa).

6.8.2 Non-Implications of Section 6.4

Following the procedures presented in Section 6.4, one can produce for each n-head finite state machine a closed expression that precisely describes the set of inputs that particular machine recognizes. However, we cannot determine, in general, if the set of inputs any such expression represents is empty or not; that follows from the results of Section 6.5 which established that for 2-way multi-head or 2-way multi-dimensional 1-head machines, the emptiness question is not answerable.

It should be recognized, however, that our inability to discern if a given expression represents an empty set or not is not a fault of the language; indeed Section 6.5 substantiates that any language used to describe n-head machines must contain this inadequacy.

6.8.3 The Advantage of End-Marks on Tapes

End-marks are special symbols, recognizable by some n-head machines, which are used solely to delineate the regions of the input tapes to which the head

movements are to be confined. When used correctly, each input tape will consist of a matrix (not necessarily a parallelepiped) of cells containing an input cell and surrounded by end-marks. Any machine recognizing end-marks will constrain its heads to remain within the region the end-marks encompass, i.e., when the head of such a machine encounters an end-mark it retreats backward into the permitted region. From the above remarks it is evident that n-head machines utilizing end-marks form a proper subset of all n-head machines.

In contrast to the case for general n-head machines, it turns out that for all end-mark n-head machines, the "particular input" decision problems is answerable. This follows from the fact that for a given end-mark machine and given set of inputs (with end-marks), there is a finite computable number of combinations of internal state and head positions in which the machine can be; thus, one can always set an upper bound on the number of cycles the machine will take to accept or strongly reject a given input.

It should be noted that, in general, the emptiness, state accessibility, and transition accessibility questions are not answerable for end-mark machines. This is proved in Theorem 19 of Rabin and Scott, the proof of which was for end-mark machines.

6.8.4 "Time" as a Tape Dimension

There are two fundamentally different ways of presenting input symbols to the heads of any machine. The first is to let the heads scan over the cells of a spatial tape; the second is to present an input symbol to each head at regularly spaced intervals of time. In the first instance, the tape exists in a two-way manner in reading it. In the second instance, the tape exists in

time only and due to the irreversible manner in which time passes, the input is one-dimensional and the head movements one way.

Thus, all n-head machines which read inputs in time (not space) are one-dimensional one-way and, therefore,

1) need at most one head per input, and

2) always yield answers to the "particular input," emptiness, state accessibility and transition accessibility questions.

6.8.5 A Consequence of Touched Heads

In Sections 6.1 through 6.7 for any machine the heads that work on the same tape oblivious of each other; i.e., any number of heads can occupy the same tape cell simultaneously and not be aware of each other's presence. Such a condition has some very practical drawbacks and we can very easily construct a model in which reading heads can detect each other's presence. In such a case, it turns out that every set of Turing definable n-tuples of tapes can be recognized with a multiple-head machine having at most m + 3 heads. This result follows from the work of Minsky and can be implemented by allowing three heads to simulate the internal state of the Turing machine in question and the remaining m-heads to read the m-input tapes, one head per tape.

6.8.6 Applications of n-Head Machines

It is quite difficult to conceive of all the possible applications of n-head machines. Finite state machines are, in essence, pattern recognition devices and in the most general sense, they can be said to recognize the symmetries of regular expressions over column alphabets. These symmetries include all of the simple n-dimensional spatial symmetries such as symmetries with

respect to points, lines, planes, etc. This property of these machines suggests that n-head machines could play a useful role in studies concerned with pattern recognition. Piatkowski has not carried out any such studies nor has he any knowledge of others having done so using n-head machines.

7. REFERENCES

1. Moore, E. F., "Shortest Path Through a Maze," Annals of the Computation Laboratory of Harvard University, Harvard University Press, Cambridge, Mass., Vol. 30, pp. 285-92 (1959).

2. Lee, C. Y., "An Algorithm for Path Connections and Its Applications," IRE Trans. on Electr. Computers, Vol. EC-10, No. 3, pp. 346-65, September, 1961.

3. Loberman, H., and Weinberger, A., "Formal Procedures for Connecting Terminals with a Minimum Total Wire Length," ACM Proc., Vol. 4, p. 428 (1957).

4. Von Neumann, J., "Probabilistic Logics," Automata Studies, Princeton University Press (1956).

5. Miller, R. E., and Selfridge, J. L., "Maximal Paths on Rectangular Boards," IBM Journal, Vol. 4, No. 5, p. 479, November, 1960.

6. Wilcox, R., and Mann, W., Redundancy Techniques for Computing Systems, Spartan Books (1962).

7. Hald, A., Statistical Theory with Engineering Applications, Wiley and Sons (1952).

8. Holland, J., "Iterative Circuit Computers," Proc. W.J.C.C., May, 1960, p. 259.

9. Unger, S. H., "A Computer Oriented Towards Spatial Problems," Proc. IRE Trans., Vol. 46, p. 1749, October, 1958.

10. Newell, A., "On Programming a Highly Parallel Machine to be an Intelligent Technician," Proc. W.J.C.C., p. 267, May 1960.

11. Arden, B. W., Galler, B. A., and Graham, R. M., "An Algorithm for Translating Boolean Expressions," J. of the ACM, p. 222, April 1962.

12. Arden, B. W., and Graham, R. M., "On GAT and the Construction of Translators," Comm. of the ACM, p. 24, July 1959.

13. Bull Gamma 60 Reference Manual, No. 0943(6-10), publ. by Compagnie des Machines Bull, Paris.

14. Holland, J. H., "Iterative Circuit Computers," Proc. of the 1960 W.J.C.C., p. 259.

15. Schwartz, E. S., "An Automatic Sequencing Procedure with Application to Parallel Programming," J. of the ACM, p. 153, October 1961.

16. Slotnick, D. L., Borch, W. C., and McReynolds, R. C., "The Solomon Computer," Proc. of the 1962 F.J.C.C., p. 97.

17. Squire, J. S., and Palais, S. M., "Programming and Design Considerations of a Highly Parallel Computer," Proc. of the 1963 S.J.C.C.

18. Hu, T. C., "Parallel Sequencing and Assembly Line Problems," J. Operations Res., Vol. 9, No. 6, pp. 841-8 (1961).

19. Church, A., Introduction to Mathematical Logic I, Princeton University Press (1956).

20. Church, A., "An Unsolvable Problem of Elementary Number Theory, see footnotes, Amer. J. Math (1936), Vol. 58, pp. 345-63.

21. Davis, M., Computability and Unsolvability, McGraw-Hill (1958).

22. Detlovs, V. K., "Normal Algorithms and Recursive Functions," Doklady Akad. Nauk. SSSR (Proc. of the Acad. of Sci., USSR), Vol. 90, No. 3, pp. 249-52 (1953).

23. Kleene, S. C., Introduction to Metamathematics, Van Nostrand (1952).

24. Markov, A. A., Theory of Algorithms, Acad. of Sci., USSR (1954). Translation available from U.S. Dept. of Commerce.

25. Smullyan, R. M., Theory of Formal Systems, Princeton University Press, Annals of Mathematical Studies 47 (1961).

26. Naur, P., et.al., "Report on the Algorithmic Language ALGOL 60," Comm. of the ACM, Vol. 3, No. 5, May 1960.

27. Henie, F. C., "Iterative Arrays of Logical Circuits," J. Wiley, New York (1961).

28. McCluskey, E. J., "Iterative Combinatorial Switching Networks—General Design Considerations," IRE Trans. on Electr. Computers, Vol. EC-7,

29. Henie, F. C., "Analysis of Bilateral Iterative Networks," IRE Trans. on Circuit Theory, Vol. CT-6, p. 35 (1959).

30. Holland, J., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously," Proc. E.J.C.C., p. 108, December 1959.

31. Amarel, S., Review of "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously," by J. Holland, and "Iterative Circuit Computers," by J. Holland, IRE Trans. on Electr. Computers, Vol. EC-9, p. 384, September 1960.

32. Bauer, W., "Horizons in Computer System Design," Proc. W.J.C.C., p. 41, May 1960.

33. Squire, J., "A Comparative Study of Module Communications," Internal Report, The University of Michigan's Information Systems Laboratory (1962).

34. Carroll, A. B., and Confort, W. T., "The Logical Design of a Holland Machine," Internal Report, The University of Michigan, Electrical Engineering Department (1961).

35. West, G. P., and Koerner, R. J., "Communications Within a Polymorphic System," Proc. W.J.C.C., p. 225, December 1960.

36. Carlsen, R. A., Feingold, M. G., and Fife, D. W., "A Simulation of the AN/FSQ-27 Data Processing System," RADC-TR-61-254, The University of Michigan, Department of Electrical Engineering (1961).

37. Holland, John H.,"Outline for a Logical Theory of Adaptive Systems," ACM J., p. 297, July 1962.

38. McCormick, B. H. and Divilbiss, J. L., "Tentative Logical Realization of a Pattern Recognition Computer," Engineering Summer Conference Report, The University of Michigan (1961).

39. Harrison, M. A., "Electrical Engineering 467 Class Notes," The University of Michigan, Electrical Engineering Department, October 1961.

40. Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata, Princeton University Press, Automata Studies, Annals of Mathematics Studies (1956).

41. McNaughton, R. F., and Yamada, H., "Regular Expressions and State Graphs for Automata," IRE Trans. on Electr. Computers, Vol. EC-9, No. 1., March 1960.

42. Moore, E. F., Gedanken-Experiments on Sequential Machines, Princeton University Press, Annals of Mathematics Studies, Automata Studies, (1956).

43. Rabin, M. O., and Scott, D., "Finite Automata and Their Decision Problems," IBM J. of Res. and Development, Vol. 3, No. 2, April 1959.

44. Shepardson, J. C., "The Reduction of Two-Way Automata to One-Way Automata," IBM J. of Research and Development, Vol. 3, No. 2, April 1959.

45. Minsky, M. L., Recursive Unsolvability of Post's Problem of "Tag" and Other Related Topics, Annals of Mathematics, Vol. 74, No. 3, pp. 437-55 (1961).