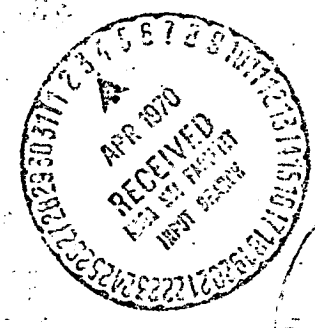
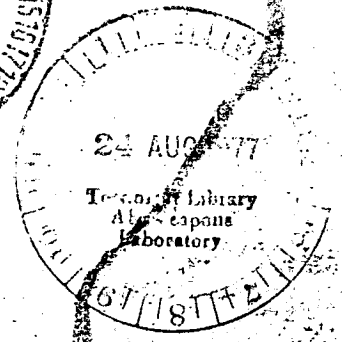
RECOGNITION AND REPRESENTATION OF PARALLEL PROCESSABLE STREAMS IN COMPUTER PROGRAMS





CHAPTER I

Introduction

1.1 Motivation

Computational processes have traditionally been approached in what is essentially a serial or sequential manner. The entire process--beginning with the writing of a program and concluding with the actual execution of the program--has been based on the assumption that an instruction stream is processed one step at a time.

Initial efforts in the field of information processing with regard to computers were limited mainly to technological improvements of existing facilities. Some of the goals in this area included an increase in computational speed and a decrease in physical size and power consumption. After these efforts were successfully launched, however, it was realized that the serial approach to computational processes did not result in efficient use of available resources. This realization culminated in techniques such as multi-programming and time-sharing.

Although these techniques did yield higher resource utilization, the speed of computational processes was now a function of technological limitations. Further progress in this area resulted in the concept of Parallel Processing, a concept whereby several segments of an instruction stream are processed concurrently.

Common to any discussion on parallel processing is an assumption of the existence of a large number of processing units and memories controlled by an operating system. In the past, the prohibitive expense

implied by this assumption sealed the fate of any progress in this area. State-of-the-art advances, however, and in particular, anticipated advances generated by Large Scale Integration, have given fresh impetus to research in this area. Given a program with a large degree of inherent parallelism, the theoretical time required for problem solution can be reduced " ... by a factor at least as great as the number of processors" [1].

The motives for parallel processing include the following:

1. Real-time urgency. Parallel processing can increase the speed of computation beyond the limit imposed by technological limitations.

2. Reduction of turnaround time of high priority jobs.

3. Reduction of memory and time requirements for "housekeeping" chores. The simultaneous but properly interlocked operations of reading inputs into memory and error checking and editing can reduce the need for large intermediate storages or costly transfers between members in a storage hierarchy.

4. An increase in simultaneous service to many users. In the field of the computer utility, for example, periods of peak demand are difficult to predict. The availability of spare processors enables an installation to minimize the effects of these peak periods. In addition, in the event of a system failure, faster computational speeds permit service to be provided to more users before the failure occurs.

The subject of parallel processing can be divided into three separate problem areas. The first problem occurs at the time a program is written and involves an explicit indication of parallelism within a program. In some cases this indication can be accomplished by means of the programming language itself. The second problem involves detection at compilation time of the segments of a program which can be processed

concurrently. The final problem involves communication and utilization of this information by the computing system.

Chapter I of this paper introduces the general subject of parallel processing by defining the concepts of explicit and implicit parallelism and discussing problems associated with these areas.

Chapter II is primarily a survey of existing techniques for detection of sub-task parallelism as exemplified by arithmetic expressions. Chapter II concludes with the introduction of a new technique for detecting sub-task parallelism in arithmetic expressions.

The major portion of this paper is contained in Chapter III. In this chapter a new technique is introduced for detecting parallel processable segments within a single program. The technique has been incorporated in the form of a FORTRAN recognizer program. This recognizer inputs a source program also written in FORTRAN and outputs the parallel processable segments of the source program. Details, limitations, and overall flow charts of the recognizer are also contained in Chapter III. Detailed flow charts of the recognizer are given in Appendix A.

Throughout the remainder of this paper two thoughts should be kept in mind. In order for parallel processing to be truly effective, the overhead which accompanies this mode of operation must be minimized. Thus any approach to this subject must include machine organization which can take advantage of parallel processable streams. In addition, it is felt that successful recognition of these streams at compilation time will minimize overhead during execution time.

1.2 Hierarchies of Parallelism

Within an individual program, parallelism can exist at several

levels--from the level of statements or groups of statements of procedural languages to the level of micro operations. Discussion in this paper will be confined to intra-program parallelism as opposed to program parallelism. The latter term refers to the type of processing in which independent programs are processed concurrently; it is intended to be synonymous with the conventional meaning of multiprocessing in a multiprogrammed environment. Intra-program parallelism, on the other hand, refers to the type of processing in which a single program can be partitioned into tasks which can be performed in parallel. Two types of intra-program parallelism will be considered: global and local.

In the global type of parallel processing a program is partitioned into tasks which can be performed in parallel. Throughout this discussion the terms "task" and "process" are intended to mean a self-contained portion of a computation which once initiated can be carried out to its completion without the need for additional inputs. The completion of a task is significant in that its occurrence can initiate the execution of another set of tasks. An example of such a task would be a single FORTRAN statement or a set of such statements. Consider, for example, the hypothetical FORTRAN program shown in Figure 1 which illustrates both global and local parallelism.

Because the two arithmetic expressions have independent input sets, they can theoretically be executed in parallel. This is an example of independent tasks consisting of single statements. The two subroutine CALL statements, on the other hand, are examples of independent tasks which can consist of many statements. Again because of the disjoint input sets, these two statements could theoretically be executed in parallel.

The type of reasoning suggested here has been investigated by

```
                    .
                    .
      READ 100, A, B, C, D
                    .
                    .
      X = A**2 - 2.0*A*B + B**2
      Y = C**2 + 2.0*C*D + D**2
                    .
                    .
                    .
   10 Z = (A*B) + (C*D)
                    .
                    .
                    .
      CALL SUB1 (A, B, E)
      CALL SUB2 (C, D, F)
                    .
                    .
      END
```

Figure 1. Sample FORTRAN Program Illustrating Global and Local Parallelism.

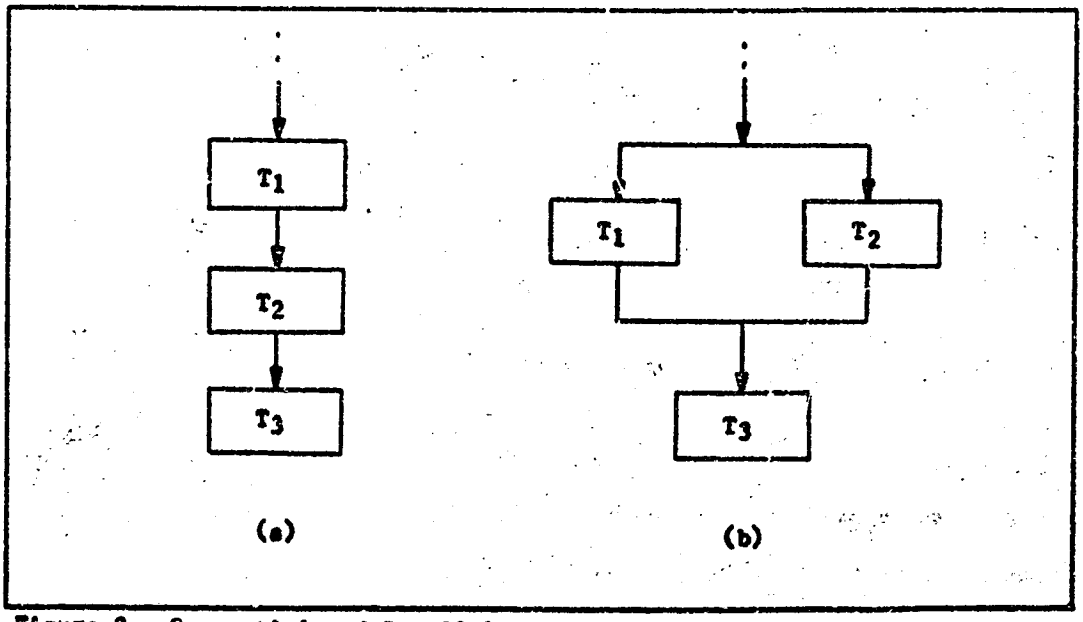


Figure 2. Sequential and Parallel Execution of Two Tasks.

several sources; a reference prominent in this area is that due to Bernstein [3]. Consider several tasks, T_i, of a sequentially organized program illustrated by a flow chart as shown in Figure 2(a). If the execution of task T_3 is independent of whether tasks T_1 and T_2 are executed sequentially or in parallel as shown in Figure 2(b), then parallelism is said to exist between T_1 and T_2.

With reference to Figure 1, parallelism on a local level can be illustrated by statement 10. As shown in Figure 3, "sub-tasks" 10a and 10b within the task depicted by statement 10 can be executed sequentially, (a) or in parallel (b) without altering the outcome at the completion of the task.

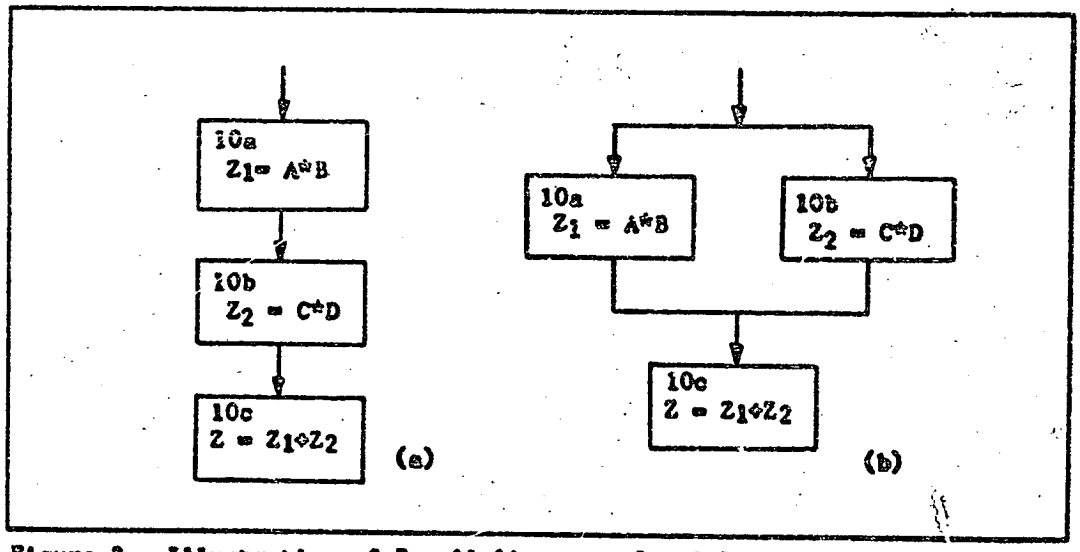


Figure 3. Illustration of Parallelism on a Local Level.

A second illustration of local parallelism is given by the individual sub-tasks within subroutine SUB1. Conceivably, the subroutine could itself consist of statements having the same form as the example arithmetic expressions within the calling program. These sub-tasks

within the subroutine could themselves be executed in parallel.

1.3 Explicit and Implicit Parallelism

The problem of recognising parallel processable tasks within a program is difficult primarily because programs are usually written in what is essentially a serial manner. The solution to this problem can be approached from two directions. The first approach places the burden upon the programmer and requires an Explicit indication of parallelism. This information is generated by means of additional instructions within the programming language itself. The second approach relieves the programmer of any additional duties and relies totally upon indicators existing in the program itself and is known as Implicit parallelism.

Needless to say, each approach has certain advantages. If parallel processing is ever to be truly successful, it can only do so by avoiding the overhead which a complex compiler and a supervisory program would create. By allowing the programmer to indicate the parts of a program which can be executed in parallel, much of this undesirable overhead can be prevented. This approach, however, is of little value when one considers the many thousands of programs already in existence, since it is safe to assume that most would not be recoded in order to take advantage of inherent parallelism. This problem would be minimised, however, if parallelism within these programs could be detected using implicit recognition techniques. This cannot be accomplished, however, without a corresponding increase in system complexity.

A problem common to both approaches is that concurrently executable processes may compete for the same resources (memory, I/O, etc.). Thus the methods of indicating parallelism to the computing

system and the benefits derived by parallel processing are intimately related to the overall architecture of the computing system, the task management algorithms, etc. [5].

When parallel tasks use the same hardware resources like memory and I/O, access to which is serial, they have to be queued up for service. The solution is to overlap the parallel processes such that the critical resource will not create a bottleneck. Dijkstra, Knuth, and Coffman [14, 15, 16] have developed efficient scheduling procedures for using common resources.

When parallel tasks manipulate common data (e.g., one task inputs data whereas another edits the data), the sequence and the ordering of the operations with respect to each other may be important. Thus when sequentially coded tasks are prepared for parallel processing, it is important to determine whether or not the outputs of parallel paths are the same regardless of the order and sequence of execution. If the results of this determination are dissimilar, then task ordering information must be incorporated into the execution of these tasks.

1.4 Explicit Parallelism

In the explicit approach to parallelism, the programmer himself indicates the portions of a program which can be executed in parallel. This is normally done by providing the programmer with additional instructions which identify the parallel processable streams. One of the most frequently mentioned explicit approaches to parallelism is the FORK and JOIN [8]. FORK is an instruction which indicates the initiation of parallel tasks; JOIN is an instruction which initiates a new task only after the processes initiated by the FORK are completed and merge into the

JOIN. The use of these instructions is illustrated in Figure 4.

The FORK at location 100 means: "Set the contents of location 299 to 2; then fork to 101 and 200." The "2" in the instruction indicates the number of processors which the FORK at location 100 will activate (if they are available). As a processor completes its instructions, it branches to location 299 where it decrements the counter by 1. If the result is not zero, the processor is released. The processor which sets

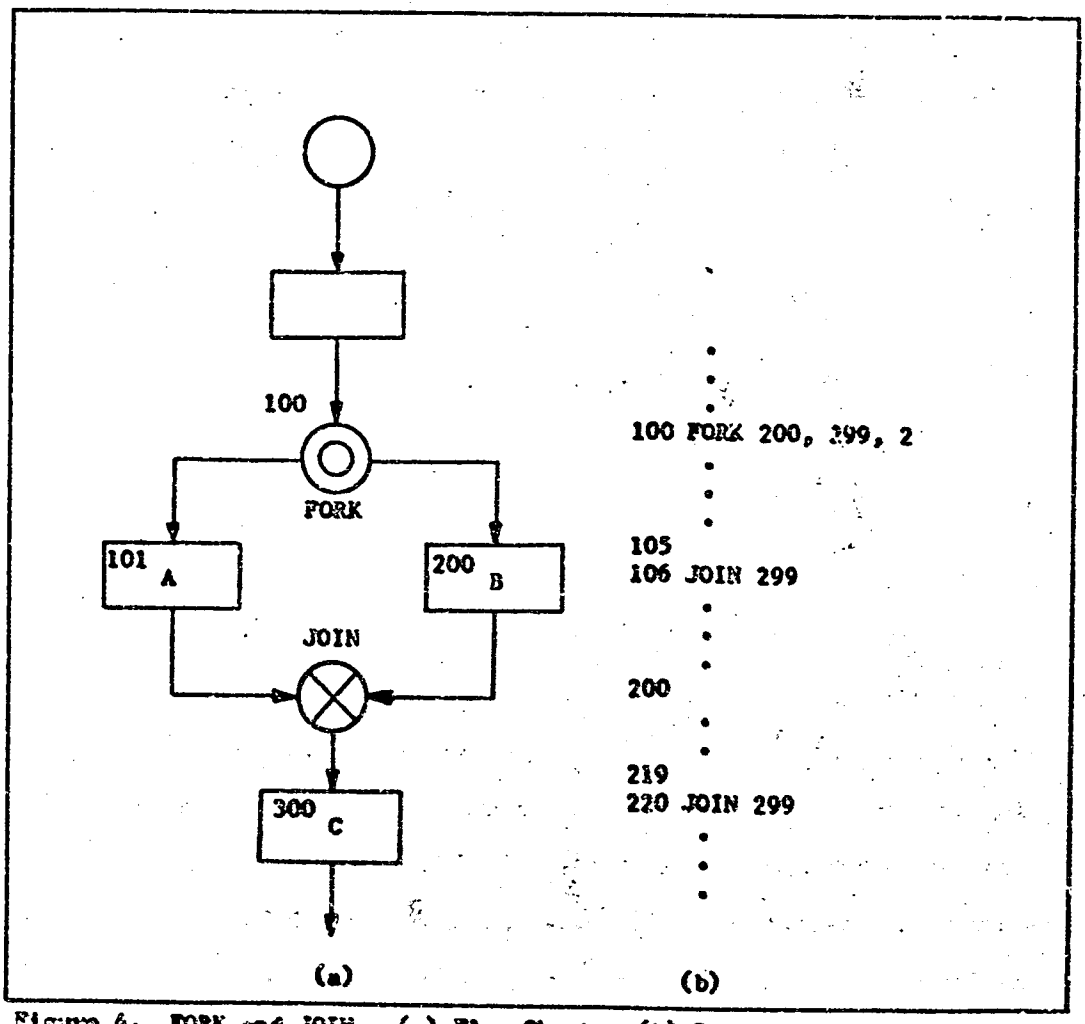


Figure 4. FORK and JOIN. (a) Flow Chart. (b) Instructions

the counter to zero simply transfers to location 300 and continues. A distinct advantage of this approach is that it is simple and natural.

The basic premise of the PARALLEL FOR technique [9] is that FOR statements in ALGOL programs and similar constructs in other languages permit exploitation of parallel activity. As an example, consider the addition of two I x J matrices M1 and M2. Mathematically this would be represented as

$$M \leftarrow M1 + M2$$

In ALGOL, the instructions to do this would read as shown in Figure 5. Since the pairwise additions are independent, they can be performed in parallel. The implementation of this process as a PARALLEL FOR is shown in Figure 6. B1 initializes the generation of the values of the parameters

```
FOR K1 := 1 STEP 1 UNTIL I DO
FOR K2 := 1 STEP 1 UNTIL J DO
M[K1, K2] := M1[K1, K2] + M2[K1, K2] ;
```

Figure 5. ALGOL Instructions for Addition of Two Matrices.

of the FOR statement and establishes the new parallel path counter (PPC). PREP is a function performed before a new set of parallel paths begins. If other PPC's exist for this process, they are pushed down, which allows sets of parallel paths to be nested. PREP sets the new PPC to one. B2 is the stepping or incrementing function. B3 is the end test and transfers to JOIN when all paths have been started. B4 starts a new parallel path. B5 is a parallel path. B6 is the JOIN. The function AND(L) is a simple two-way fork; it requests the controller to start a parallel sequence at L and add one to the current PPC for this process. JOIN is a function used

to terminate a set of parallel paths. When executed, it reduces the current PPC for the process by one. If PPC is then zero, it is popped up and processing continues. If PPC is not zero, meaning that more paths remain to be completed, it releases the processor executing the JOIN. For the example given, the implementation of this process would result in J simultaneous pairwise additions as specified by the inner FOR statement.

An attractive feature of this scheme is that it is independent of the number of processors available. At any given time the maximum number of active paths equals the number of available processors. When a processor terminates its duties it is either released or it initiates a new parallel path.

The FORK and JOIN and the PARALLEL FOR are but two examples of explicit indication of parallelism. Because of the relative simplicity of indicating inherent parallelism explicitly, there has been an increasing amount of effort in this direction. PL/1, for example, provides the TASK option with the CALL statement that indicates concurrent execution of parallel tasks. TRANQUIL, an ALGOL-like language [4], has been designed to exploit parallelism in algorithms to be implemented in array processors such as the ILLIAC IV.

Sutherland [6] has proposed a graphical language (intended primarily for display) that exhibits parallelism in a computation. The computation is modeled as a directed graph in which a node represents a primitive function or a function of previously defined functions and primitives. The directed branches represent direction of control and data flow. The operational parallelism is indicated by multiple branches going to different functional nodes. The method is simple and direct, but the representation can be unwieldy in large computations.

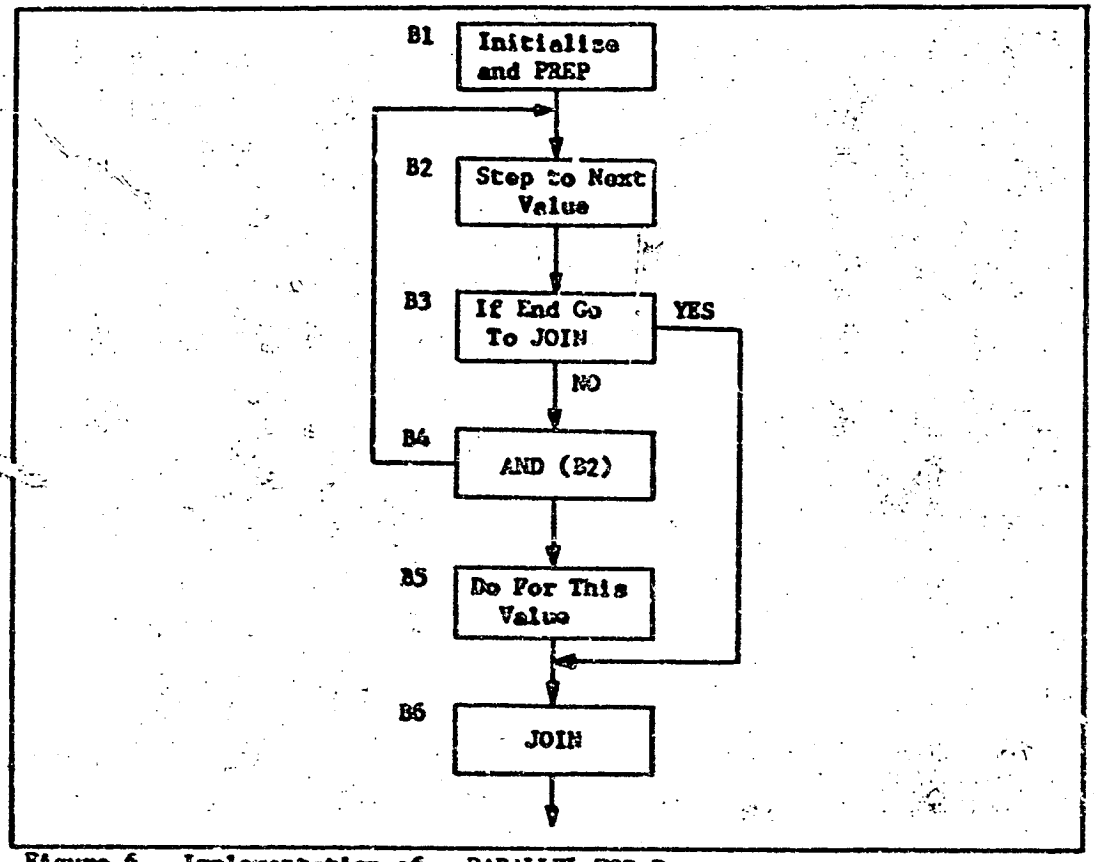


Figure 6. Implementation of a PARALLEL FOR Process.

Another approach to this problem is suggested in Single Assignment Languages [7]. Here all the statements are assignments. Only one statement is allowed to assign a value to a variable. Since the assignments indicate the functional relationships between the variables and the generated functions, the order of execution of the various quantities is also defined. The parallel operations are thus implicitly specified by this representation.

Since explicit indicators of parallelism are generally formulated at the time of coding, it is highly desirable that these indicators and related symbols be simple and natural. Some considerations associated with

these indicators are given here.

Because programs are usually represented by flow charts [5, 10, 11], any discussion on explicit indicators can be simplified by referring to flow charts. Necessary to such a representation are nodes and branches which define computational procedures. A task node represents a FORK operation. A decision node represents a computation whose outcome determines the next sequence of tasks to be undertaken. The JOIN node is the point of convergence of two or more parallel tasks; that is, control will move on to the next sequence of operations only after all the converging tasks are completed. The branches of a graph indicate the permissible transitions to the next set of computations as indicated by their orientation (direction).

In addition to these indicators, additional primitives should be included to protect the use of common data. The statement LOCK (X1, X2,...), for example, makes the data variables X1, X2,..., the exclusive property of the branch issuing the LOCK statement; the dual statement UNLOCK (X1, X2,...) releases the previously locked statements.

Some of the problems associated with the FORK and JOIN and similar type statements [12] include the fact that not all paths from a FORK statement have to converge to a single JOIN statement [9]. Thus the conventional FORK statement must be modified to permit nonconverging paths to release their processors after their completion. This can be accomplished by means of the IDLE statement which releases a processor and initiates no further action.

Another problem arises when two or more concurrent tasks seek access to the same critical resource. Simultaneous access can destroy useful data if it is not properly controlled. One way to prevent this is

to lock out common data from other processes and thus give its use to one process exclusively. If an interrupt occurs when LOCK is in effect, however, the common data may never permit its use even to high priority processes. The solution is either to inhibit the interrupt until an UNLOCK is issued by the process and/or warn the process of the occurrence of an interrupt and allow it some delay to unlock the data. Furthermore, a process can lock the data and go into an endless loop due to errors. An obvious solution would be to allow the system to limit the time during which a lock can be effective [13].

1.5 Implicit Parallelism

From the outset it must be stated that this approach to parallel processing is associated with sophisticated compiling and supervisory programs and the related overhead. Its attractiveness lies in that it is independent of the programmer, and existing programs would not have to be modified to take advantage of inherent parallelism. An algorithm written by Fisher [17] utilizes the input and output sets of each process within a program to determine essential ordering and thus inherent parallelism. Consider the flow diagram of the process shown in Figure 7. The algorithm requires as inputs

1. The number of processes to be analyzed. For this example each box in the flow diagram represents a process.

2. The input and output set for each process. The input set for process G, for example, consists of the variables v and w. The output set consists of the single variable z.

3. The given permissible ordering $_{o}S$ among the processes. This relation, illustrated in Figure 7(b), is simply a representation of the

possible paths which exist between the processes.

4. Any initially known essential order $_oT$ among the processes. This relation includes arcs connecting conditionals to their various conditional successors. The flow diagram of this example consists of only one conditional, process D. Thus process D and its possible successors E and F comprise the relation $_oT$ (Figure 7(c)).

Given this information the algorithm generates:

1. T, the essential serial ordering relation, and

2. S, the covering for the essential serial ordering relation. This relation represents T without those arcs that directly join processes which are joined indirectly by a sequence of two or more arcs in T.

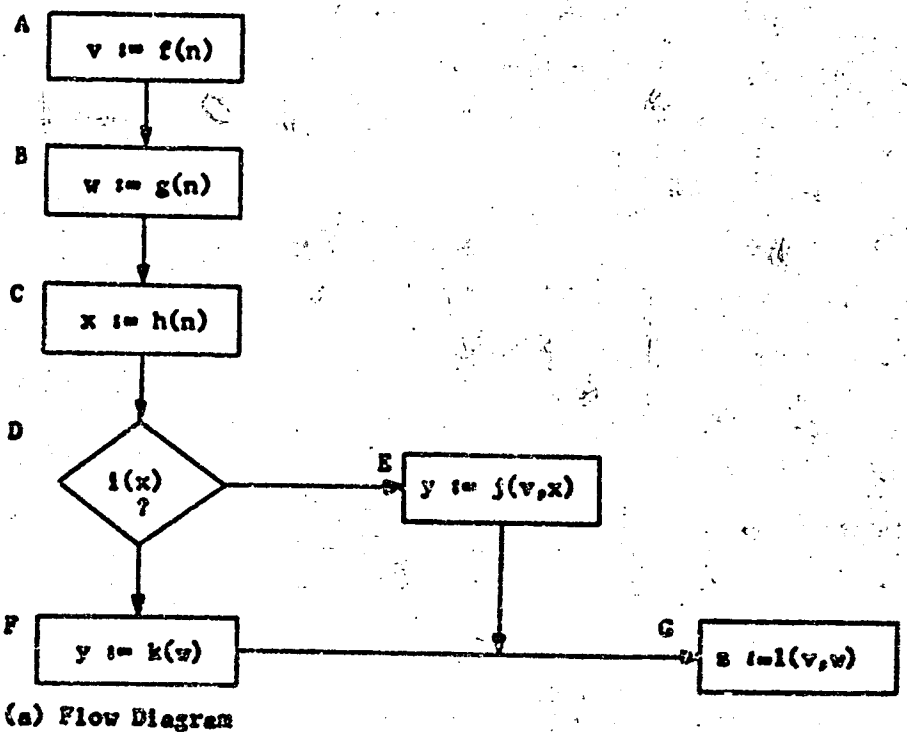
Once the essential serial ordering among the processes is known, parallelism inherent in the overall process can be determined.

Reference to Figure 7(d) shows that processes A, B, and C can be processed simultaneously. Similarly, processes D and G can be processed after the results of processes A, B, and C are available.

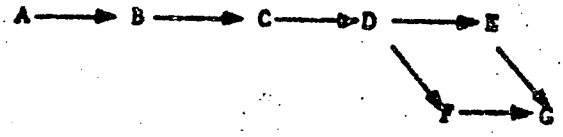
The implication from the relation S is that the overall process can be accomplished in three units of time as shown below.

UNIT OF TIME	PROCESSES COMPUTED
1	A, B, C
2	D, G
3	E or F

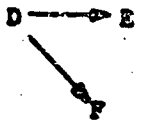
A possible shortcoming of the algorithm is that it fails to expressly provide any indication of flexibility in the ordering of processing among the processes. For example, the two variations which follow can also be accomplished in three units of time with no violation of the essential serial order.



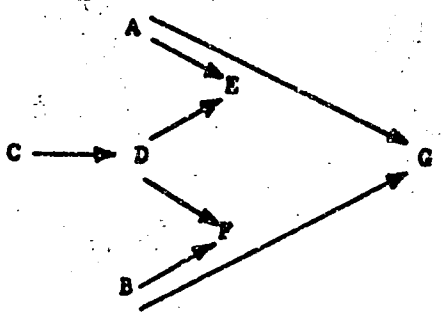
(a) Flow Diagram



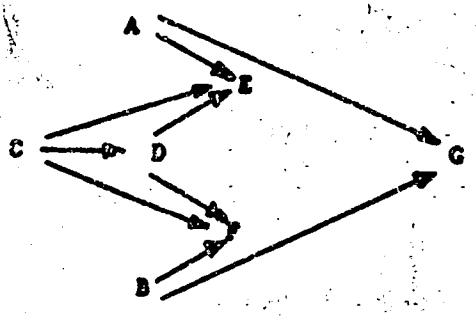
(b) $_{o}S$: Given Permissible Ordering Relation



(c) $_{o}T$: Known Essential Serial Ordering



(d) S: Cover for the Essential Serial Order



(e) T: Essential Serial Ordering Relation

Figure 7. Graphical Representation of a Program.

UNIT OF TIME	PROCESSES COMPUTED
1	A, C
2	B, D
3	G, E or F
1	B, C
2	A, D
3	G, E or F

Thus if only two processors were available for the three units of time, either of the two orderings shown above could get the job done. These alternatives are not apparent from the covering relation S.

Further discussion on implicit parallelism will be continued in Chapters II and III and will take two distinct forms. Chapter II will be confined to parallelism on a local level as exemplified by sub-tasks within arithmetic expressions. In Chapter III a technique will be introduced for detecting parallelism on a global level.

CHAPTER II

Algorithms for Detection of Sub-Task Parallelism

This chapter will be devoted to implicit recognition of sub-task parallelism within arithmetic expressions. The first half of the chapter will present a brief survey of existing algorithms which are designed to accomplish this purpose [23]. The second half will introduce a technique which approaches the problem in a somewhat different manner. The arithmetic expression which will be used as an example for each algorithm is given below.

A←B+C↑D*E*F+G↑H

Throughout this discussion, the usual precedence between operators will apply. In order of increasing precedence, the precedence between operators will be as follows: + and -, * and /, and ↑, where ↑ stands for exponentiation.

2.1 Kellerman's Algorithm [20]

This algorithm assumes that the input string is written in reverse Polish notation and contains only binary operators. The string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. These temporary results will be considered as operands during the next passes. Temporary results generated during a given pass are said to be at the same level and therefore can be executed in parallel. There will be as many passes as there are levels in the syntactic tree. The compilation of the expression listed above is shown in Figure 8.

Although this algorithm is simple and fast, it has two pronounced

shortcomings. The first is that it requires the input string to be in Polish notation; the second is its inability to handle operators which are not commutative.

i	INPUT STRING AFTER THE ith PASS	TEMPORARY RESULTS GENERATED DURING ith PASS
0	AB↑C↑DE*F*↑G↑H↑	
1	R1 C↑R2 F*↑G↑H↑	R1=A↑B R2=D*E
2	R3 R4↑G↑H↑	R3=R1↑C R4=R2*F
3	R5 G↑H↑	R5=R3↑R4
4	R6 H↑	R6=R5↑G
5	R7	R7=R6↑H

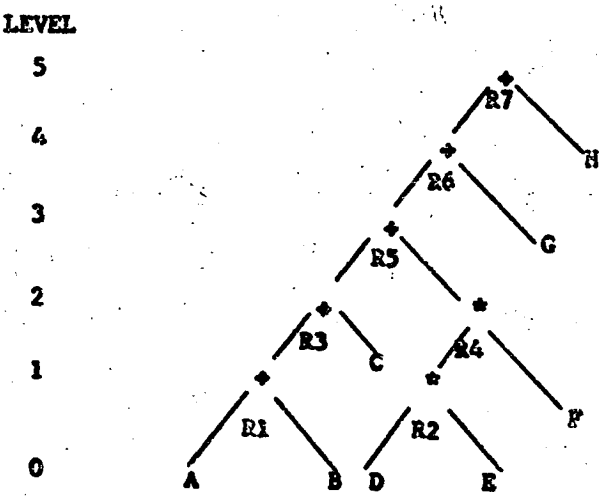


Figure 8. Parallel Computation of A↑B↑C↑D*E*F↑G↑H Using Hellerman's Algorithm.

2.2 Stone's Algorithm [21]

The basic function of this algorithm is to combine two subtrees of the same level into a level that is one higher. For example, A and B,

initially of level 0, are combined to form a subtree of level 1. The algorithm then searches for another subtree of level 1 by attempting to combine C and D. Since precedence relationships between operators prohibit this combination, the level of subtree (A↔B) is incremented by one. The algorithm now searches for a subtree of level 2 by attempting to combine C, D, and E. Since this combination is also prohibited, subtree (A↔B) is incremented to level 3. The next search is successful, and a subtree of level 3 is obtained by combining C, D, E, and F. These two subtrees are then combined to form a single subtree of level 4.

In a similar manner the subtree (G↔H), originally of level 1, is successively incremented until it achieves a level of 4; at that time it is combined with the other subtree of the same level to form a final tree of level 5.

The algorithm yields an output string in reverse Polish which does not expressly show which operations can be performed in parallel. Even though the output string is generated in one pass, the recursiveness of the algorithm causes it to be slow, and at least one additional pass would be required to specify parallel computations.

2.3 Squire's Algorithm [22]

The goal of this algorithm is to form quintuples of temporary results of the form:

Ri (operand 1, operator, operand 2, start level=max [end level op. 1; end level op. 2], end level=start level+1).

All temporary results which have the same start level can be computed in parallel. Initially, all variables have a start and end level equal to zero.

Scanning begins with the rightmost operator of the input string and proceeds from right to left until an operator is found whose priority is lower than that of the previously scanned operator. In the example the scan would yield the following substring:

D*E*F+G+H

Now a left to right scan proceeds until an operator is found whose priority is lower than that of the left-most operator of the substring. This yields: D*E*F. At this point a temporary result R1 is available of the form:

R1(D,*,E,0,1).

The temporary result, R1, replaces one of the operands and the other is deleted together with its left operator. The new substring is then:

R1*F+G+H.

The left to right scans are repeated until no further quintuple can be produced, and at that time, the right to left scan is re-initiated. The results of the process are shown in Figure 9.

Although the example shows the algorithm applied to an expression containing only binary operators, the algorithm can also handle subtraction and division with a corresponding increase in complexity.

A significant feature of this algorithm is that Polish notation plays no part in either the input string or the output quintuples. Because of the many scans and comparisons the algorithm requires, it becomes more complex as the length of the expression and the diversity of operators within the expression increase.

2.4 Baer and Bovet's Algorithm [23]

The algorithm uses multiple passes. To each pass corresponds a

INITIAL STRING: A+B+C+D*E*F+G+H

RIGHT TO LEFT SCAN

D*E*F+G+H

A+B+C+R2+G+H

LEFT TO RIGHT SCAN

R1*F+G+H
R2+G+H

R3+C+R2+G+H
R4+R3+R2+H
R4+R5+R2
R6+R2
R7

QUINTUPLES	Op.1	OPERATOR	Op.2	START	END
R1	D	*	E	0	1
R2	F	*	R1	1	2
R3	A	+	B	0	1
R4	C	+	G	0	1
R5	H	+	R3	1	2
R6	R4	+	R5	2	3
R7	R2	+	R6	3	4

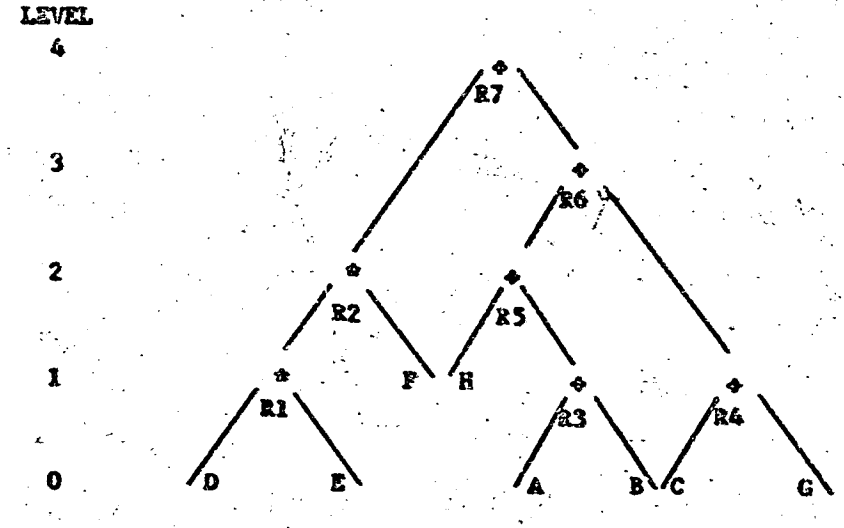


Figure 9. Parallel Computation of A+B+C+D*E*F+G+H Using Squire's Algorithm.

level. All temporary results which can be generated at that level are constructed and inserted appropriately in the output string produced by the corresponding pass. Then, this output string becomes the input string for the next level until the whole expression has been compiled. Thus the

number of passes will be equal to the number of levels in the syntactic tree. During a pass the scanning proceeds from left to right and each operator and operand is scanned only once.

The simple intermediate language which this algorithm produces is the most appropriate for multiprocessor compilation in that it shows directly all operations which can be performed in parallel, namely those having the same level number. The syntactic tree generated by this algorithm is shown in Figure 10.

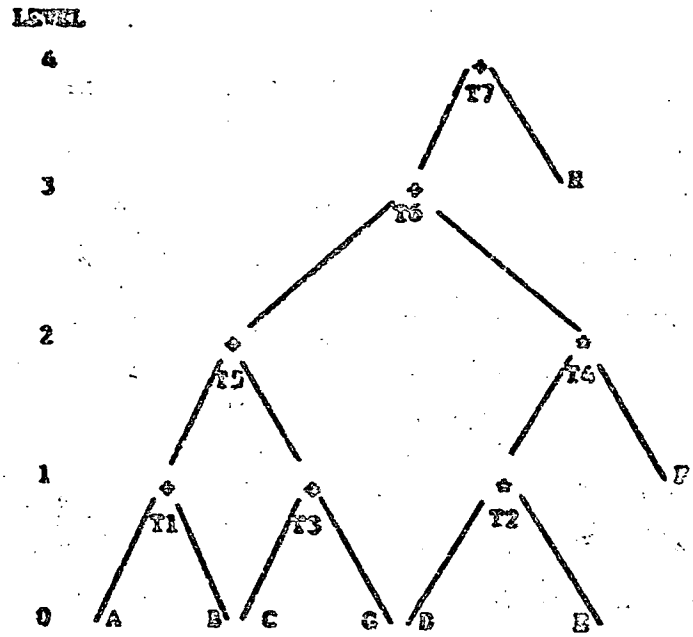


Figure 10. Parallel Computation of A*B+C*D-F*G*E Using Baer and Bovet's Algorithm.

2.5 Observations

Of the four algorithms mentioned here, Squire's and Baer's and Bovet's algorithms are the most powerful as well as the most complex, although the former algorithm requires more passes than the latter.

Although Hellerman's algorithm is the simplest and most rapid, it is handicapped by its requirement for a Polish input string and its limitation to binary operators. Stone's algorithm requires the fewest passes, but its recursiveness causes it to be slow. In addition, the reverse Polish string which it generates is not in a form suitable for multiprocessor utilization.

2.6 A New Algorithm for Detection of Sub-Task Parallelism

This section will introduce a technique whose goals are: 1) to produce a binary tree which illustrates the parallelism inherent in an arithmetic expression; and 2) to determine the number of registers needed to evaluate large arithmetic or Boolean expressions without intermediate transfers to main memory.

This technique is prompted by the fact that existing computing systems possess multiple arithmetic units which can contain a large number of active storages (registers). In addition, the superior memory bandwidths of the next generation of computers will simplify some of the requirements of this technique.

In the material presented below, a complex arithmetic expression is examined to determine its maximum computational parallelism. This is accomplished by repeated rearrangement of the given expression. During this process the given expression in reverse Polish form is also tested for "well formation", i.e., errors and oversights in the syntax, etc.

The arithmetic expression which was used as a model earlier will also be used here, namely A+B*C+D**E*F+G+H. The details of the algorithm follow.

1. The first step is to rewrite the expression in reverse Polish

form and to reverse its order.

$$\diamond\ H\ \diamond\ G\ \diamond\ *\ F\ *\ E\ D\ \diamond\ C\ \diamond\ B\ A$$

2. Starting with the rightmost symbol of the string, assign a weight to each member of the string based on the following procedure:

Assign to symbol S_i the value $V_i = (V_{i-1}) \diamond R_i$ $i = 1,2,\ldots,n$

where $R_i = 1 - \delta(S_i)$ given that

$\delta(S_i) = 0$ if S_i is a variable

$\delta(S_i) = 1$ if S_i is a unary operator

$\delta(S_i) = 2$ if S_i is a binary operator

and $V_{i-1} = V_{i-2} \diamond R_{i-1}$, $V_{i-2} = V_{i-3} \diamond R_{i-2}$, etc.,

such that $V_{i-(i-1)} = V_1 = R_1$, and $V_0 = 0$.

Using this procedure, the following expression results:

	Root Node								V_m					
i	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
S_i	$\diamond$	R	$\diamond$	G	$\diamond$	*	F	*	E	D	$\diamond$	C	$\diamond$	B	A
V_i	1	2	1	2	1	2	3	2	3	2	1	2	1	2	1

Note that for a "well-formed expression" of n symbols $V_n = 1$.

3. At this point the root node of the proposed binary tree can be determined. Thus the given string can be divided into two independent substrings. To determine the root node, draw a line to the left of the first symbol with a weight of 1 ($i=11$, $S_i=\diamond$, $V_i=1$) to the left of the symbol with the highest weight, V_m ($i=7$, $S_i=E$, $V_i=V_m=3$). The two independent substrings consist of the strings to the left and to the right of this line. The root node will be the leftmost member of the string to the left of the line ($i=15$, $S_i=\diamond$, $V_i=1$). Note that V_i also equals 3 for $i=9$; however, V_m is chosen from the earliest occurrance of a symbol with the highest weight.

4. The next step is to look for parallelism within each of the new substrings. Consider the rightmost substring. Form a new substring

consisting of the symbols within the values of $V_i=1$ to the right and to the left of V_m. Transpose this substring with the substring to the right of it whose leftmost member has a weight of $V_i=1$.

INITIAL RIGHTMOST SUBSTRING	S_i	÷	*	F	*	E	D	÷	C	÷	B	A
	V_i	1	2	3	2	3	2	1	2	1	2	1

FINAL RIGHTMOST SUBSTRING	i	11	10	9	8	7	6	5	4	3	2	1
	S_i	÷	÷	C	÷	B	A	*	F	*	E	D
	V_i	1	2	3	2	3	2	1	2	1	2	1

This procedure is repeated until the initial V_m occupies the position i=2 in the substring. For this example this is already the case. Thus the rightmost substring is in the proper form.

5. The transposition procedure of step 4 is applied next to the leftmost substring. However, since the leftmost substring of this example consists of only two operands and one operator, no further operations are necessary.

6. The resultant binary tree is shown in Figure 11. The numbers assigned to each node represent the final weight V_i of the symbol as determined in steps 1-5 above.

Some observations and comments on this algorithm are given below.

1. The two branches on either side of the root node can be executed in parallel. Within each main branch, the transposition procedure of step 4 yields supplementary root nodes. The sub-branches on each side of the supplementary nodes can be executed in parallel.

2. The number of levels in the binary tree can be predicted from the Polish form of the original string.

No. OF LEVELS = MAX [NUMBER OF 1's; V_m] in the substring (rightmost or leftmost) containing V_m.

3. The tree is traversed in a modified postorder form [26]. The

resulting expression is

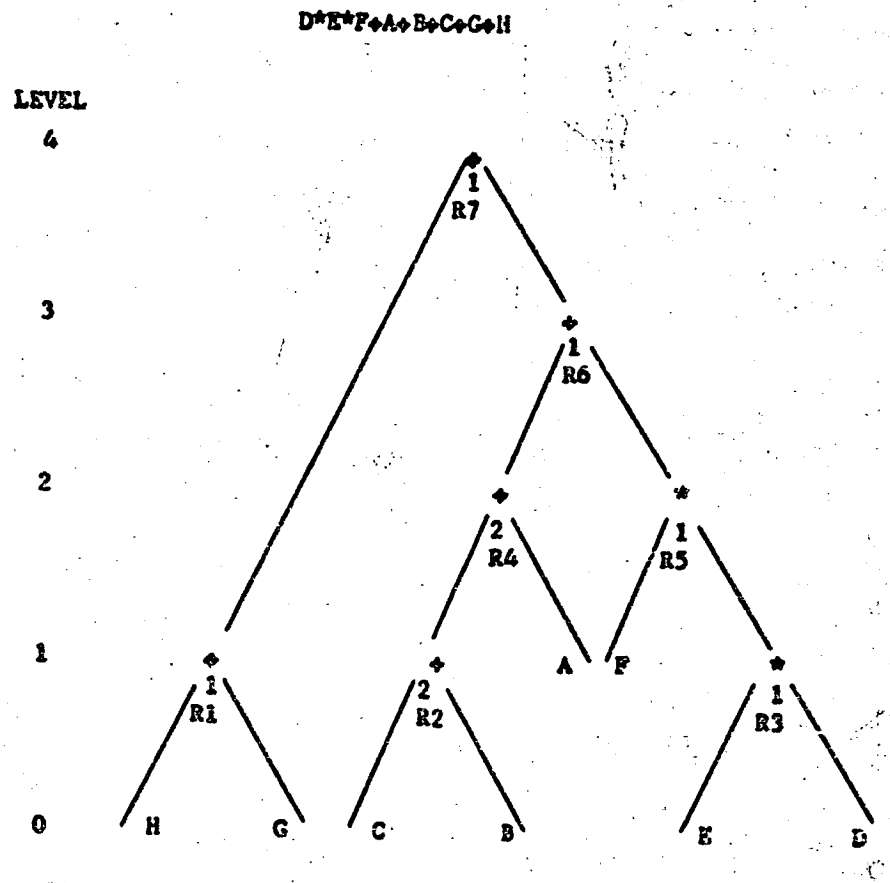


Figure 11. Binary Tree for Parallel Computation of A↓B↓C↓D*E*F↓G↓H.

4. An added feature of this technique is that the number of registers required to evaluate this expression without intermediate STORE and FETCH operations is obtained directly from the binary tree. This information is provided by the highest weight assigned to any node within the tree. Thus for this example the expression could be evaluated using at most two registers without resorting to intermediate stores and fetches.

5. This technique of recognizing parallelism on a local level has been applied to a single instruction, in particular, an arithmetic expression. It is worthwhile mentioning that each variable within the

expression can itself be the result of a processable task. Thus this technique can be extended to a higher level of parallel stream recognition, i.e., global parallelism.

6. No formal proof has been obtained for this algorithm. It has, however, been applied to many examples of varying complexity with satisfactory results.

CHAPTER III

Parallel Task Detection at A Global Level

3.1 Background Information

The problem presented here is that of compiling a conventionally written program for multi-processing. This involves the detection of the parallel processable segments and an indication of what tasks can be initiated in parallel and what tasks must be completed before the start of the next sequence of tasks. A graph theoretic model is presented which seems to provide enough generality that it is independent of the language, the application, the mode of compilation, and the number of processors in the system. Some graph terminology will be explained next to facilitate further discussion.

Computational processes are modeled by oriented graphs in which the vertices (nodes) represent single tasks and the oriented edges (directed branches) represent the permissible transition to the next task in sequence. Again, as before, a single task may represent one or more statements in a computational procedure and the associated data.

A directed graph which consists of n vertices and a set of oriented edges can be represented in a computer by its connectivity matrix C of dimension n x n such that its generic term C_{ij} is a "1" if and only if there is a directed edge from node i to node j and it is "0" otherwise [18]. The properties of the directed graph and hence of the computational process it represents can be studied by simple manipulations of the connectivity matrix.

A graph consisting of a set of vertices and edges is said to be strongly connected if and only if any node in it is reachable from any other. A subgraph of any graph is defined as consisting of a subset of vertices with all the edges between them retained. A maximal strongly connected (M.S.C.) subgraph is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. Given a connectivity matrix of a graph, all its M.S.C. subgraphs can be determined simply by well known methods [18]. A given program graph can be reduced by replacing each of its M.S.C. subgraphs by a single vertex and retaining the edges connected between these vertices and others. After the reduction, the reduced graphs will not contain any strongly connected components in them.

The paragraphs which follow will describe the sequence of operations needed to prepare for parallel processing in a multiprocessor computer a program written for a uniprocessor machine. It should be pointed out that some of the indicated operations can be performed simultaneously during compilation or assembly.

1. The first step is to derive during compilation the program graph which identifies the sequence in which the computational tasks are executed. The program graph is represented in the computer by its connectivity matrix. An example of such a graph is shown in Figure 12(a).

2. By an analysis of the connection matrix, the maximal strongly connected components are derived by simple operations [18]. This type of component is illustrated by tasks c and e in Figure 12(a). Each maximal strongly connected component (subgraph) is next considered as a single task, and the new graph, called the reduced or condensed graph, is constructed. This reduced graph is illustrated in Figure 12(b). The

condensed version of the program graph does not contain any strongly connected components, i.e., it has no loops.

3. From an input-output analysis of each task, it can be determined if some tasks can be performed in parallel. If two tasks exist in which the input parameters of one are not a function of the output parameters of the other, and vice versa, then by definition the two tasks can be executed in parallel. Thus from the input-output set analysis of the tasks, the previously obtained reduced graph is transformed into a parallel task graph which indicates parallel tasks. Whereas the reduced graph shows only permissible transitions between tasks, the parallel task graph gives consideration to the input-output relationships between tasks. In this latter graph two or more branches emanating from a single vertex represents a FORK operation, and the conjunction of two or more branches into a vertex is equivalent to a JOIN operation. Figure 13 illustretes the parallel task graph for the program graph of Figure 12.

4. From the connectivity matrix of the parallel task graph, the sequences in which the tasks must be performed are determined. The method used to determine which tasks can be performed in parallel uses the connectivity matrix of the final form of the program graph and the process of "precedence partitions" [19]. The method uses a series of eliminations (partitions) of columns (and corresponding rows) in the matrix which contain only zeros. Whenever more than one column can be removed from the matrix during a given elimination, the tasks represented by the node names of the eliminated columns can be executed in parallel. The series of eliminations used to determine the parallel processable tasks in the example are illustrated in Figure 14.

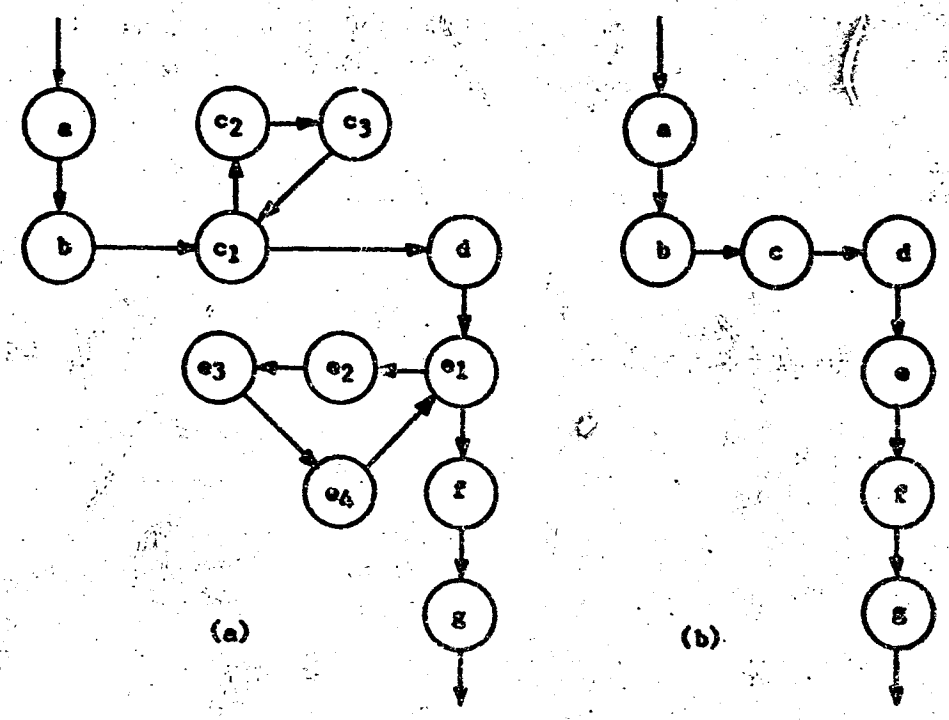


Figure 12. Graphical Representation of a Computational Process.

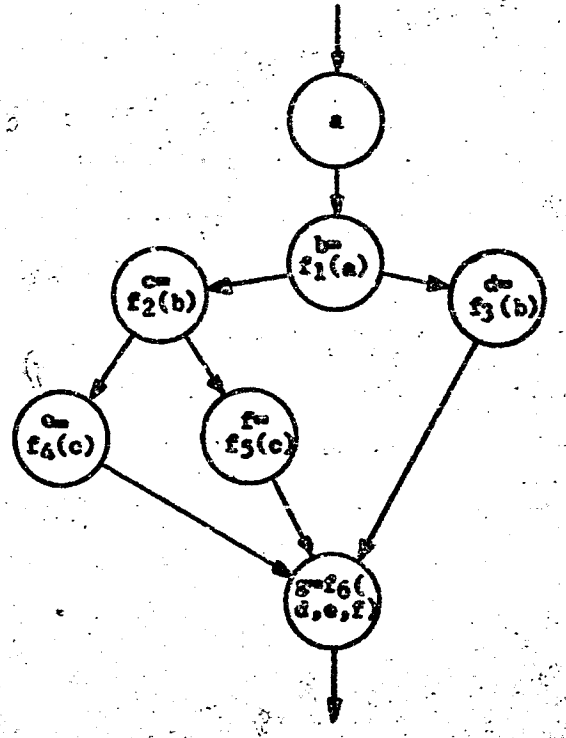


Figure 13. Representation of Parallel Processable Tasks of Figure 12.

Parallel Processable Tasks from Precedence Partitioning = [(c,d),(e,f)]

Reference to Figure 14 indicates that during the first step only node a is eliminated; the second step eliminates node b; during the third iteration nodes c and d are eliminated; the fourth iteration eliminates nodes e and f; the fifth iteration eliminates node g. Thus the precedence partitions are (a), (b), (c,d), (e,f), and (g). This indicates that task a must precede b, which in turn precedes the tasks (c,d) which can be done in parallel, etc.

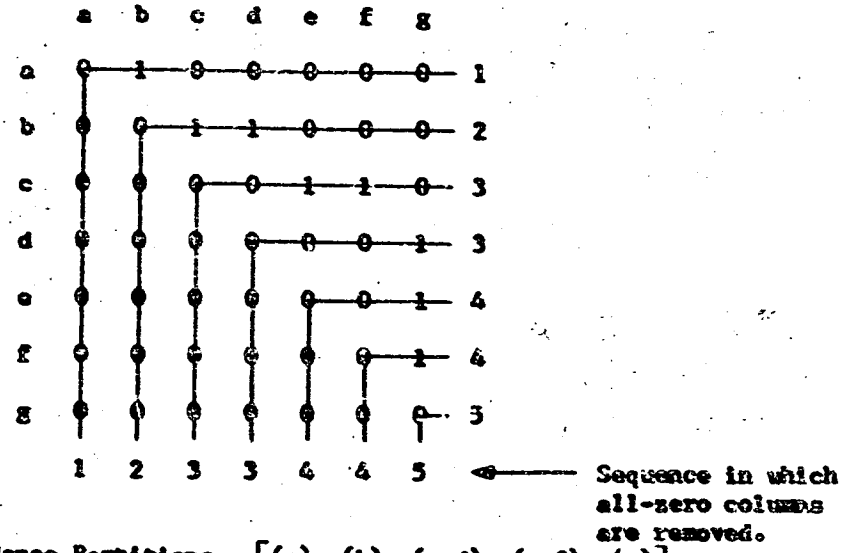


Figure 14. Precedence Partitions of the Program Graph of Figure 12.

3.2 A FORTRAN Parallel Task Recognizer

As mentioned in the introductory chapter, programs written in FORTRAN readily lend themselves to investigation of parallel processing techniques. In order to determine the degree of applicability of the method described in the previous section, it was decided to apply the method to a sample FORTRAN program. This was accomplished by writing a

program whose input consists of a FORTRAN source program; its output consists of a listing of the tasks within the source program which can be executed in parallel. The program written to accomplish this parallel task detection is known in its final form as a FORTRAN Parallel Task Recognizer.

The recognizer, also written in FORTRAN, does not investigate the actual structure of the source program language. Instead it relies on indicators generated by the way in which the program is actually written, as suggested by Berstein [3]. Consider, for example, the arithmetic expressions given below.

$$X1 = A+B$$
$$X2 = C+D$$

Because the right-hand side of the second expression does not contain a parameter generated by the computation which immediately precedes it, the two computations can be accomplished in parallel. If, on the other hand, the expressions were rewritten as shown below, the termination of the first computation would have to precede the initiation of the second.

$$X1 = A+B$$
$$X2 = X1+C$$

The key to the determination of the essential ordering, of course, is the presence of the equality sign in each of the expressions. In effect, the "right-hand side" of the second equality must be compared to the "left-hand side" of the first equality.

Although the implicit indicator here--the equality sign--is readily apparent, other (though perhaps less obvious) indicators exist in other statements. Consider the arithmetic IF statement:

IF(X-Y)3,4,5

Although no equality sign is present here, the right-hand and left-hand

suggestion is still present. The parameters within the parentheses in the statement above must be compared to the outputs of preceding statements in order to determine essential order. Thus X and Y in this statement are "right-hand" parameters. An identical argument holds for the PRINT statement:

PRINT 100, X,Y

The READ statement, on the other hand, consists entirely of "left-hand" parameters. This means that unless the READ statement is the successor of a conditional or the explicit successor of a previous statement, it can be executed concurrently with any statement that does not require its parameters with the exception of another READ statement.

A subroutine CALL statement may consist entirely of "right-hand" variables, or it may contain "right-hand" and "left-hand" variables. Consider the sequence of statements shown below.

```
X1 = A+B
X2 = C+D
CALL ALPHA (X1,X2,X3)
```

The implication here is that X3=f(X1,X2). Thus in a manner analogous to an arithmetic expression, X1 and X2 are "right-hand" variables and X3 is a "left-hand" variable.

The DO statement is unique in that it does not contain any parameters nor does it of itself generate any parameters. The repeated series of statements executed as a result of the DO statement form a "loop", however, and due to the graph considerations given earlier, all of the statements within a loop are considered as a single vertex in the reduced flow chart of a computational process. This condition is also made necessary because the rules of FORTRAN require that a DO loop be entered only through the DO statement. Statements within a DO loop, however, can

be examined for inherent parallelism in a manner similar to that applied to statements outside of a DO loop.

Because it was necessary to place an initial upper bound on the scope of this project, the recognizer in its present form possesses certain limitations. Thus only a subset of the entire FORTRAN subset is considered for recognition. As an example, the GO TO instruction is not part of the subset of instructions that are recognized. This is not to say that the recognizer does not permit transfer operations altogether, since the three-branch arithmatic IF statement is an acceptable form. IF statements, however, may themselves generate loops within the program. To eliminate these loops it would be necessary to analyse the connectivity matrix in the manner mentioned earlier before beginning the process of precedence partitions. The recognizer does not presently perform this analysis.

If it is determined, that two subroutines can be executed concurrently, no consideration is given to the fact that both may access memory locations established by COMMON or EQUIVALENCE statements.

The recognizer assumes that statements within the source program are written correctly according to the rules of FORTRAN. Thus the absence of a comma immediately after the format statement number in a PRINT statement, for example, would cause the recognizer to eliminate that statement from the recognition process.

Some limitations were placed on the recognizer merely to simplify programming and to make it more manageable in terms of memory requirements. Thus the input program can have no more than 99 executable statements, and no statement can have more than four "right-hand" or "left-hand" parameters. The recognizer cannot accept imbedded blanks within a statement, nor will it accept statements which require more than one standard punched-card

form. Source program statement numbers may be one to five digits in length, but the statement number must terminate in column five on the punched card representing the source program statement.

The assumptions and limitations listed above are not all inclusive. An attempt will be made to list as many of these limitations as possible in subsequent paragraphs as the recognizer is discussed in detail.

3.3 Theory of Operation

The recognition process consists of three phases. In Phase I, the program statements are accepted by the recognizer one at a time. The first consideration made after this READ operation is the determination of whether or not the statement is executable. This is done by comparing the statement type to the standard types accepted by the recognizer. A summary of the statement types accepted by the recognizer is given below.

Non-Executable	Executable
FORMAT	DO
REAL	READ
COMMON	Three-Branch Arithmetic IF
INTEGER	PRINT
DIMENSION	CONTINUE
	CALL
	END
	Arithmetic Expressions

In addition, the recognizer will accept comment cards and all-blank cards.

As each executable statement is read, it is analyzed for "right-hand" and "left-hand" variables as suggested earlier. As the variables within the statement are obtained, they are placed in tables containing variables only of the same kind, i.e., "right-hand" or "left-hand". An important distinction is made depending on whether or not the statement is within a DO loop. A "left-hand" variable within a DO loop, for example, is not placed in the same table as a "left-hand" variable outside of a DO

loop. Thus depending on its location within a program, a variable can be placed in one of four tables.

The location of a variable within a table is determined by the Executable Statement Number (ESN) assigned to the statement containing the variable at the time the statement is read. The tables mentioned here are dimensioned (50 x 7). All variables of the same type within a statement will occupy the same line but different columns of a particular table.

An example of what has been discussed here is given below. Assume that the statement shown is the fourth executable statement which has been read. Assume further that the statement is not located within a DO loop. The entries in the appropriate tables are shown below the statement.

X1 = A+B*(C-D)

LEFT-HAND TABLE OUTSIDE DO

2	4	1	X1			

RIGHT-HAND TABLE OUTSIDE DO

1	4	4	A	B	C	D

The first column in each table corresponds to the number of entries which have been made in that table. The second column contains the Executable Statement Number. The third column contains the number of variables contained in that entry. Columns 4-7 contain the actual variables; they correspond in number to the entry in column 3.

The DO statement generates an entry in a table which records the

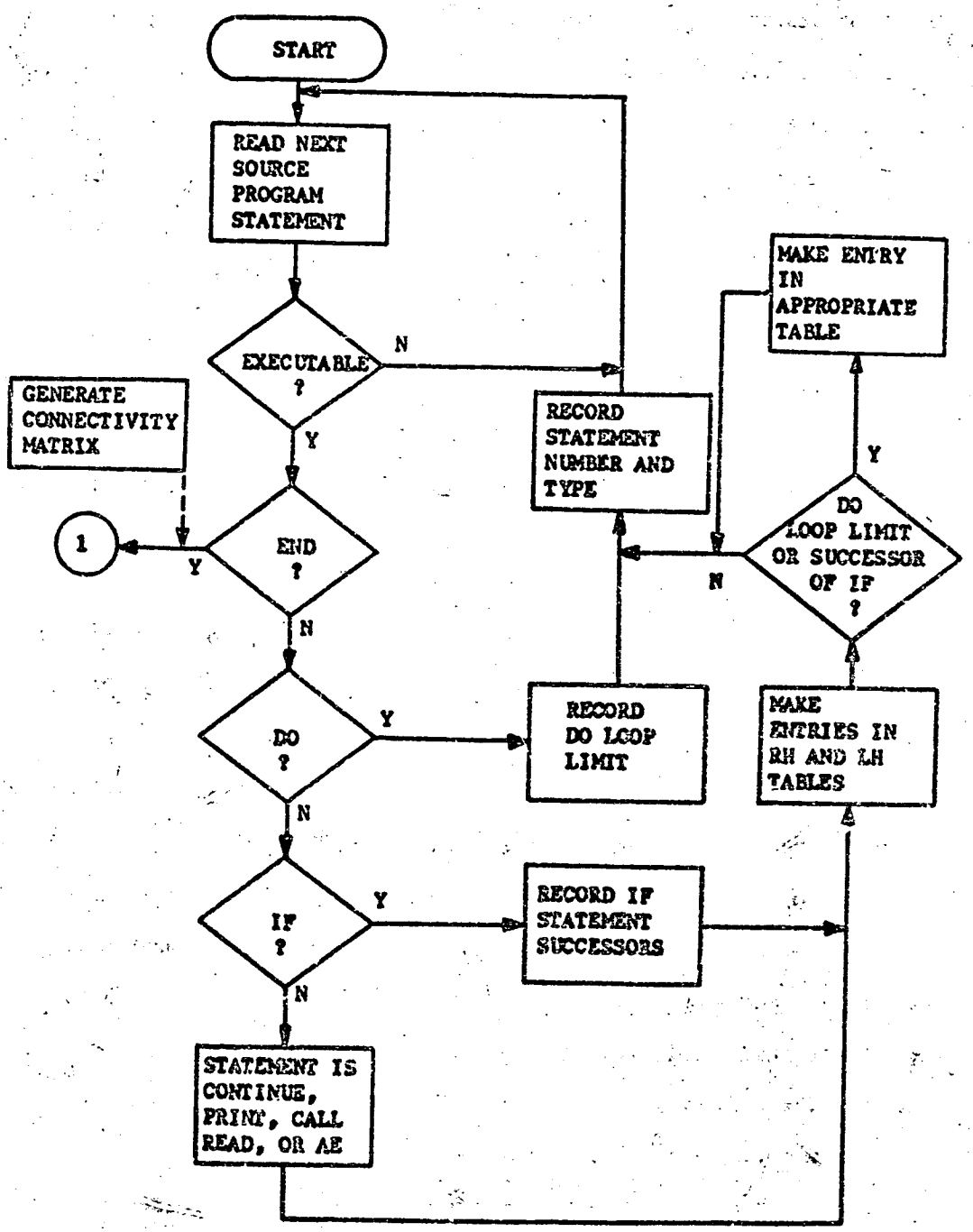


Figure 15. Phase I of the Recognition Process.

limit of the DO loop, i.e., the statement number. The IF statement also generates entries in a table which records the statement numbers of the three possible IF statement successors.

In addition, the address field of each executable statement is scanned to determine if it is a possible successor of an IF statement or the limit of a DO loop. When a match is found, the ESN of the statement under study is entered in the appropriate table.

Throughout this READ process, non-executable statements are ignored. The READ process is continued until an END statement is reached. At that point the recognizer is ready to begin Phase II of the recognition process--generation of the connectivity matrix. A flow chart of Phase I of the recognition process is shown in Figure 15.

The goal of the recognizer during the second phase is the generation of the connectivity matrix of the source program. This is accomplished by analyzing the variables in each of the four tables generated during Phase I. In general, variables in the "right-hand" tables are compared to entries in the "left-hand" tables. When a match is found, a "1" is entered at the appropriate point in the connectivity matrix.

The key to this matching process is the executable statement number. An entry in a "right-hand" table will not be compared to an entry in a "left-hand" table when the ESN of the latter exceeds the ESN of the former. The reasoning here is that a "right-hand" entry cannot affect the statements which succeed it in the sequential process.

(For the sake of convenience and brevity, the terms "right-hand" and "left-hand" will be written as RH and LH, respectively, throughout the remainder of this discussion.)

The unique structure of DO loops create an additional important

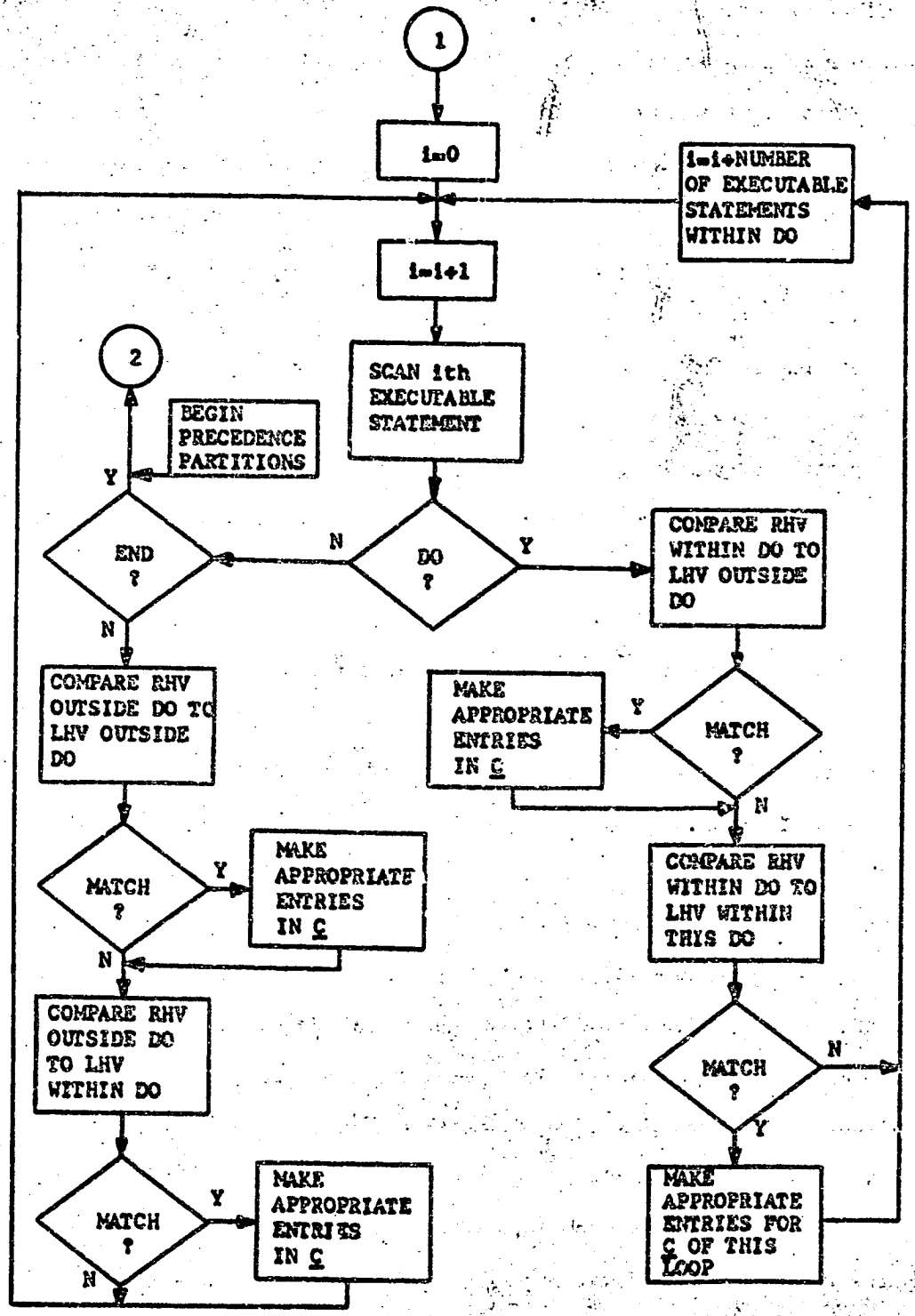


Figure 16. Phase II of the Recognition Process.

consideration during Phase II. Recall that a RH variable may exist within a DO loop or outside of a DO loop. For this reason it is necessary to compare RH variables to variables in both LH tables.

If a RH variable within a DO depends on the outcome of a statement which precedes the DO statement, the entry in the connectivity matrix is from the statement containing the LH variable to the DO statement and not to the statement containing the RH variable. Similarly, if a RH variable outside of a DO loop depends on the outcome of any statement within a DO block, the entry in C is from the DO statement to the statement containing the RH variable and not from the statement generating the required outcome. In effect, then, an entire DO loop is considered as either a predecessor or successor of a particular statement. This is in line with the graph policies presented earlier.

This consideration may produce a disparity, however, when the time comes to implement the parallel processable blocks generated by the recognizer. For example, if a source program contains twenty executable statements fifteen of which are contained within a DO loop, a high degree of potential parallelism is lost if the statements within the DO are not themselves considered for possible concurrent execution. For this reason, in addition to the overall connectivity matrix, the recognizer will generate a matrix for each DO block and output the components of the block which may be executed in parallel. In its present form, the recognizer will not accept more than five DO blocks nor will it recognize imbedded DO's, i.e., a DO within a DO.

A final consideration important to the matching process is created by the IF statement. If the successors of an IF statement located within a DO loop are also located within the loop, no complications arise. The

relationship between the IF statement and its successors will be reflected in the connectivity matrix for that DO loop. If, on the other hand, a possible successor of this IF statement is located outside of the DO, then their relationship will show up in the matrix for the overall program.

If it is determined that for a given program N tasks can be initiated simultaneously and one of these tasks is a successor of an IF statement, then the concurrent execution of all N tasks may not be permissible. Only those tasks within the block which would normally follow the successor of the IF statement can be executed in parallel. The flow chart for Phase II of the recognition process is shown in Figure 16.

Phase III of the recognition process is devoted to the generation of the parallel processable tasks within the source program. This is accomplished by performing precedence partitions on the connectivity matrices generated in Phase II. Precedence partitions were discussed in the first section of this chapter.

The output is in the form of a list of blocks. Each block contains the statement numbers of the statements within the program which can be executed concurrently.

Appendix B of this paper contains an example which illustrates in detail the method discussed here. Appendix A contains a complete FORTRAN listing of the recognizer program with accompanying flow charts.

3.4 Observations

This chapter has introduced a method for recognizing parallelism on a global level. The method has been implemented in the form of a FORTRAN recognizer. On a limited basis, at least, it has been shown that the method of precedence partitions can be successfully employed to

determine parallel processable tasks within a computational process. The recognizer itself possesses both major and minor limitations which must be eliminated before the recognition techniques used here can be applied to programs more complex in nature than the example program.

Some of the limitations of the recognizer can be eliminated quite easily. Recognition of a more complete set of FORTRAN instructions, for example, can be accomplished during Phase I by simple expansion of the recognition process. The generation of the connectivity matrix during Phase II, however, would be somewhat more complicated because of the many types of loops and branches which can be created by several instructions in the FORTRAN set.

In order to accommodate a source program which has more than 99 executable statements, it would be necessary to expand the size of the appropriate tables. Similar action would also be necessary to accommodate executable statements with more than four "right-hand" or "left-hand" parameters.

Because FORTRAN subroutines are compiled separately, it may be desirable to write portions of the main program in the form of subroutines if memory limitations were to become a problem. In its present form, the main program is the largest component of the recognizer program and requires 51556_8 words in memory.

As was stated earlier, the method of precedence partitions is independent of the source program language. The recognition process, however, is based on the language used by the source program. Thus the recognizer cannot accept a source program written in a language other than FORTRAN. Furthermore, it is not possible to implement a recognition process which is independent of the source program language.

It is quite likely, however, that FORTRAN is not the best language to use in implementing a recognizer program. Further work in this area may determine that a list processing language, for example, would be considerably more efficient for work of this nature.

More important, however, is a need to determine the benefits accrued, if any, by applying these techniques to the fundamental problem--the reduction of total processing time. Before this type of information can be obtained, it will be necessary to make exhaustive measurements in a manner similar to that proposed by Russell and Estrin [25]. It is not enough to merely determine that two tasks can be executed concurrently; it must be determined whether or not this concurrent execution will actually result in a more efficient utilization of system resources.

The concept of parallel processing can be easily defended on an intuitive basis. Proof of its actual worth, however, is much harder to obtain. That parallel processing will play a vital role in the future of information processing cannot be denied. Much work remains to be done before it can live up to its promise.

A P P E N D I X A

The Recognizer Program--Flow Charts And Listing

This Appendix includes a complete listing of the recognizer program and accompanying flow charts. The listing begins on page 73.

Definitions of some of the important terms and variable names used in the flow charts and the recognizer program are given below.

NEST - Table which records the executable statement type.
NTABESN - Table which stores executable statement number and type.
NES - Table which stores the executable statements and their assigned number.
NLRODO - Table in which LH variables outside of a DO loop are stored.
NLRWDO - Table in which LH variables within a DO loop are stored.
NRRODO - Table in which RH variables outside of a DO loop are stored.
NRRWDO - Table in which RH variables within a DO loop are stored.
NTABDO - Table which records the address and statement number of a DO loop limit.
NTABIF - Table which records the address and statement number of IF statement successors.
NC - The connectivity matrix for the overall source program.
NC_i - Connectivity matrices for each DO loop within the source program.
NIS - Temporary store for each source program statement.
EX - Table which contains acceptable executable statement types.
NONEX - Table which contains acceptable non-executable statement types.
TEST, MX, MZ, MZX - Tables which contain constants for input word manipulation.
KESN - Executable statement number.
NDO - Determines if variables are to be stored in tables for variables within a DO loop or outside of a DO loop.
NLRW - Index for NLRWDO.
NLRO - Index for NLRODO.
NRRW - Index for NRRWDO.
NRRO - Index for NRRODO.
NTDO - Index for NTABDO.
NTIF - Index for NTABIF.
NIFO, NIFW - Tables for storing IF statement successors.
NPP - Table which contains the parallel processable segments of the source program.

Recognizer Subroutines

Listed here are all the subroutines used by the recognizer program. A brief statement on the purpose and peculiarities of the subroutine accompanies each subroutine listed.

Subroutine MDO

This subroutine is accessed every time a DO statement within the source program is read by the recognizer. The primary purpose of this subroutine is to record the limit of the DO loop as written in the DO statement. Scanning begins with column 10 and stops when a blank column is encountered. A character other than a blank after the loop limit will cause an error, and a diagnostic will be issued.

Subroutine MREAD

The purpose of this subroutine is to record variables as input by the READ statement. The subroutine begins a scan of the input statement with column 11, but no variable is expected until the comma after the input format statement number has been recognized. Scanning is completed after the entire card containing the READ statement has been read.

Subroutine MVAR

This subroutine can only be called from one of the other subroutines. Its function is to take a word transmitted by the calling subroutine and right-justify the variables found therein and return the variables to the calling subroutine. MVAR has the capability of accepting variables which begin in one word and terminate in an adjacent word. The process of right-justifying the variables within a word is based on instructions transmitted by the calling subroutine. This information includes the

number of charactors in the variable name and the location in the transmitted word at which the variable terminates.

Subroutine MIF

The purpose of this subroutine is to record the variables used in IF statements and to record the successors of IF statements. Although scanning begins with colum 9, the presence of a variable is not expected until the leading left parenthesis has been detected. The search for variables ceases when the rightmost right parenthesis is found. At that point the search begins for the successors of the IF statement. A character other than a blank between the rightmost right parenthesis and the first digit of the first successor will cause the scan to stop: a diagnostic will be issued. The scanning process will normally cease when a blank is next encountered. Thus there should be no imbedded blanks within the three successors.

Subroutine MPRINT

The purpose of this subroutine is to record the variable names issued in the PRINT statement. The search for the variables does not begin until after the comma after the output format statement number has been encountered. The search continues until a blank after the variable list is found.

Subroutine MCALL

This subroutine records the names of the parameters issued in the CALL statement. In addition to recognising the variable names, the subroutine must also determine whether they are right-hand or left-hand variables. This is accomplished by comparing each variable name with the

variable names found in both left-hand tables. If a match is found, the variable is labelled as a right hand variable. If no match is found, it is assumed that the variable was generated within the called subroutine. Consequently, it is labelled as a left-hand variable.

The search continues until after the blank after the rightmost right parenthesis is found.

Subroutine MAE

This subroutine is called whenever an arithmetic statement is found in the source program. MAE detects the variables on both sides of the equality sign and labels them accordingly. The subroutine determines that the end of a variable name has been reached when it encounters a special character. This character will also initiate the start of a search for a new variable. The search continues until a blank is found.

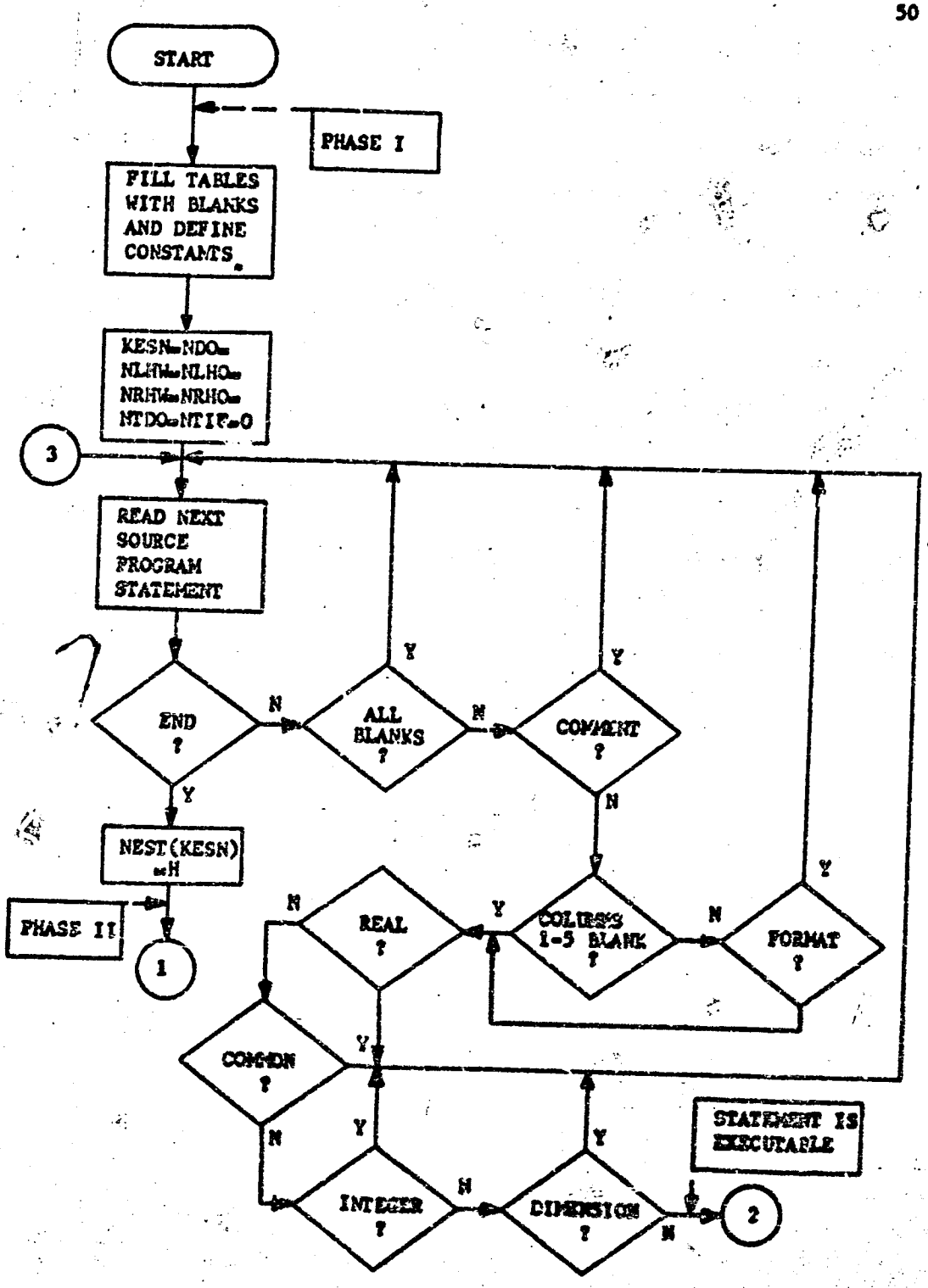


Figure 17. Flow Diagram of the Main Program.

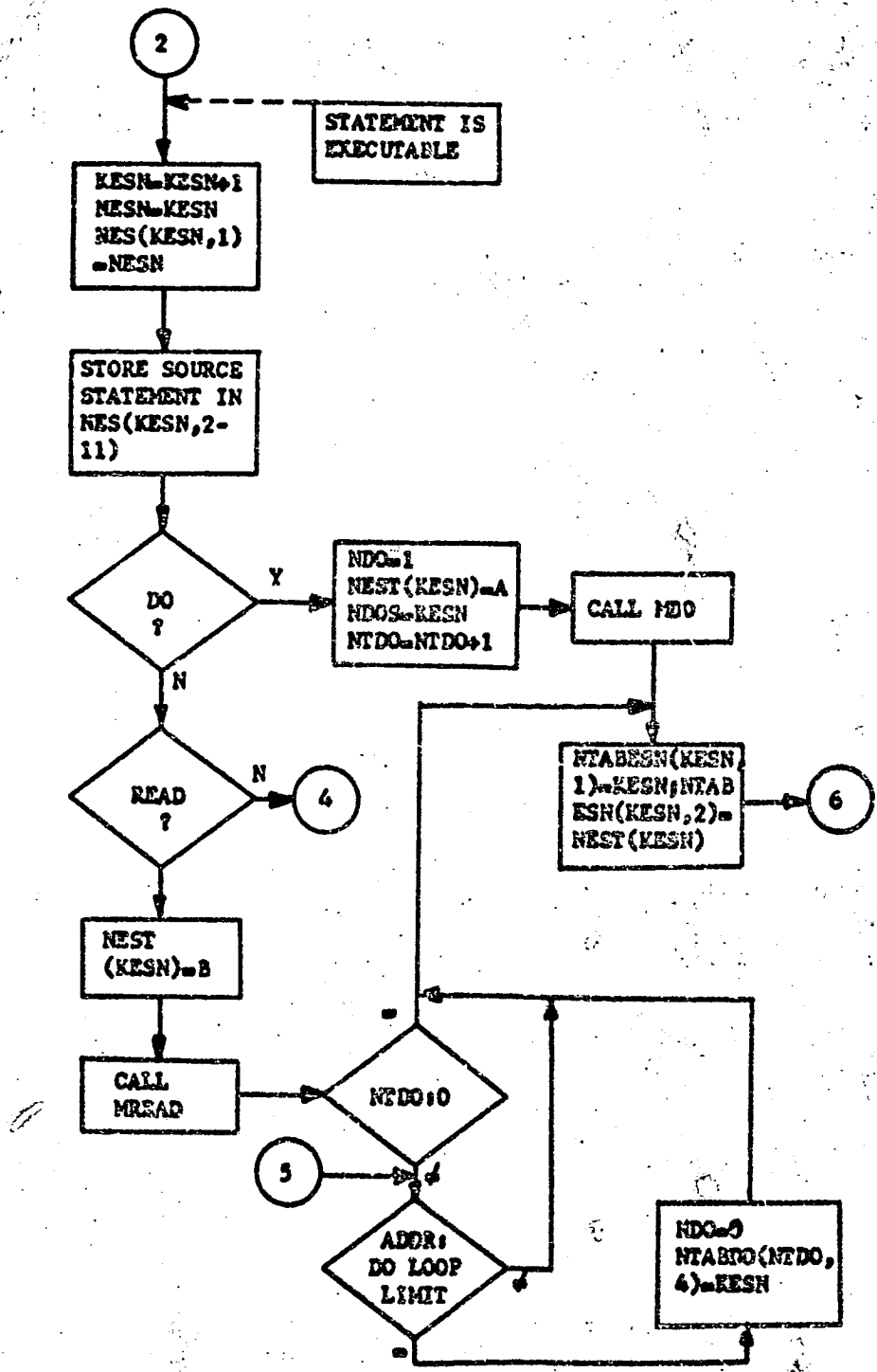


Figure 17. Flow Diagram of the Main Program. (Continued)

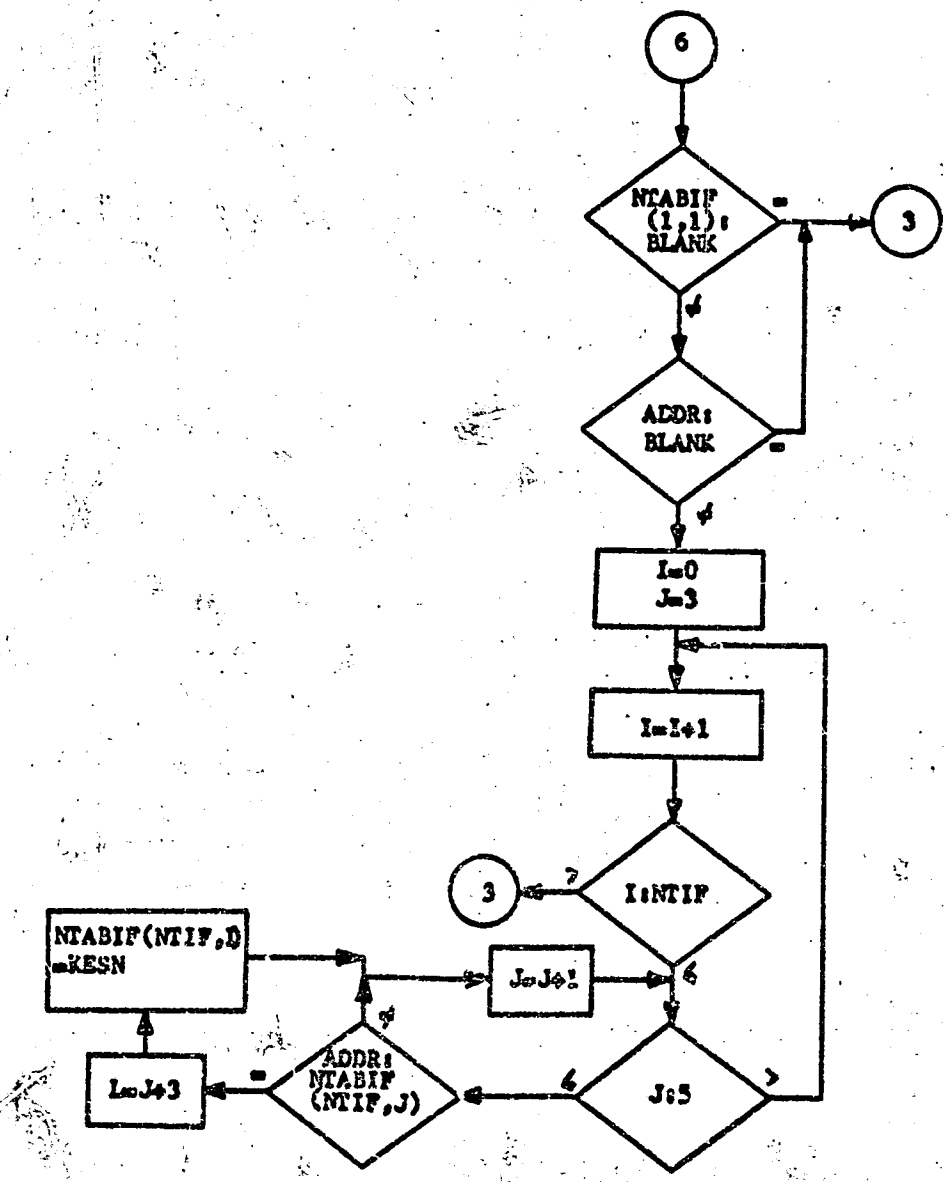


Figure 17. Flow Diagram of the Main Program. (Continued)

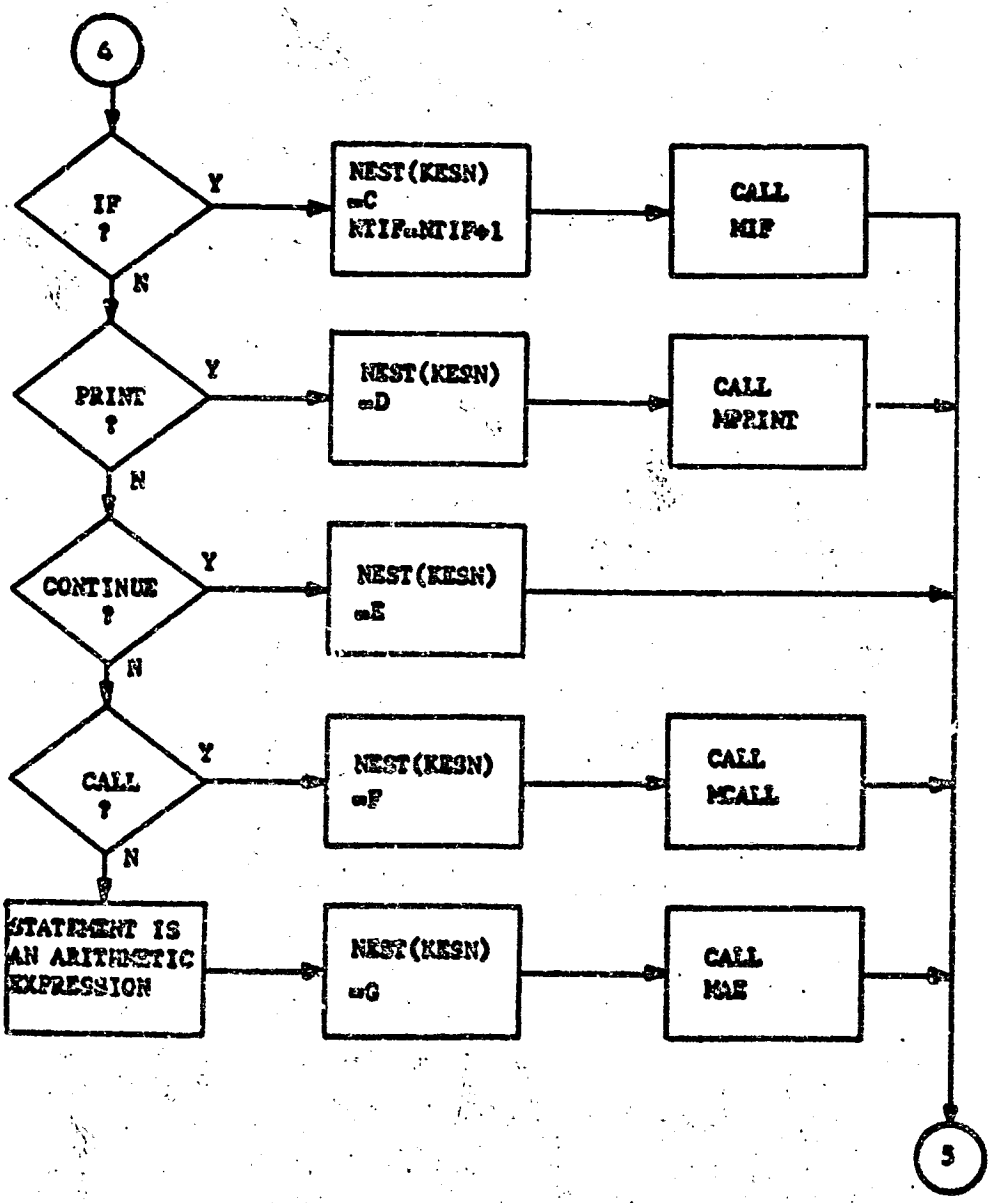


Figure 17. Flow Diagram of the Main Program. (Continued)

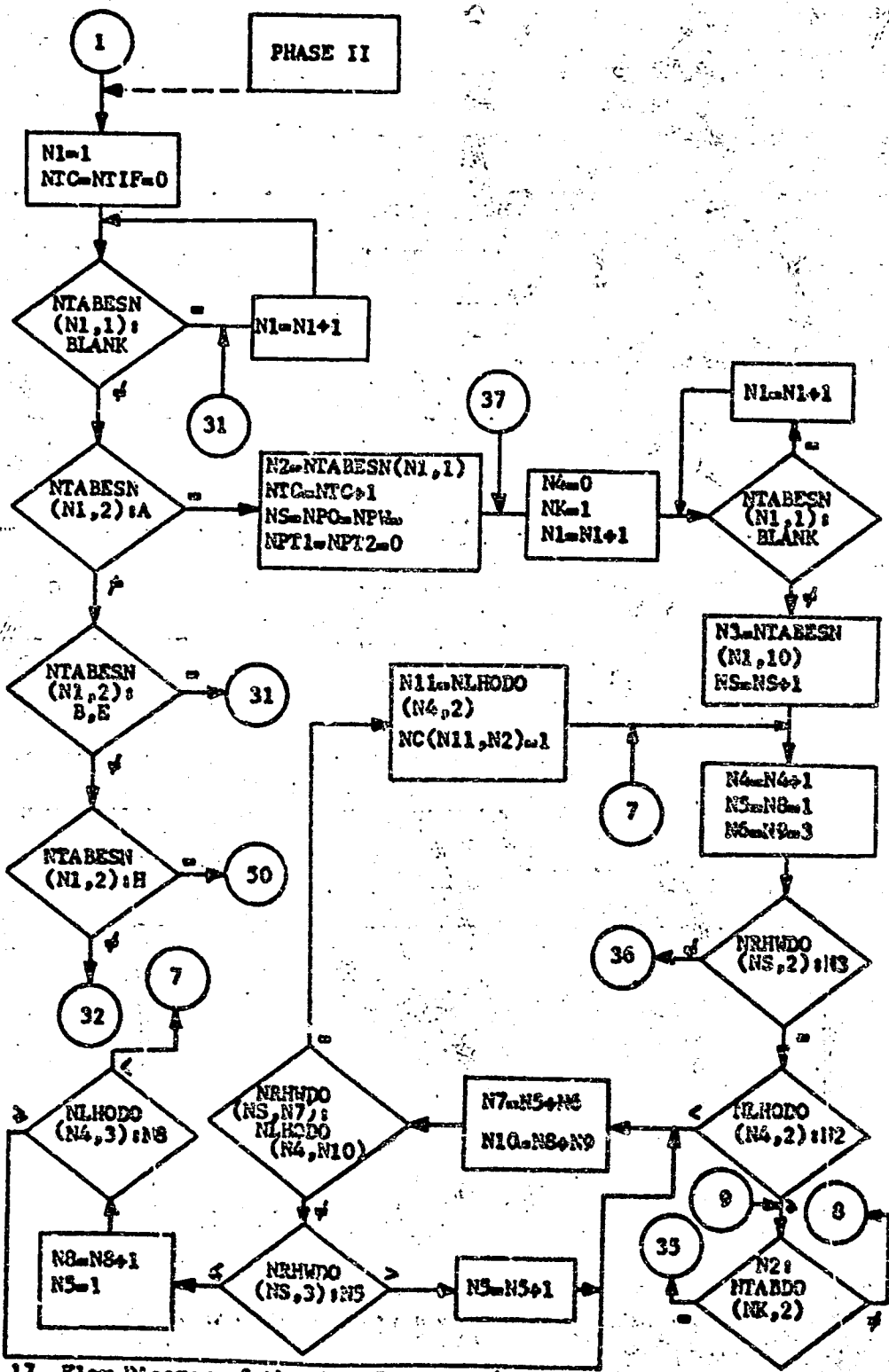


Figure 17. Flow Diagram of the Main Program. (Continued)

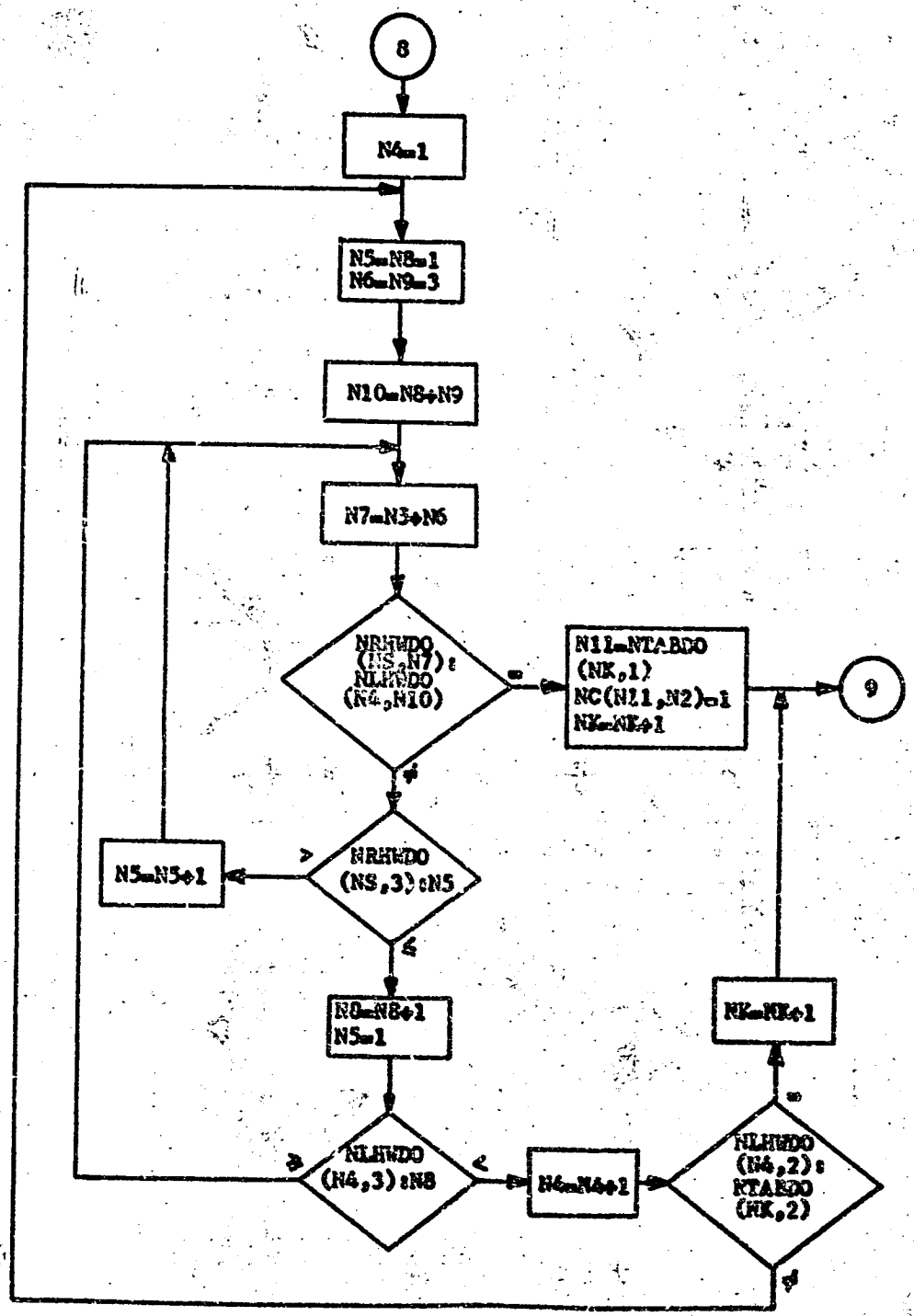


Figure 17. Flow Diagram of the Main Program. (Continued)

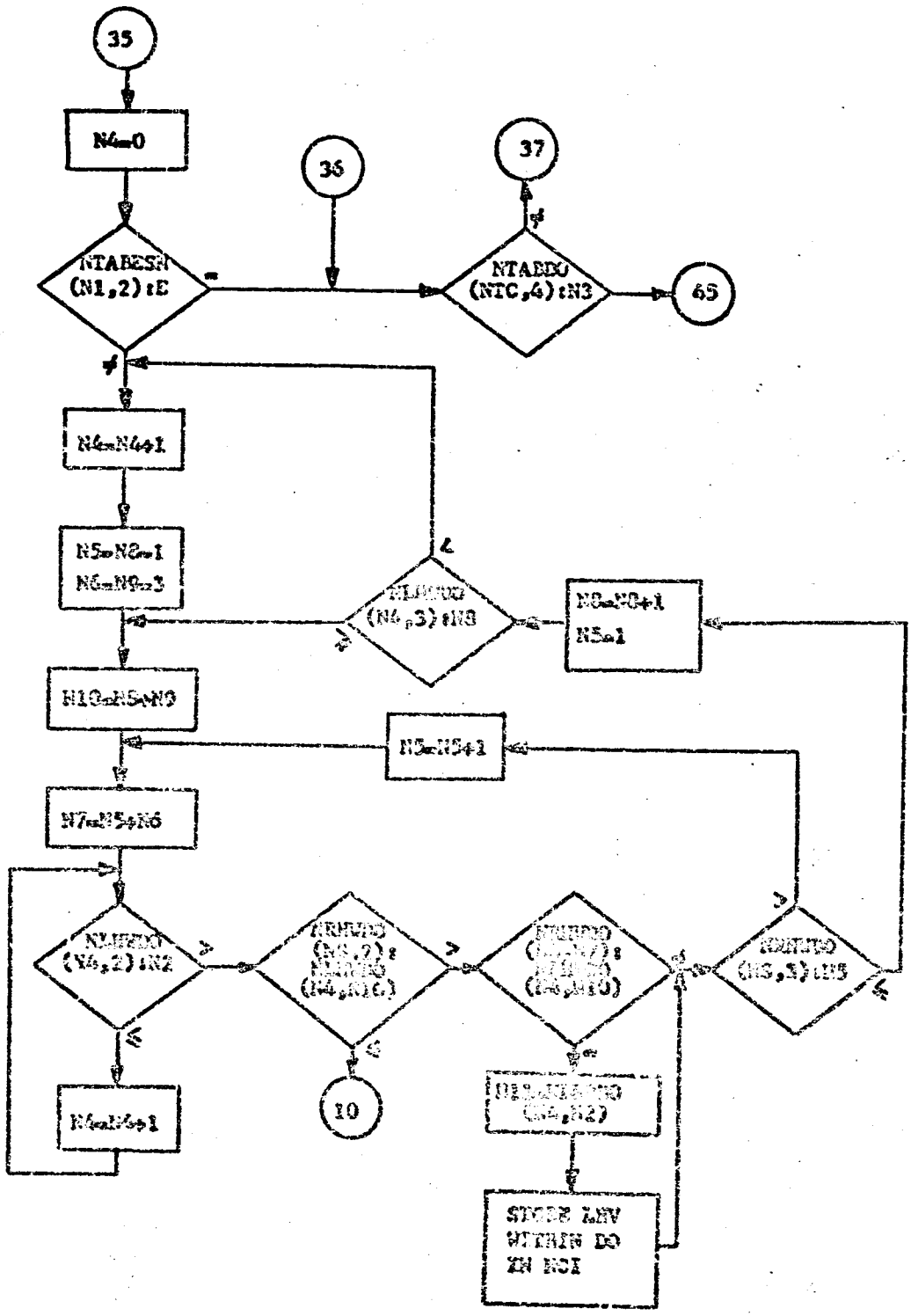


Figure 17. Flow Diagram of the Main Program. (Continued)

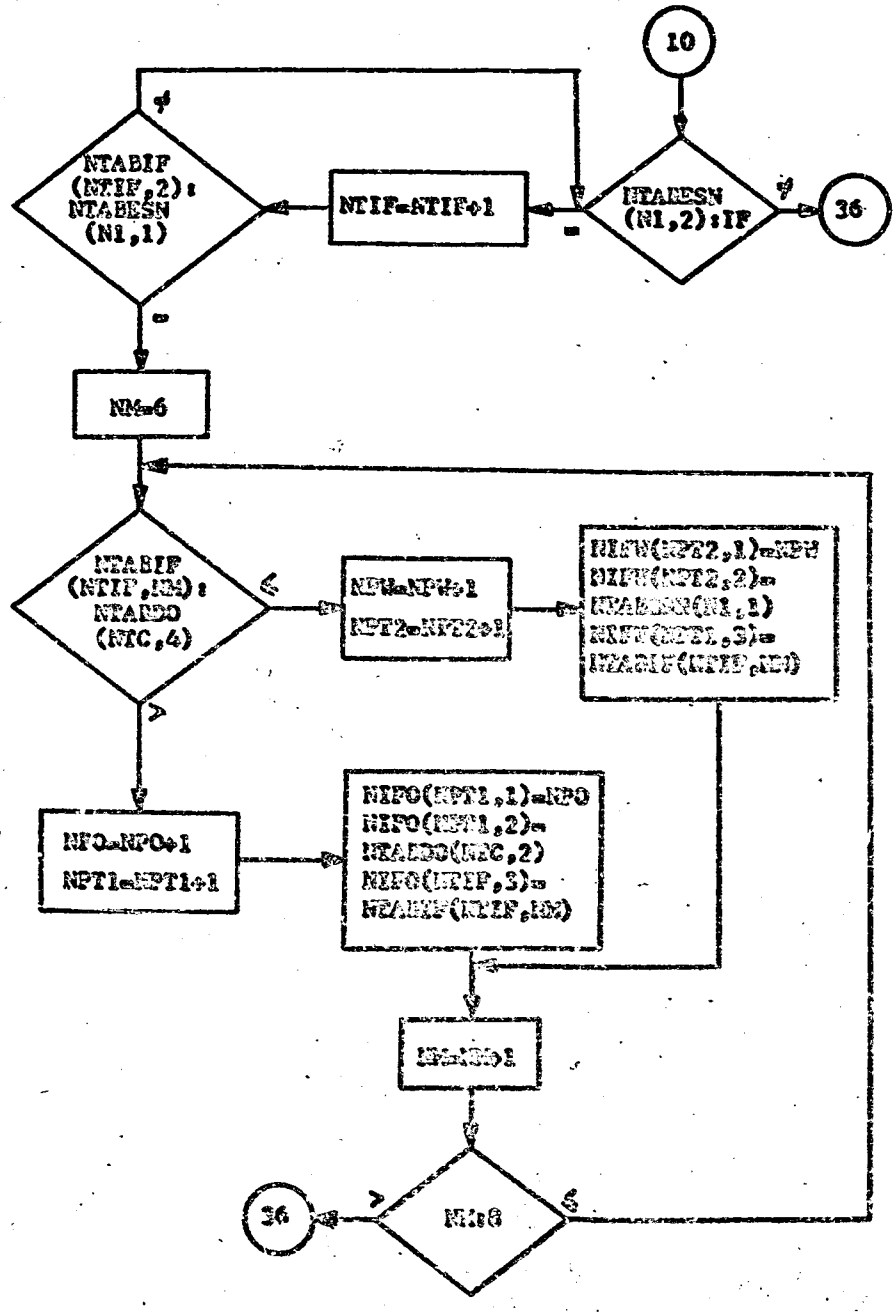


Figure 17. Flow Diagram of the Main Program. (Continued)

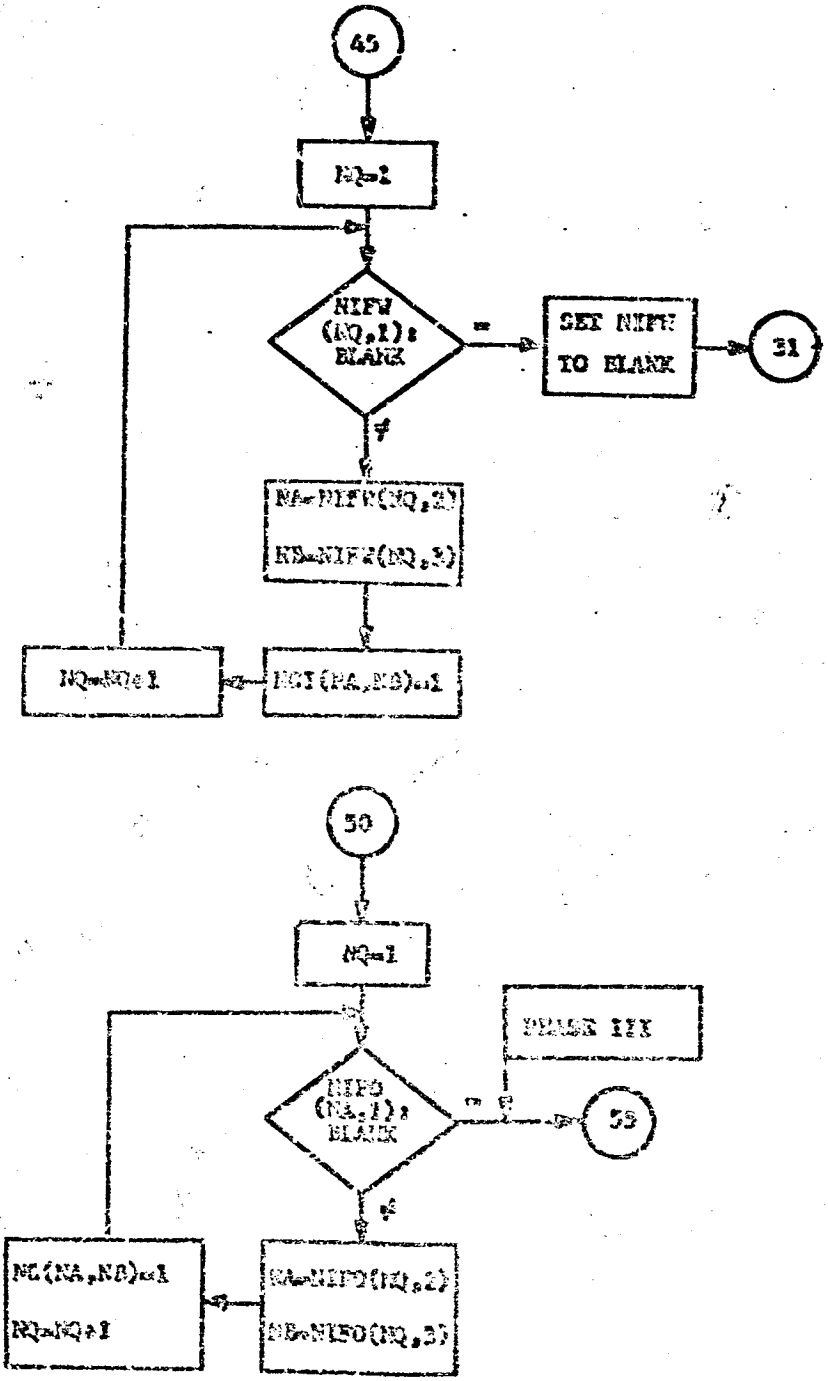


Figure 17. Flow Diagram of the Main Program. (Continued)

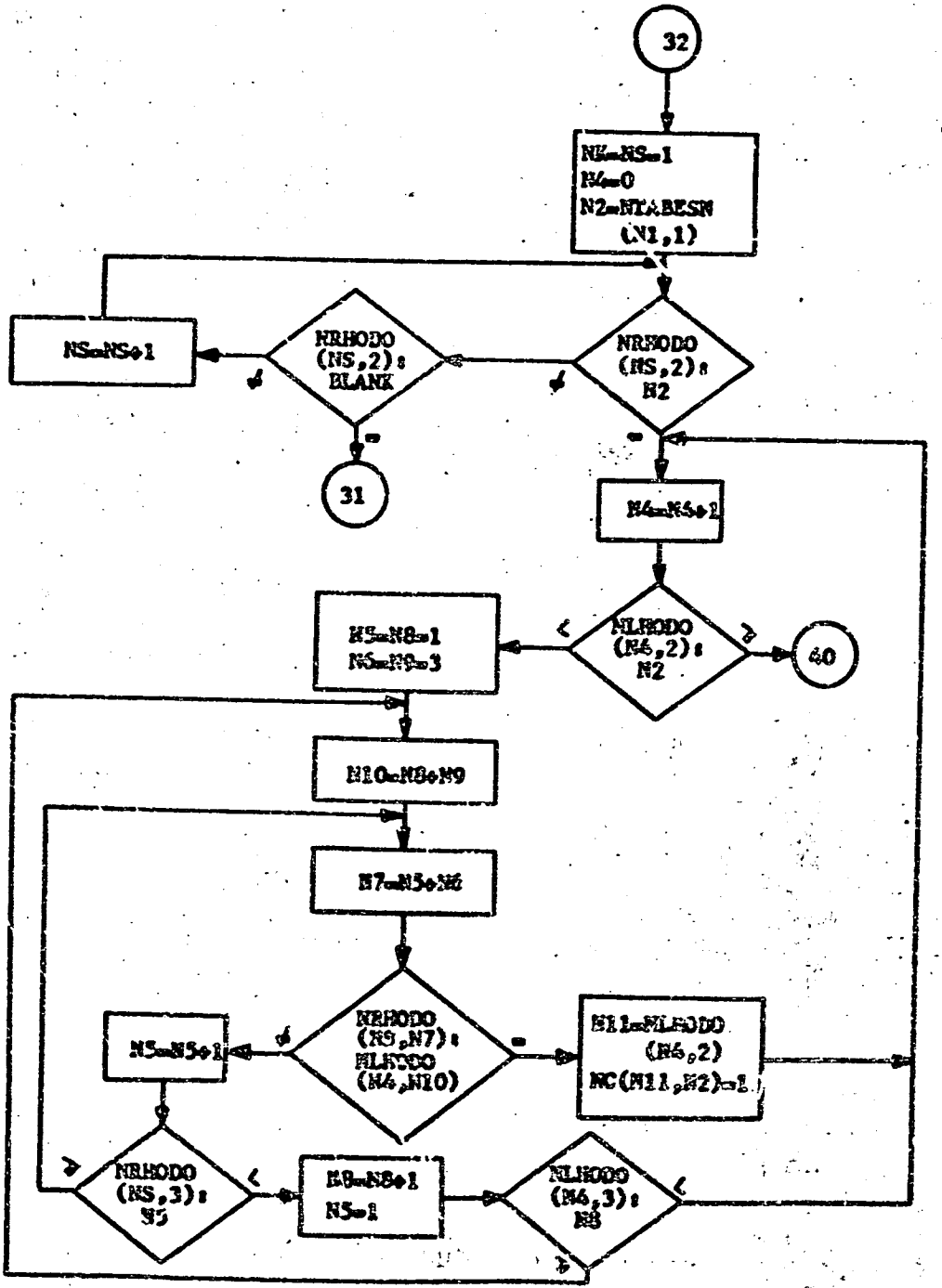


Figure 17. Flow Diagram of the Main Program (Continued)

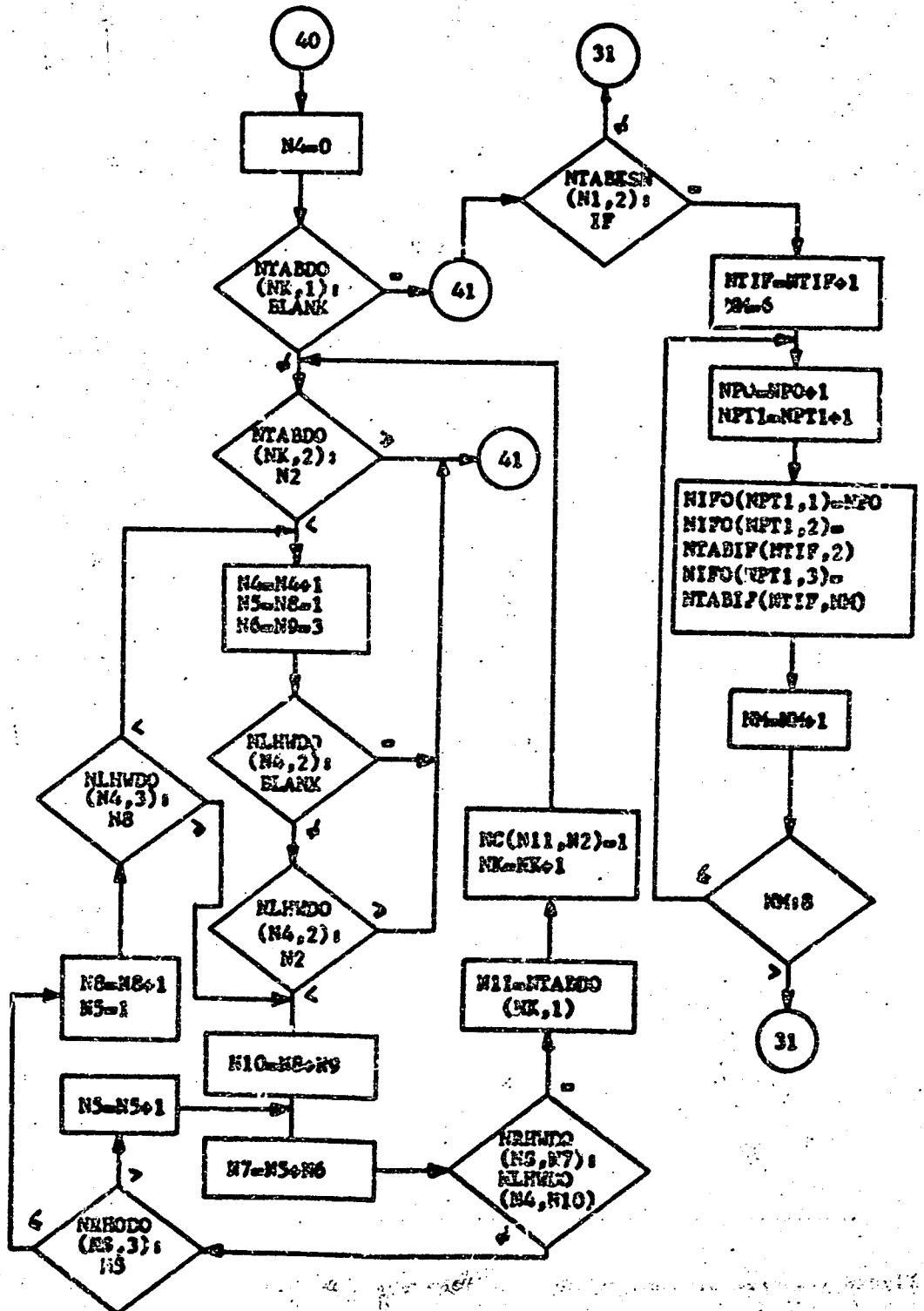


Figure 2%. Flow Diagram of the Main Program. (Continued)

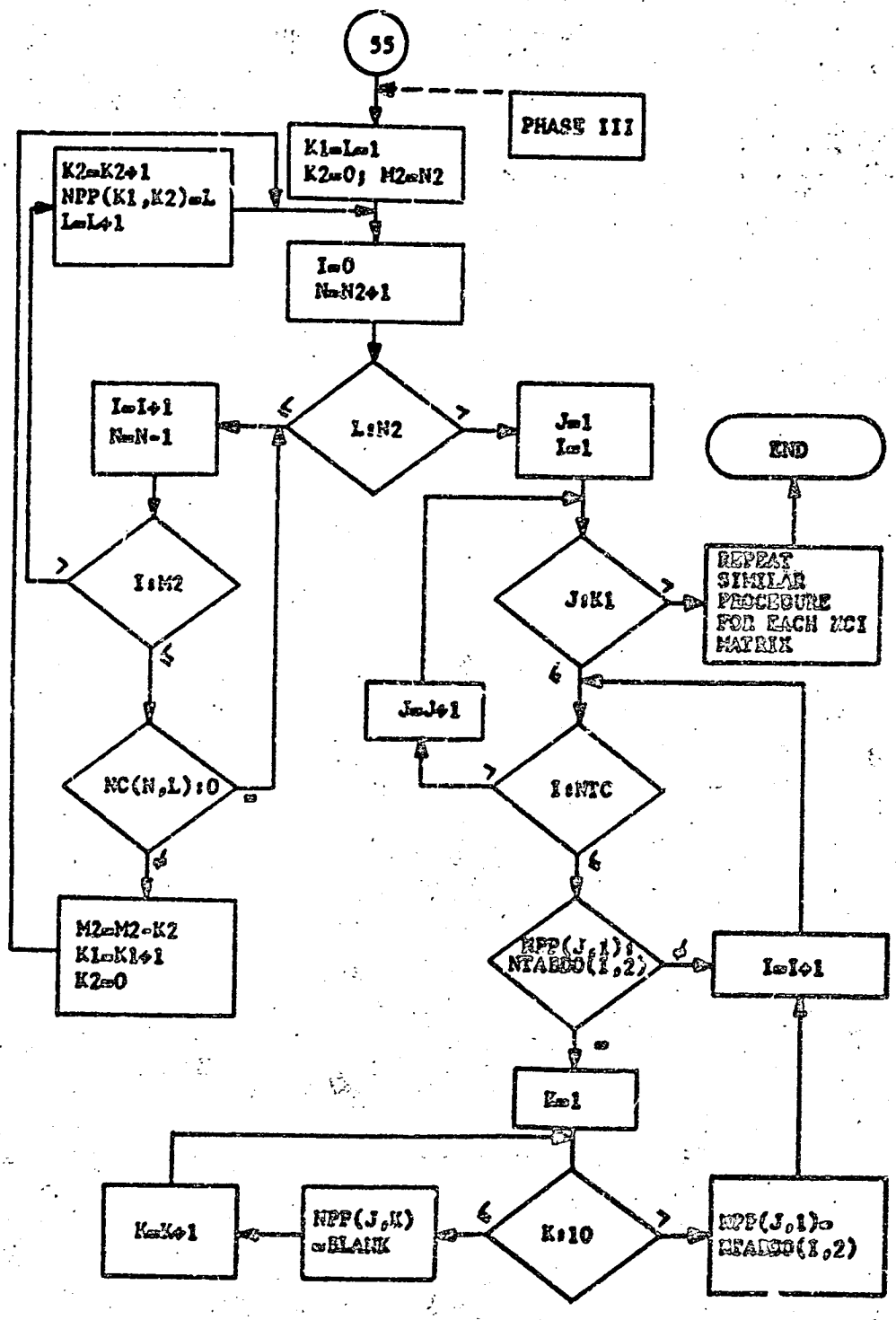


Figure 17. Flow Diagram of the Main Program. (Continued)

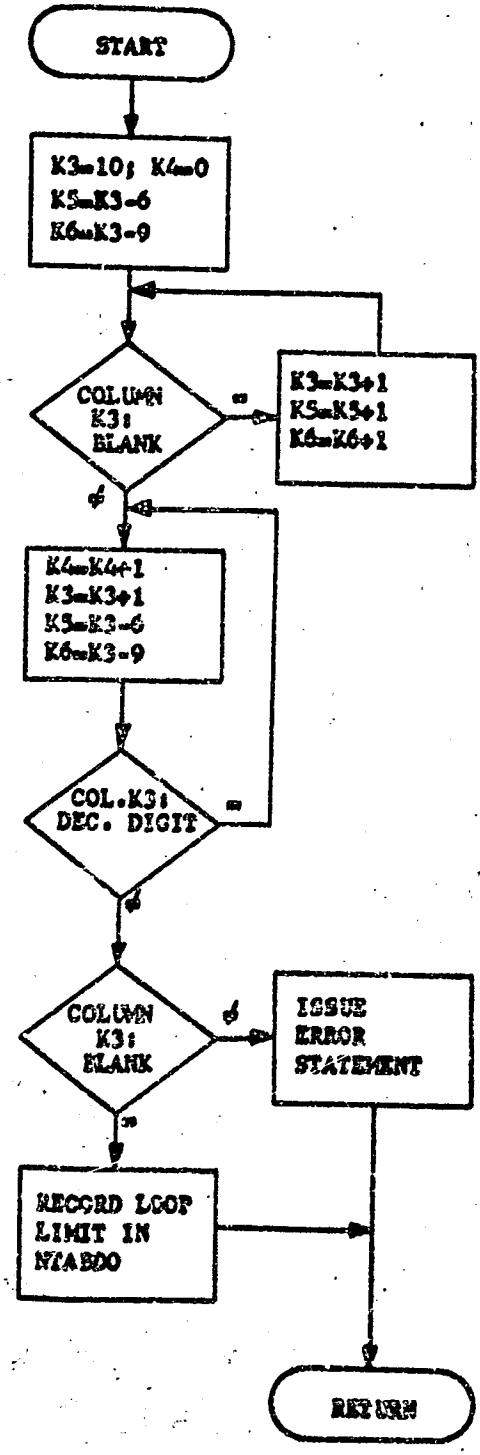


Figure 18. Flow Diagram of Subroutine MDO.

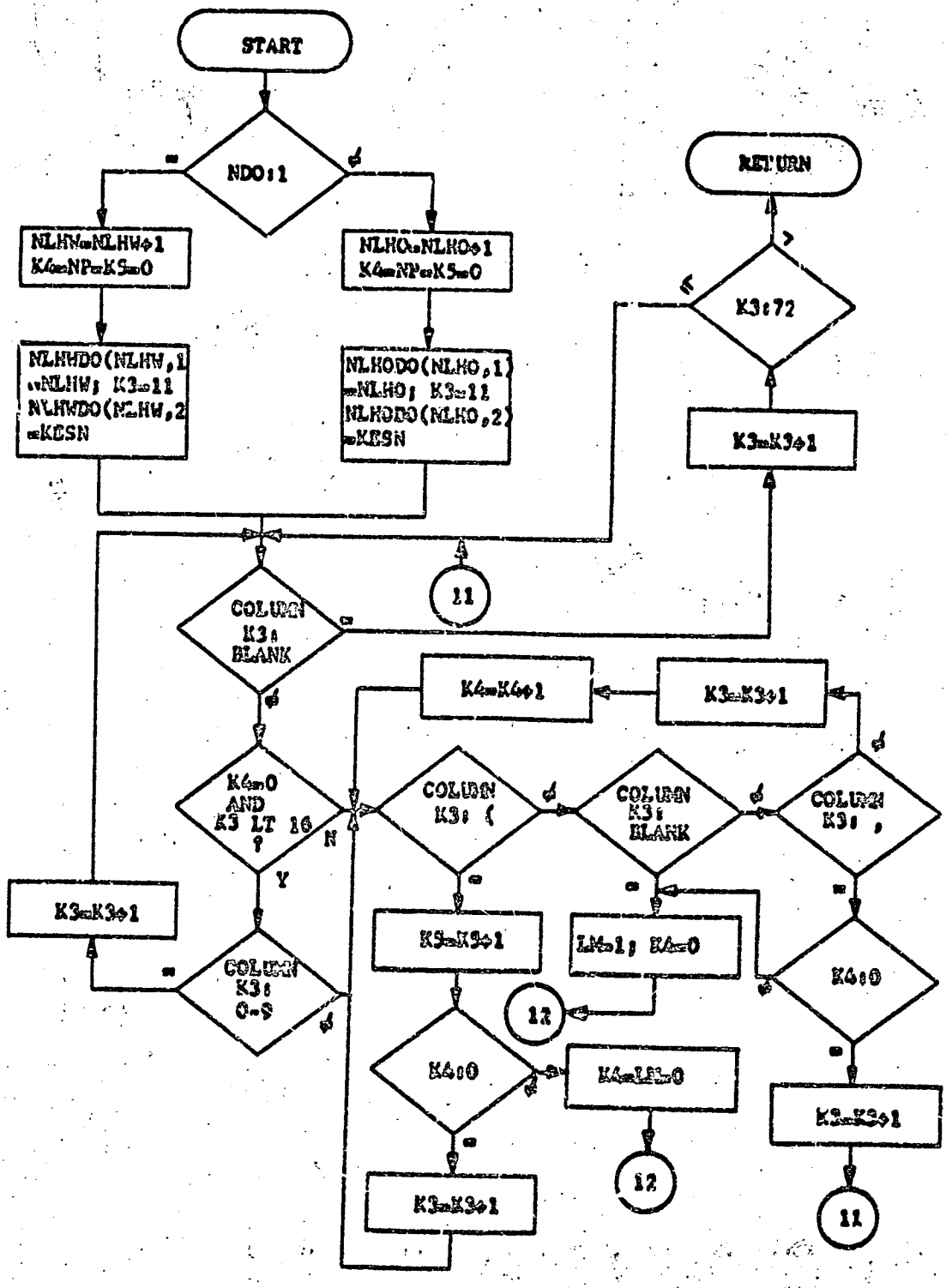


Figure 19. Flow Diagram of Subroutine IREAD.

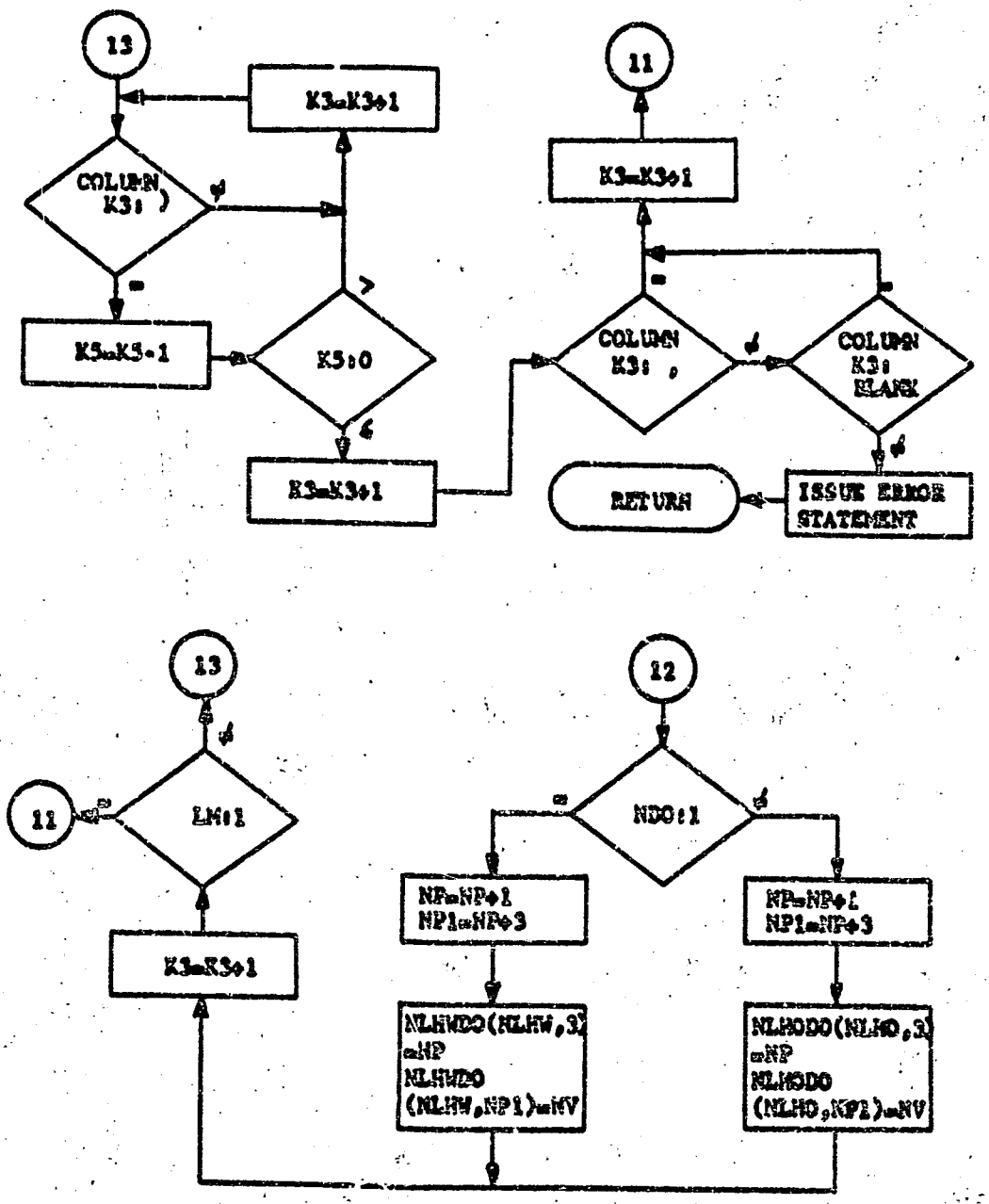


Figure 19. Flow Diagram of Subroutine MREAD. (Continued)

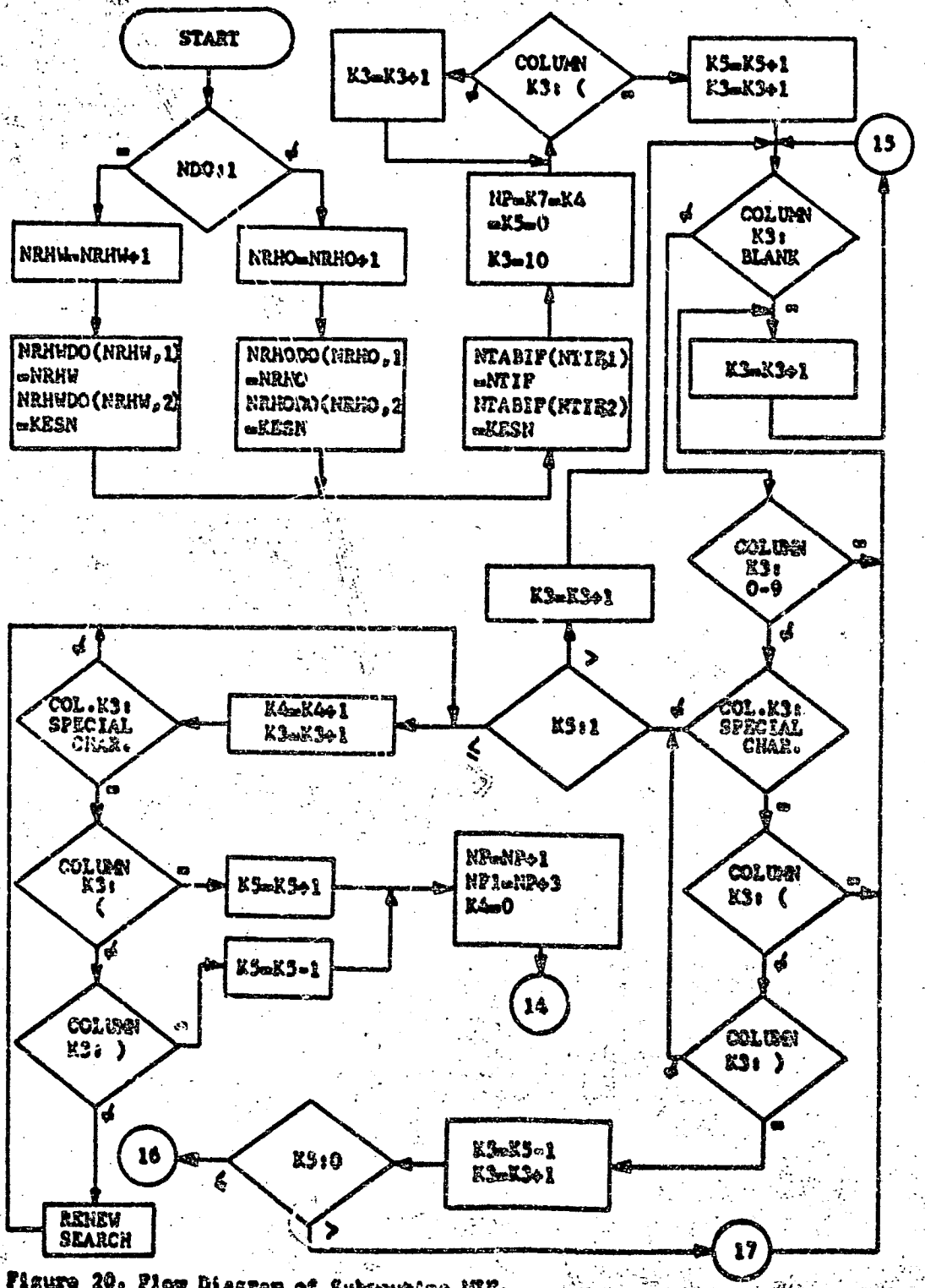


Figure 20. Flow Diagram of Subroutine NIF.

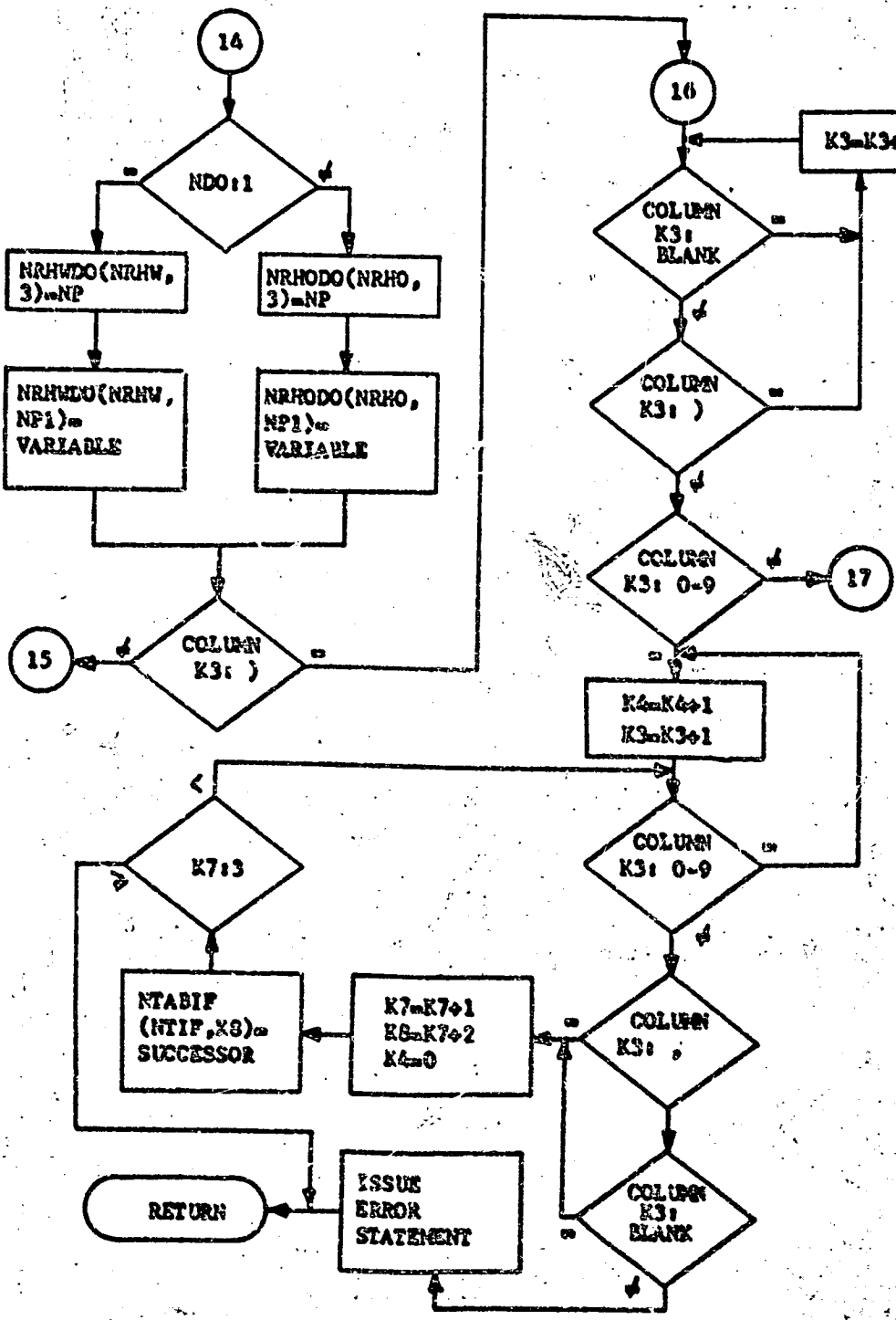


Figure 20. Flow Diagram of Subroutine NIF. (Continued)

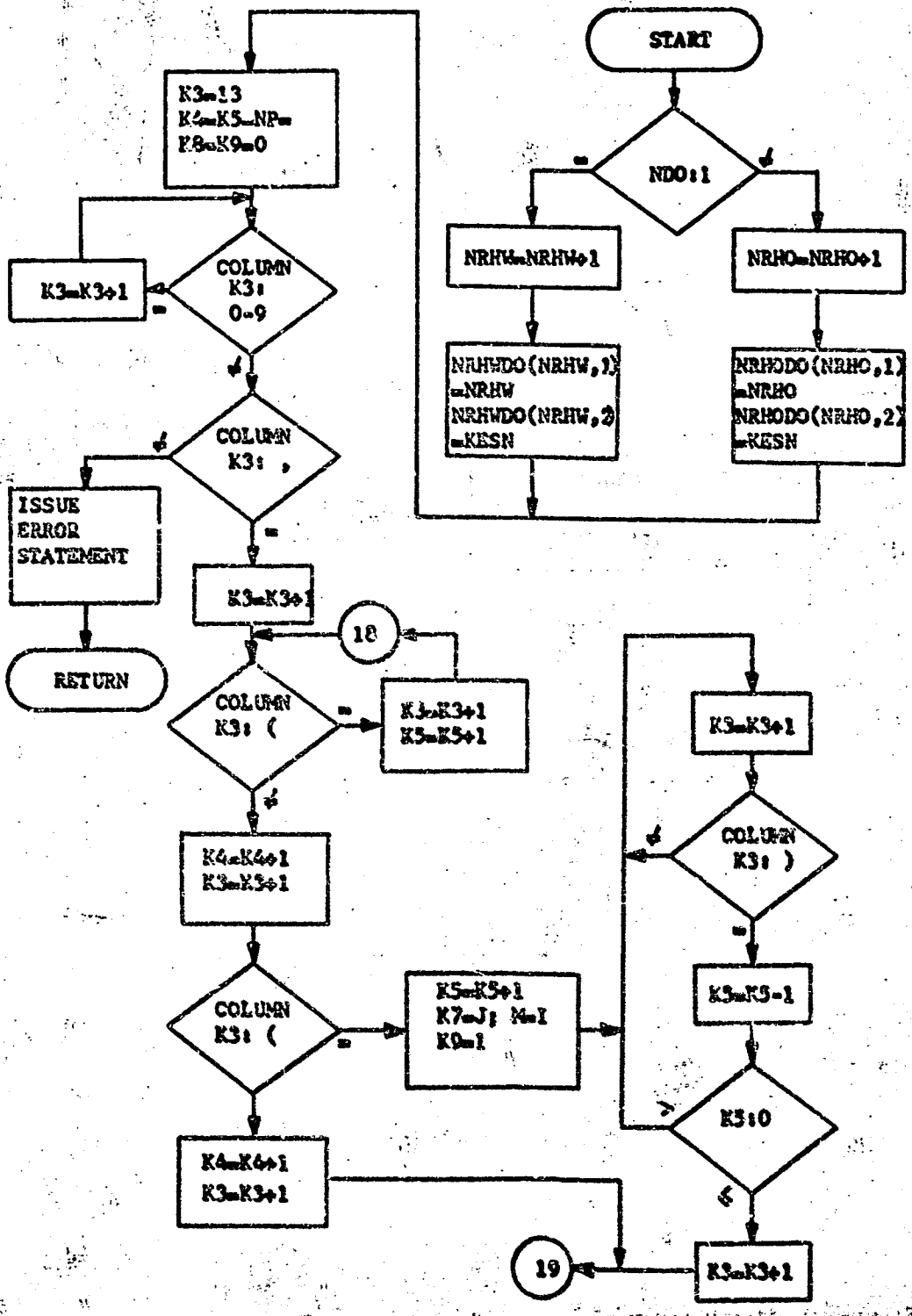


Figure 21. Flow Diagram of Subroutine NPRINT.

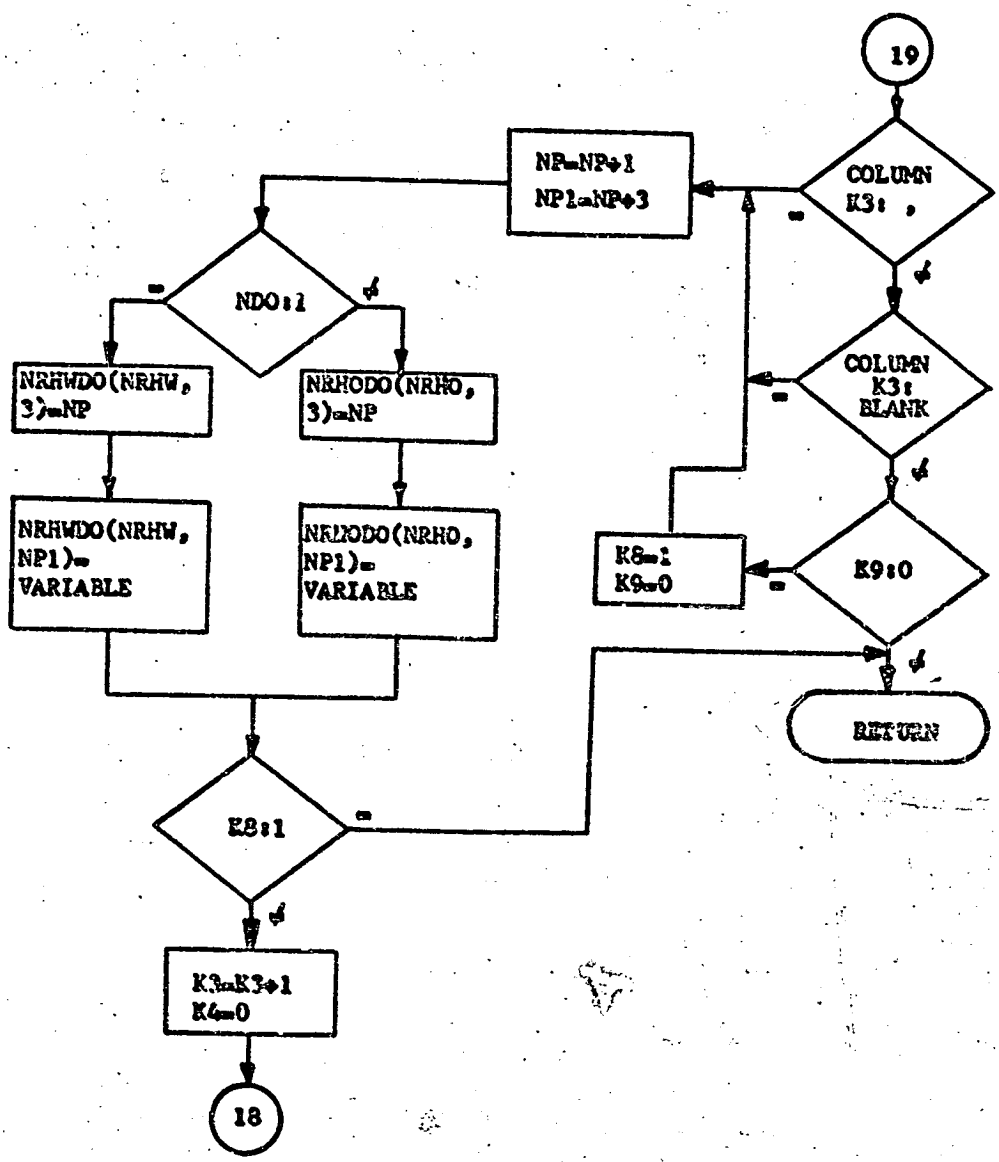


Figure 21. Flow Diagram of Subroutine MPRINT. (Continued)

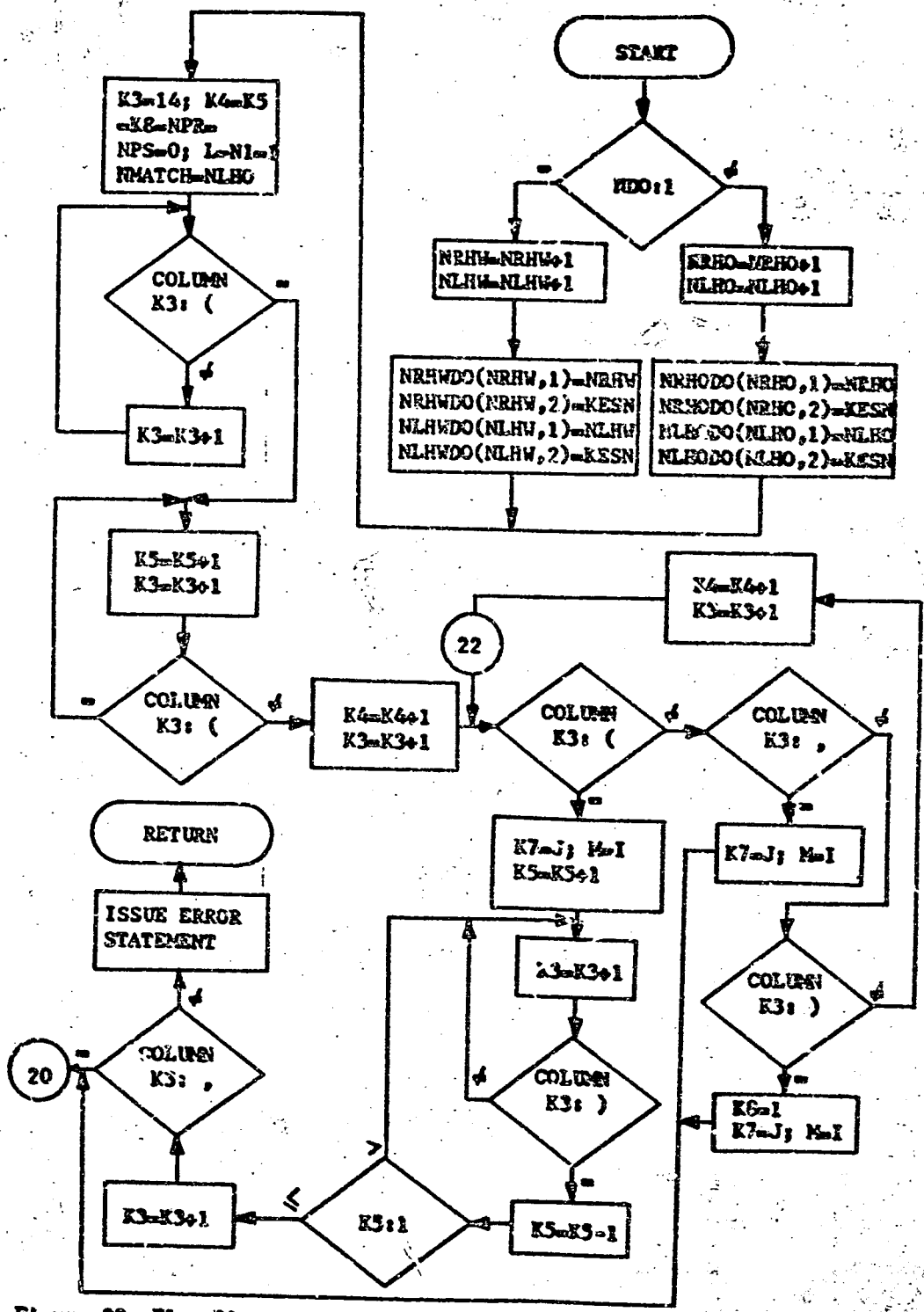


Figure 22. Flow Diagram of Subroutine MCALL.

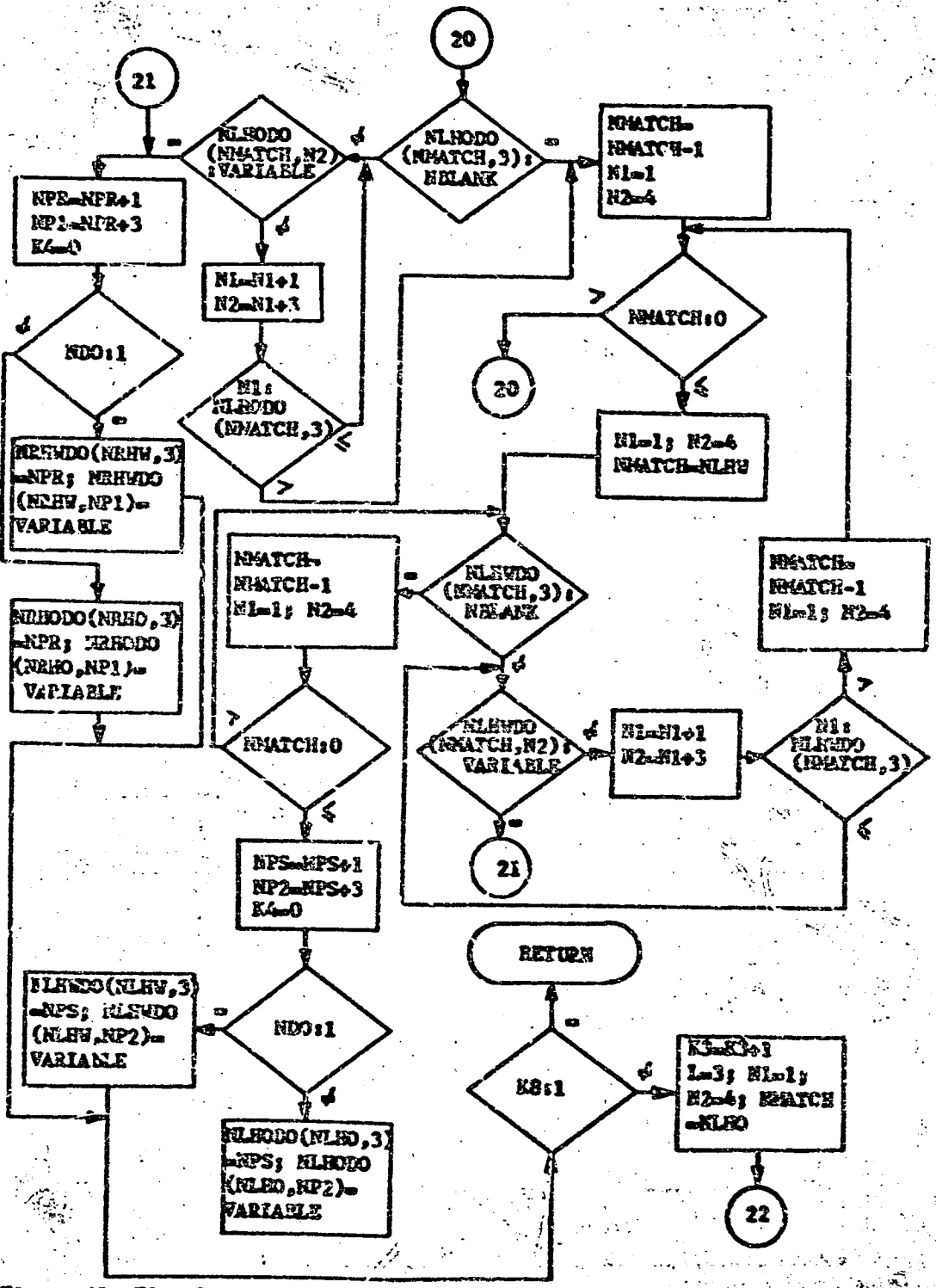


Figure 12. Flow Diagram of Subroutine MCALL. (Continued)

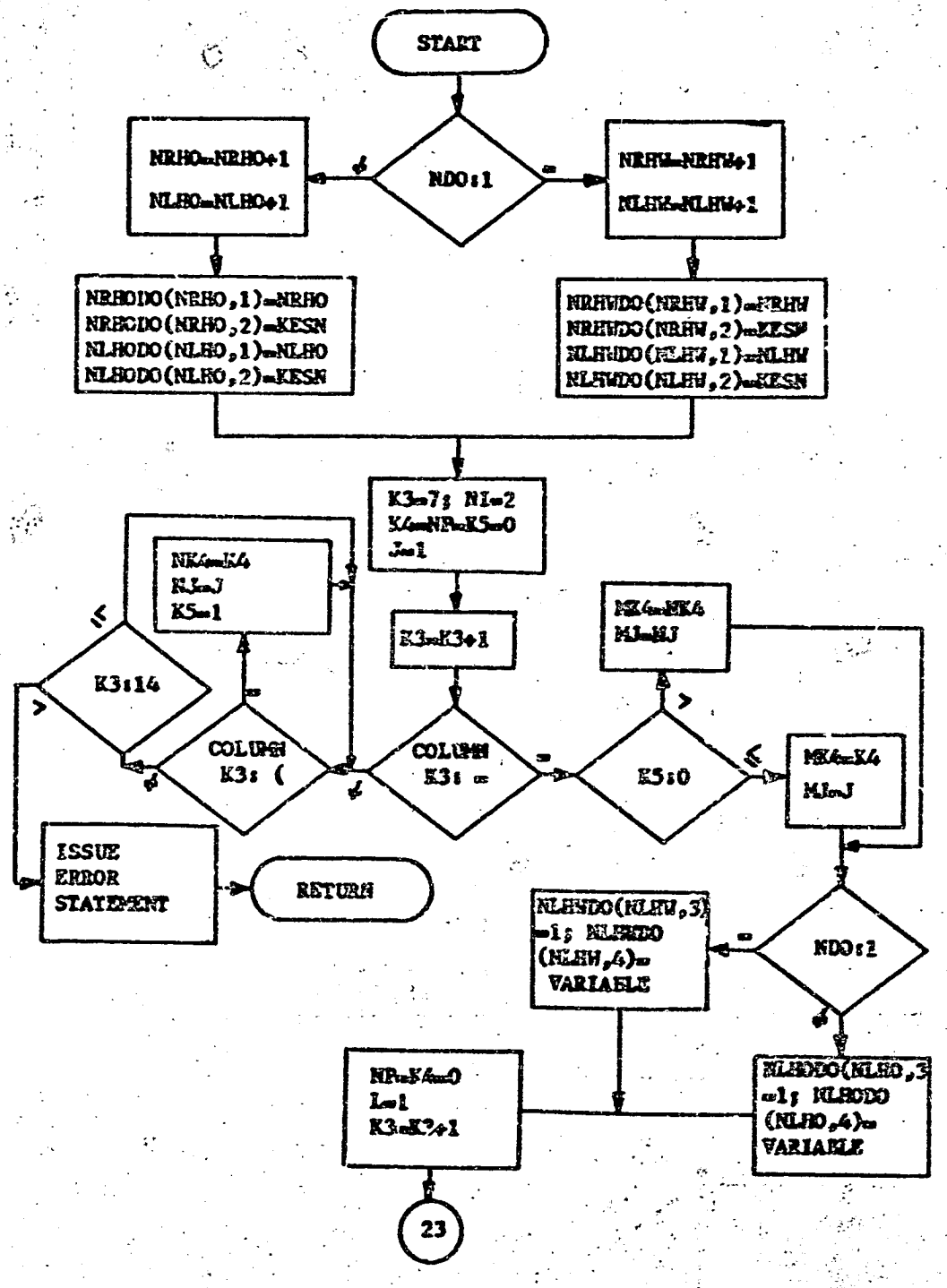


Figure 23. Flow Diagram of Subroutine MAE.

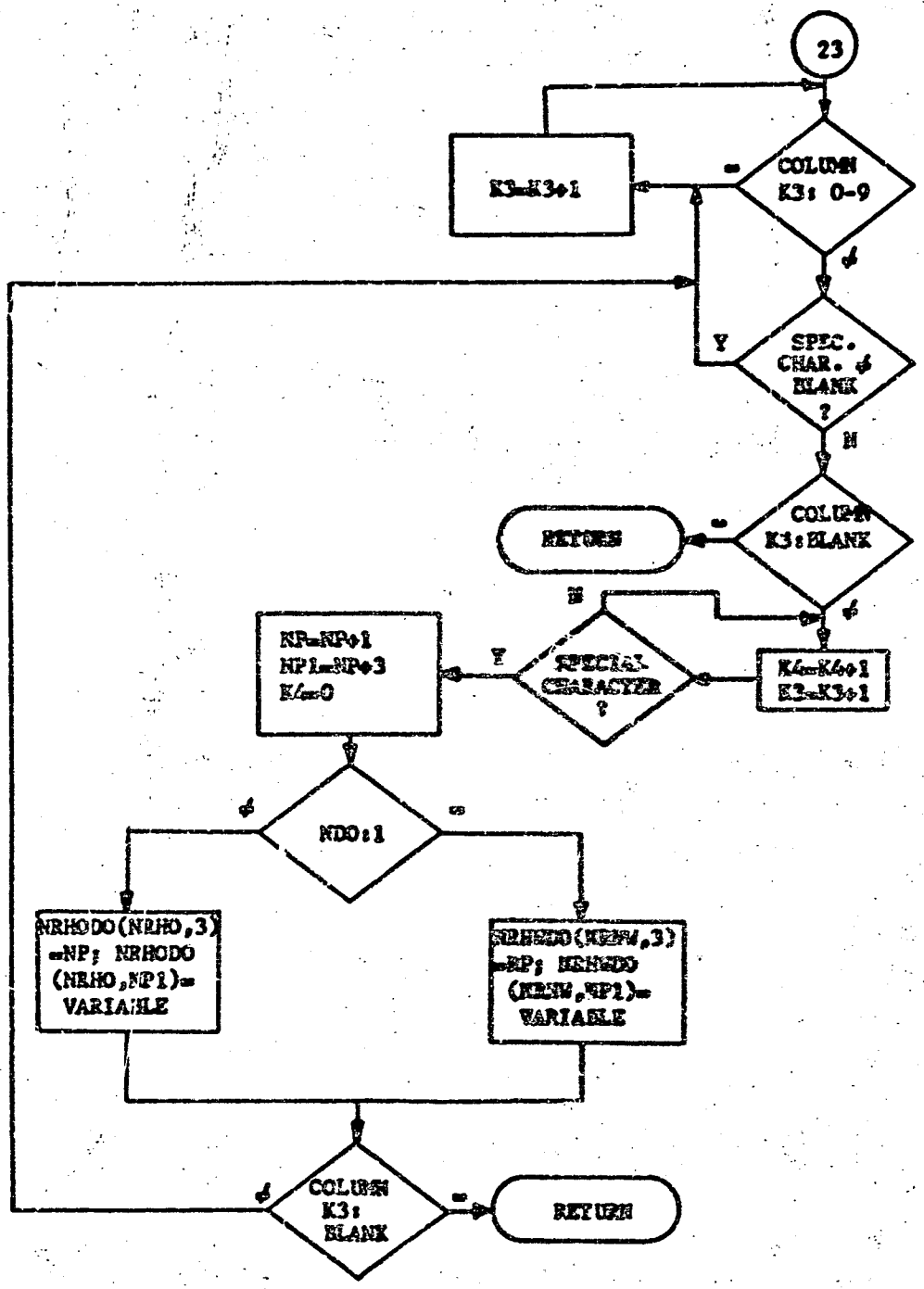


Figure 23. Flow Diagram of Subroutine M&E. (Continued)

```
      PROGRAM PPS(INPUT,OUTPUT)
      DIMENSION NEST(99),NTABESN(99,2),NES(99,11),NLHODO(50,7),
     1NLHWDO(50,7),NRHODO(50,7),NRHWDO(50,7),NTABDO(5,4),
     2NTABIF(20,8),NC(99,99),NC1(30,30),NC2(30,30),NC3(30,30),
     4NC4(30,30),NC5(30,30),NIS(10),EX(7),NONEX(5),EXT(8),
     5TEST(11),NX(10),MX(10),NMX(10),NM(5),NIFO(10,3),
     6NIFW(10,3),NPP(30,10)
      DATA NBLANK/55B/
      DATA BLANK/5555555555555555B/
C     FORMAT,REAL,COMMON,INTEGER,DIMENSION
      DATA(NONEX(I),I=1,5)/06172215012400000B,22050114000000000B,
     103171515171600000B,  11162405070522000B,  04111505162311178/
C     DO,READ,IF,PRINT,CONTINUE,CALL,END
      DATA (EX(I),I=1,7)/04170000000000000B,  22050104000000000B,
     111060000000000000B,  20221116240000000B,  03171624111625058,
     203011414000000000B,  05160400000000000B/
C     A,B,C,D,E,F,G,H
      DATA (EXT(I),I=1,8)/01B,02B,03B,04B,05B,06B,07B,10B/
      DATA(TEST(I),I=1,11)/77000000000000000B,  77770000000000000B,
     177777700000000000B,  77777777000000000B,  77777777770000000B,
     277777777777700000B,  77777777777777000B,  77777777777777778,
     31B,                  777777777777B,555555555555B/
      DATA (NM(I),I=1,5)/77B,7777B,777777B,77777777B,7777777777B/
      DATA (NX(I),I=1,8)/77000000000000  B,  77000000000000B,
     177000000000000B,  77000000000B,  77000000B,  7700000B,
     277000B,  77B/
      DATA(MX(I),I=1,8)/1000000000000000B,  100000000000000B,
     110000000000B,  1000000000B,  10000000B,  100000B,
     2100B,  1B/
      DATA (NMX(I),I=1,10)/33B,34B,35B,36B,37B,40B,
     141B,42B,43B,44B/
      TYPE INTEGER BLANK,EX,EXT,TEST
      DO 50 I=1,99
      NEST(I)=BLANK
   50 CONTINUE
      DO 52 J=1,2
      DO 51 I=1,99
   51 NTABESN(I,J)=BLANK
   52 CONTINUE
      DO 54 J=1,11
      DO 53 I=1,99
      NES(I,J)=BLANK
   53 CONTINUE
   54 CONTINUE
      DO 56 J=1,7
      DO 55 I=1,50
      NLHODO(I,J)=BLANK
      NLHWDO(I,J)=BLANK
   55 CONTINUE
   56 CONTINUE
      DO 58 J=1,7
      DO 57 I=1,50
      NRHODO(I,J)=BLANK
      NRHWDO(I,J)=BLANK
   57 CONTINUE
```

```
   58 CONTINUE
      DO 60 J=1,4
      DO 59 I=1,5
      NTABDO(I,J)=BLANK
   59 CONTINUE
   60 CONTINUE
      DO 62 J=1,8
      DO 61 I=1,20
      NTABIF(I,J)=BLANK
   61 CONTINUE
   62 CONTINUE
      DO 64 J=1,99
      DO 63 I=1,99
      NC(I,J)=0
   63 CONTINUE
   64 CONTINUE
      DO 66 J=1,30
      DO 65 I=1,30
      NC1(I,J)=0
      NC2(I,J)=0
      NC3(I,J)=0
      NC4(I,J)=0
      NC5(I,J)=0
   65 CONTINUE
   66 CONTINUE
      DO 94 J=1,10
      DO 92 I=1,3
      NIFW(J,I)=BLANK
      NIFO(J,I)=BLANK
   92 CONTINUE
   94 CONTINUE
      KSN=1
      KESN=NDO=NLHW=NLHO=NRHW=NRHO=NTDO=NTIF=NERR=0
C     READ NEXT SOURCE PROGRAM STATEMENT
   70 READ 2000,(NIS(I),I=1,10)
 2000 FORMAT(R6,9R8)
C     CHECK FOR END
      NIST=NIS(2)
      NIST=NIST.AND.TEST(3)
      IF(NIST.EQ.EX(7))290,72
C     CHECK FOR AN ALL-BLANK CARD
   72 DO 73 I=2,10
      IF(NIS(I).EQ.BLANK)73,75
   73 CONTINUE
   74 CONTINUE
   78 KSN=KSN+1
      GO TO 70
C     CHECK FOR COMMENT CARD
   75 NIST=NIS(1)
      NIST=NIST.AND.MX(3)
      NIST=NIST/MX(3)
      NIST=NIST.AND.MX(8)
      IF (NIST.EQ.EX(3))74,76
C     CHECK COLUMNS 1 TO 5
   76 NIST=NIS(1)
```

```
      MIST=NIST.AND.TEST(10)
      IF (TEST(11).EQ.MIST)80,77
C     CHECK FOR FORMAT IN COLS. 7-12
   77 MIST=NIS(2)
      MIST=NIST.AND.TEST(6)
      IF(MIST.EQ.NONEX(1))74,80
C     CHECK FOR REAL IN COLS. 7-10
   80 MIST=NIS(2)
      MIST=NIST.AND.TEST(4)
      IF (MIST.EQ.NONEX(2))74,81
C     CHECK FOR COMMON IN COLS. 7-12
   81 MIST=NIST.AND.TEST(5)
      IF (MIST.EQ.NONEX(3))74,82
C     CHECK FOR INTEGER IN COLS. 7-13
   82 MIST=NIST.AND.TEST(7)
      IF (MIST.EQ.NONEX(4))74,83
C     CHECK FOR DIMENSION IN COLS. 7-15
   83 MIST=NIST.AND.TEST(8)
      IF (MIST.EQ.NONEX(5))74,200
C     STATEMENT IS EXECUTABLE
  200 KESN=KESN+1
      NESN=KESN
      NES(KESN,1)=NESN
      DO 201 I=1,10
      J=I+1
      NES(KESN,J)=NIS(I)
  201 CONTINUE
C     CHECK FOR DO
      MIST=NIST.AND.TEST(2)
      IF (MIST.EQ.EX(1))220,203
C      CHECK FOR READ
  203 MIST=NIST.AND.TEST(4)
      IF (MIST.EQ.EX(2))230,204
C     CHECK FOR IF
  204 MIST=NIST.AND.TEST(2)
      IF(MIST.EQ.EX(3))240,205
C     CHECK FOR PRINT
  205 MIST=NIST.AND.TEST(5)
      IF (MIST.EQ.EX(4))250,206
C     CHECK FOR CONTINUE
  206 MIST=NIST.AND.TEST(8)
      IF (MIST.EQ.EX(5))260,207
C     CHECK FOR CALL
  207 MIST=NIST.AND.TEST(4)
      IF (MIST.EQ.EX(6))270,280
C     STATEMENT IS A DO
  220 NDO=1
      NEST(KESN)=EXT(I)
      NDOS=KESN
      NTDO=NTDO+1
      CALL MDO(NTDO,NIS,NTABDO,NERR,KESN)
  300 NTABESN(KESN,1)=KESN
      NTABESN(KESN,2)=NEST(KESN)
      IF(NTABIF(1,1).EQ.BLANK)74,340
C     SEE IF COLS. 1-5 MATCH SUCCESSORS OF IF STATEMENTS
```

```
  340 MIST=NSAD
      IF (MIST.EQ.TEST(11))74,350
C     ADDRESS FIELD IS NOT BLANK
  350 DO 365 I=1,NTIF
      IF(NTABIF(NTIF,1).EQ.BLANK)74,362
  362 DO 360 J=3,5
      IF(MIST.EQ.NTABIF(NTIF,J))363,360
  363 L=J+3
      NTABIF(NTIF,L)=KESN
  360 CONTINUE
  365 CONTINUE
      GO TO 74
C     STATEMENT IS A READ
  230 NEST(KESN)=EXT(2)
      CALL MREAD(NLHW,NLHO,NLHWDO,NLHODO,NDO,KESN,NIS,NX,MX,
     1NMX,NERR)
  235 IF (NTDO.EQ.0)300,236
C     LOOK FOR MATCH BETWEEN ADDRESS
C     FIELD OF KESN AND DO LOOP LIMIT
  236 I=9
      DO 239 J=1,5
      I=I-1
      IF(I.EQ.8)228,229
  228 NSAD=NIS(1)
      NSAD=NSAD.AND.TEST(7)
      NSAD=NSAD/MX(7)
      NSAD=NSAD.AND.77777777778
  229 NSD=NSAD
      NSD=NSD.AND.NX(I)
      NSD=NSD/MX(I)
      NSD=NSD.AND.NX(8)
      IF(NSD.EQ.NBLANK)231,239
  239 CONTINUE
  231 IF(I.EQ.8)232,233
  232 NSAD=NIS(1)
      GO TO 234
  233 NSAD=NSAD.AND.NM(8-I)
  234 IF(NSAD.EQ.TEST(11))300,237
  237 IF(NSAD.EQ.NTABDO(NTDO,3))238,300
  238 NDO=0
      NTABDO(NTDO,4)=KESN
      GO TO 300
C     STATEMENT IS AN IF
  240 NEST(KESN)=EXT(3)
      NTIF=NTIF+1 $ NBIF=NTIF
      CALL MIF(NRHW,NRHO,NRHWDO,NRHODO,KESN,NERR,NIS,NX,MX,
     1NMX,NDO,NTIF,NTABIF)
      GO TO 236
C     STATEMENT IS A PRINT
  250 NEST(KESN)=EXT(4)
      CALL MPRINT(NRHW,NRHO,NRHWDO,NRHODO,KESN,NIS,NX,MX,
     1NMX,NDO,NERR)
      GO TO 236
C     STATEMENT IS A CONTINUE
  260 NEST(KESN)=EXT(5)
```

```
      GO TO 236
C     STATEMENT IS A CALL
  270 NEST(KESN)=EXT(6)
      CALL MCALL(NDO,NRHO,NLHO,NRHW,NLHW,NRHODO,NLHODO,
     1NRHWDO,NLHWDO,KESN,NX,MX,NMX,NERR,NIS)
      GO TO 236
C     STATEMENT IS AN AE OR A REPLACEMENT
  280 NEST(KESN)=EXT(7)
      CALL MAE(NRHO,NLHO,NRHW,NLHW,NRHODO,NLHODO,NRHWDO,
     1NLHWDO,NDO,NX,MX,NMX,NERR,NIS,KESN)
      GO TO 236
  290 KESN=KESN+1
      NEST(KESN)=EXT(8)
      NTABESN(KESN,1)=KESN
      NTABESN(KESN,2)=NEST(KESN)
      NES(KESN,1)=KESN
      DO 292 I=1,10
      J=I+1
      NES(KESN,J)=NIS(I)
  292 CONTINUE
      GO TO 500
C     STATEMENT NUMBERS 500-999 COMPRISE PHASE II
  500 CONTINUE
      PRINT 2010
      DO 2015 J=1,KESN
      PRINT 2020,(NES(J,I),I=1,11)
 2015 CONTINUE
      PRINT 2025
      DO 2035 J=1,KESN
      PRINT 2030,(NTABESN(J,I),I=1,2)
 2035 CONTINUE
      PRINT 2040
      DO 2045 J=1,NBIF
      PRINT 2042,(NTABIF(J,I),I=1,8)
 2045 CONTINUE
      PRINT 2050
      DO 2055 J=1,NTDO
      PRINT 2052,(NTABDO(J,I),I=1,4)
 2055 CONTINUE
      PRINT 2060
      PRINT 2065
      DO 2070 J=1,NLHO
      PRINT 2075,(NLHODO(J,I),I=1,7)
 2070 CONTINUE
      PRINT 2080
      PRINT 2065
      DO 2085 J=1,NRHO
      PRINT 2075,(NRHODO(J,I),I=1,7)
 2085 CONTINUE
      PRINT 2090
      PRINT 2065
      DO 2092 J=1,NLHW
      PRINT 2075, (NLHWDO(J,I),I=1,7)
 2092 CONTINUE
      PRINT 2094
```

```
      PRINT 2065
      DO 2096 J=1,NRHW
      PRINT 2075,(NRHWDO(J,I),I=1,7)
 2096 CONTINUE
 2010 FORMAT(1H1,*THE EXECUTABLE STATEMENTS OF THE SOURCE*,
     1* PROGRAM*,/,1X,*AND THEIR STATEMENT NUMBERS ARE*,
     2* LISTED BELOW.*,/)
 2020 FORMAT(1H ,I2,R6,9R8)
 2025 FORMAT(1H1,2X,*NTABESN*,/,3X,*(1) (2)*)
 2030 FORMAT(1H ,I4,R4)
 2040 FORMAT(1H1,17X,*NTABIF*,/,*(1) (2) (3)  (4)*,
     1*  (5)  (6)  (7)  (8)*)
 2042 FORMAT(1H ,I2,1X,I2,1X,3(R4,1X),3(I4,1X))
 2050 FORMAT(1H ,//,7X,*NTABDO*,/,1X,*(1) (2)  (3)   (4)*)
 2052 FORMAT(1H ,2(I2,2X),R4,1X,I4)
 2060 FORMAT(1H1,16X,*NLHODO*)
 2065 FORMAT(1H ,*(1) (2) (3)   (4)     (5)     (6)     (7)*)
 2075 FORMAT(1H0,3(I2,2X),4(R7,1X))
 2080 FORMAT(1H1,16X,*NRHODO*)
 2090 FORMAT(1H1,16X,*NLHWDO*)
 2094 FORMAT(1H1,16X,*NRHWDO*)
      N1=1 $ NTC=NTIF=0
  502 IF(NTABESN(N1,1).EQ.BLANK)504,506
  504 N1=N1+1 $ GO TO 502
  506 CONTINUE
      IF(NTABESN(N1,2).EQ.EXT(1))600,510
  510 IF(NTABESN(N1,2).EQ.EXT(2))504,520
  520 IF(NTABESN(N1,2).EQ.EXT(3))800,522
  522 IF(NTABESN(N1,2).EQ.EXT(4))800,524
  524 IF(NTABESN(N1,2).EQ.EXT(5))504,526
  526 IF(NTABESN(N1,2).EQ.EXT(6))800,528
  528 IF(NTABESN(N1,2).EQ.EXT(7))800,530
  530 IF(NTABESN(N1,2).EQ.EXT(8))950,550
  550 PRINT 2100,N1,(NTABESN(N1,I),I=1,2)
 2100 FORMAT(1H1,*PPS WAS UNABLE TO DETECT ONE OF THE STANDARD*,
     1/,* FORMS IN THE TABLE OF EXECUTABLE STATEMENT NUMBERS*,
     2/,*N1=*,I3,*NTABESN(N1,1)=*,I3,*NTABESN(N1,2)=*,R1)
      GO TO 1999
C     STATEMENT IN THE 600S AND 700S ARE USED TO
C     ANALYZE SOURCE PROGRAM STATEMENTS WITHIN A DO LOOP.
  600 N2=NTABESN(N1,1) $ NS=0 $ NTC=NTC+1 $ NPO=NPW=NPT1=NPT2=0
  610 N4=0 $ NK=1 $ N1=N1+1
  612 IF(NTABESN(N1,1).EQ.BLANK)614,616
  614 N1=N1+1 $ GO TO 612
  616 N9=NTABESN(N1,1)
      NS=NS+1
  618 N4=N4+1 $ N3=N8=1 $ N6=N9=3
C     COMPARE KESN OF DO STATEMENT TO
C     KESN OF STATEMENTS PRECEDING DO
  620 IF(NRHWDO(NS,2)-N3)672,626,672
  626 IF(NLHODO(N4,2).GE.N2)650,628
  628 N7=N5+N6 $ N10=NR+N9
C     COMPARE RHV WITHIN DO TO LHV OUTSIDE DO
      IF(NRHWDO(NS,N7).EQ.NLHODO(N4,N10))630,632
  630 N11=NLHODO(N4,2)
```

```
      NC(N11,N2)=1
      GO TO 618
  632 IF(NRHWDO(NS,3).GT.N5)634,636
  634 N5=N5+1 $ GO TO 628
  636 N8=N8+1 $ N5=1
      IF(NLHODO(N4,3).LT.N8)618,628
  650 IF(N2.EQ.NTABDO(NK,2))670,652
  652 N4=1
  654 N5=N8=1 $ N6=N9=3
      N10=N8+N9
  656 N7=N5+N6
      IF(NRHWDO(NS,N7).EQ.NLHWDO(N4,N10))658,660
  658 N11=NTABDO(NK,1)
      NC(N11,N2)=1
      NK=NK+1
      GO TO 650
  660 IF(NRHWDO(NS,3).GT.N5)662,664
  662 N5=N5+1 $ GO TO 656
  664 N8=N8+1 $ N5=1
      IF(NLHWDO(N4,3).GE.N8)656,666
  666 N4=N4+1
      IF(NLHWDO(N4,2).EQ.NTABDO(NK,2))668,654
  668 NK=NK+1 $ GO TO 650
C     BEGIN COMPARISON OF RHV WITHIN DO TO LHV WITHIN DO
  670 N4=0
      IF(NTABESN(N1,2).EQ.EXT(5))672,674
  672 IF(N3.EQ.NTABDO(NTC,4))750,610
  674 N4=N4+1 $ N5=N8=1 $ N6=N9=3
  676 N10=N8+N9
  678 N7=N5+N6
  680 IF(NLHWDO(N4,2).GT.N2)684,682
  682 N4=N4+1 $ GO TO 680
  684 IF(NRHWDO(NS,2).GT.NLHWDO(N4,2))686,720
  686 IF(NRHWDO(NS,N7).EQ.NLHWDO(N4,N10))700,688
  688 IF(NRHWDO(NS,3).GT.N5)690,692
  690 N5=N5+1 $ GO TO 678
  692 N6=N8+1 $ N5=1
      IF(NLHWDO(N4,3).LT.N8)674,676
C     GENERATE ENTRIES IN NTC MATRICES
  700 N11=NLHWDO(N4,2)
      IF(NTC.EQ.1)702,704
  702 NC1(N11,N3)=1
      GO TO 718
  704 IF(NTC.EQ.2)706,708
  706 NC2(N11,N3)=1
      GO TO 718
  708 IF(NTC.EQ.3)710,712
  710 NC3(N11,N3)=1
      GO TO 718
  712 IF(NTC.EQ.4)714,716
  714 NC4(N11,N3)=1
      GO TO 718
  716 NC5(N11,N3)=1
      GO TO 718
  718 CONTINUE
```

```
      GO TO 688
C     CHECK FOR IF
  720 IF(NTABESN(N1,2).EQ.EXT(3))722,672
  722 NTIF=NTIF+1
      IF(NTABIF(NTIF,2).EQ.NTABESN(N1,1))724,722
  724 NM=6
C     STORE SUCCESSORS OF IF STATEMENT WITHIN DO
  725 IF(NTABIF(NTIF,NM).GT.NTABDO(NTC,4))726,734
  726 NPO=NPO+1 $ NPT1=NPT1+1
      NIFO(NPT1,1)=NPO
      NIFO(NPT1,2)=NTABDO(NTC,2)
      NIFO(NPT1,3)=NTABIF(NTIF,NM)
  728 NM=NM+1
      IF(NM.GT.8)672,725
  734 NPW=NPW+1 $ NPT2=NPT2+1
      NIFW(NPT2,1)=NPW
      NIFW(NPT2,2)=NTABESN(N1,1)
      NIFW(NPT2,3)=NTABIF(NTIF,NM)
      GO TO 728
C     MAKE ENTRY IN NCI FROM IF STATEMENT TO ITS SUCCESSORS
  750 NQ=1
  752 IF(NIFW(NQ,1).EQ.BLANK)754,758
  754 IF(NP.EQ.0)504,755
  755 DO 757 J=1,NP
      DO 756 I=1,3
      NIFW(J,I)=BLANK
  756 CONTINUE
  757 CONTINUE
      GO TO 504
  758 NA=NIFW(NQ,2)
      NB=NIFW(NQ,3)
      IF(NTC.EQ.1)764,770
  764 NC1(NA,NB)=1
      GO TO 766
  770 IF(NTC.EQ.2)772,774
  772 NC2(NA,NB)=1
      GO TO 766
  774 IF(NTC.EQ.3)776,778
  776 NC3(NA,NB)=1
      GO TO 766
  778 IF(NTC.EQ.4)780,782
  780 NC4(NA,NB)=1
      GO TO 766
  782 NC5(NA,NB)=1
  766 NQ=NQ+1
      GO TO 752
C     STATEMENTS IN THE 800S ANALYZE ALL SOURCE PROGRAM
C     EXECUTABLE STATEMENTS OUTSIDE OF A DO.
  800 NK=NS=1 $ N4=0
      N2=NTABESN(N1,1)
  802 IF(NRHODO(NS,2).EQ.N2)806,804
  804 IF(NRHODO(NS,2).EQ.BLANK)504,805
  805 NS=NS+1 $ GO TO 802
  806 N4=N4+1
      IF(NLHODO(N4,2).LT.N2)808,840
```

```
  808 N5=NR=1 $ N6=N9=3
  810 N10=N8+N9
  812 N7=N5+N6
C     COMPARE RHV OUTSIDE DO TO LHV OUTSIDE DO
      IF(NRHODO(NS,N7).EQ.NLHODO(N4,N10))814,816
  814 N11=NLHODO(N4,2)
      NC(N11,N2)=1
      GO TO 806
  816 N5=N5+1
      IF(NRHODO(NS,3).LT.N5)818,812
  818 N8=NR+1 $ N5=1
      IF(NLHODO(N4,3).LT.N8)806,810
  840 N4=0
  841 IF(NTABDO(NK,1).EQ.BLANK)870,842
  842 IF(NTABDO(NK,2).LT.N2)844,870
  844 N4=N4-1 $ N5=N8=1 $ N6=N9=3
      IF(NLHWDO(N4,2).EQ.BLANK)870,845
  845 IF(NLHWDO(N4,2).LT.N2)846,870
  846 N10=N8+N9
C     COMPARE RHV OUTSIDE DO TO LHV WITHIN DO
  848 N7=N5+N6
      IF(NRHODO(NS,N7).EQ.NLHWDO(N4,N10))850,860
  850 N11=NTABDO(NK,2)
      NC(N11,N2)=1
      NK=NK+1
      GO TO 841
  860 IF(NRHODO(NS,3).GT.N5)862,864
  862 N5=N5+1
      GO TO 848
  864 N8=NR+1 $ N5=1
      IF(NLHWDO(N4,3).LT.N8)844,846
C     CHECK FOR IF STATEMENT OUTSIDE DO
  870 IF(NTABESN(N1,2).EQ.EXT(3))872,504
  872 NTIF=NTIF+1 $ NM=6
C     STORE SUCCESSORS OF IF STATEMENT OUTSIDE DO
  874 NPO=NPO+1 $ NPT1=NPT1+1
      NIFO(NPT1,1)=NPO
      NIFO(NPT1,2)=NTABIF(NTIF,2)
      NIFO(NPT1,3)=NTABIF(NTIF,NM)
      NM=NM+1
      IF(NM.GT.8)504,874
C     MAKE ENTRIES IN NC FOR SUCCESSORS
C     OF IF STATEMENT OUTSIDE DO
  950 NQ=1
  951 IF(NIFO(NQ,1).EQ.BLANK)1000,952
  952 NA=NIFO(NQ,2)
      NB=NIFO(NQ,3)
      NC(NA,NB)=1
      NQ=NQ+1 $ GO TO 951
 1000 CONTINUE
      PRINT 1800
      DO 1030 J=1,20
      PRINT 1805,J,(NC(J,I),I=1,20)
 1030 CONTINUE
      PRINT 1810
```

```
      DO 1032 J=1,20
      PRINT 1805,J,(NC1(J,I),I=1,20)
 1032 CONTINUE
      K1=L=1 $ K2=0$ M2=N2
 1040 I=0 $ N=N2+1
      IF(L.GT.N2)1070,1051
 1051 I=I+1 $ N=N-1
      IF(I.GT.M2)1054,1052
 1052 IF(NC(N,L).EQ.0)1051,1060
 1054 K2=K2+1
      NPP(K1,K2)=L
      L=L+1
      GO TO 1040
 1060 M2=M2-K2
      K1=K1+1 $ K2=0
      GO TO 1040
 1070 DO 1080 J=1,K1
      DO 1078 I=1,NTC
      IF(NPP(J,1).EQ.NTABDO(I,2))1072,1078
 1072 DO 1074 K=1,10
      NPP(J,K)=0
 1074 CONTINUE
      NPP(J,1)=NTABDO(I,2)
 1078 CONTINUE
 1080 CONTINUE
      PRINT 1090
 1090 FORMAT(1H1,*THE BLOCKS CONTAINING THE STATEMENT NUMBERS*,
     1* OF THE*,/,1X,*INPUT PROGRAM WHICH CAN BE EXECUTED IN*,
     2* PARALLEL ARE GIVEN BELOW*)
      DO 1094 J=1,K1
      PRINT 1092,(NPP(J,I),I=1,10)
 1094 CONTINUE
 1092 FORMAT(1H0,10X,10I5)
 1800 FORMAT(1H1,20X,*NC*,/,5X,*1 2 3 4 5 6 7 8 9 0*,
     1* 1 2 3 4 5 6 7 8 9 0*)
 1805 FORMAT(1H ,I3,20I2)
 1810 FORMAT(1H0,20X,*NC1*,/,5X,*1 2 3 4 5 6 7 8 9 0*,
     1* 1 2 3 4 5 6 7 8 9 0*)
 1999 CONTINUE
      END
      SUBROUTINE MDO(NTDO,NIS,NTABDO,NERR,KESN)
      DIMENSION NTABDO(5,4),N(10),NUM(10),M(7),L(4),NM(5),NIS(10)
      DATA BLANK/1R /
      DATA (N(I),I=1,8)/77000000000000 B, 770000000000B,
     177700000000B, 7700000000B, 770000B, 7700B,
     27700B, 77B/
      DATA (M(I),I=1,5)/10000000B, 1000000B, 100000B, 10000B, 1000B/
      DATA(NUM(I),I=1,10)/33B,34B,35B,36B,37B,40B,41B,42B,43B,44B/
      DATA (L(I),I=1,4)/777777700B, 77777700000B,
     177770000000B, 7700000000B/
      DATA(NM(I),I=1,5)/77B,7777B,777777B,77777777B,7777777777B/
      TYPE INTEGER BLANK
      K3=10 $ K4=0
      K5=K3-6 $ K6=K3-9
   22 K=NIS(2)
```

```
      K=K.AND.N(K5)
      K=K/M(K6)
      K=K.AND.N(8)
C     CHECK FOR BLANK IN COLUMN K3
      IF (K.EQ.BLANK)30,40
   30 K3=K3+1
      K5=K5+1
      K6=K6+1
      GO TO 22
C     COLUMN K3 DOES NOT CONTAIN A BLANK
   40 K4=K4+1
      K3=K3+1
      K5=K3-6
      K6=K3-9
C     CHECK FOR DECIMAL DIGIT IN COLUMN K3
      K=NIS(2)
      K=K.AND.N(K5)
      K=K/M(K6)
      K=K.AND.N(8)
      DO 50 J=1,10
      IF (K.EQ.NUM(J))40,50
   50 CONTINUE
   60 IF (K.EQ.BLANK)70,65
   65 NERR=1
      PRINT 100,NTDO,K3,K4,K5,K6
  100 FORMAT(1H ,*ERROR IN FORMAT OF DO STATEMENT*,
     11H ,5HNTDO=,I2,3HK3=,I2,2X,3HK4=,I2,2X,3HK5=,
     2I2,2X,3HK6=,I2)
      GO TO 99
C      LOOP LIMIT HAS BEEN READ
   70 GO TO (72,74,76,78),K4
   72 K7=4 $ K8=1
      GO TO 80
   74 K7=3 $ K8=2
      GO TO 80
   76 K7=2 $ K8=3
      GO TO 80
   78 K7=1 $ K8=4
C      MIS CONTAINS LOOP LIMIT
   80 MIS=NIS(2).AND.L(K7)
      MIS=MIS/M(K8)
      MIS=MIS.AND.NM(K4)
      NTABDO(NTDO,1)=NTDO
      NTABDO(NTDO,2)=KESN
      NTABDO(NTDO,3)=MIS
   99 RETURN
      END
      SUBROUTINE MREAD(NLHW,NLHO,NLHWDO,NLHODO,
     1NDO,KESN,NIS,NX,MX,NMX,NERR)
      DIMENSION NLHWDO(50,7),NLHODO(50,7),NIS(10),
     1NX(10),MX(10),NMX(10)
      DATA COMMA,LP,RP/56B,51B,52B/
      DATA BLANK/55B/
      TYPE INTEGER COMMA,BLANK,RP
      IF(NDO.EQ.1)20,30
```

```
   20 NLHW=NLHW+1
      NLHWDO(NLHW,1)=NLHW
      NLHWDO(NLHW,2)=KESN
      GO TO 40
   30 NLHO=NLHO+1
      NLHODO(NLHO,1)=NLHO
      NLHODO(NLHO,2)=KESN
   40 K3=11 $ K4=NP=K5=0
      L=1
      DO 82 I=2,10
      IF(I.EQ.2)41,42
   41 K6=5
      GO TO 43
   42 K6=1
   43 DO 80 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      GO TO (44,47,56,62),L
C     CHECK FOR BLANK IN COLUMN K3
   44 IF (K.EQ.BLANK)45,90
   45 K3=K3+1
      IF (K3-72)46,46,99
   46 L=1
      GO TO 80
C     CHECK FOR LEFT PARENTHESIS
   47 IF(K.EQ.LP)48,49
   49 IF(K.EQ.BLANK)72,70
   48 K5=K5+1
      IF(K4.EQ.0)50,52
   50 K3=K3+1
      L=2
      GO TO 80
   52 CALL NVAR(NIS,I,K4,J,NV)
      LM=K4=0 $ GO TO 73
   54 K3=K3+1
      L=3
      GO TO 80
   56 IF(K.EQ.RP)58,54
   58 K5=K5-1
      IF(K5.GT.0)54,60
   60 K3=K3+1
      L=4
      GO TO 80
   62 IF(K.EQ.COMMA)64,66
   64 K3=K3+1
      L=1
      GO TO 80
   66 IF(K.EQ.BLANK)64,96
C      CHECK FOR END OF VARIABLE
   70 IF(K.EQ.COMMA)71,78
   71 IF(K4.EQ.0)64,72
   72 CALL NVAR(NIS,I,K4,J,NV)
      LM=1 $ K4=0
```

```
   73 IF(NDO.EQ.1)74,76
   74 NP=NP+1
      NP1=NP+3
      NLHWDO(NLHW,3)=NP
      NLHWDO(NLHW,NP1)=NV
      IF(LM.EQ.1)64,54
   76 NP=NP+1
      NP1=NP+3
      NLHODO(NLHO,3)=NP
      NLHODO(NLHO,NP1)=NV
      IF(LM.EQ.1)64,54
C     CHECK FORMAT STATEMENT NUMBER IN READ STATEMENT
   90 IF(K4.EQ.0 .AND.K3.LE.16)91,47
   91 DO 92 L=1,10
      IF(K.EQ.NMX(L))93,92
   92 CONTINUE
      GO TO 47
   93 K3=K3+1
      L=1
      GO TO 80
   78 K4=K4+1
      K3=K3+1
      L=2
   80 CONTINUE
   82 CONTINUE
   96 PRINT 100,K3,K4,K5,K6,KESN
      NERR=2
  100 FORMAT(1H ,*NO COMMA FOLLOWING RIGHT PARENTHESIS*,
     1* AFTER PARAMETER NAME*,1H ,3HK3=,I2,2X,3HK4=,I2,2X,
     23HK5=,I2,2X,3HK6=,I2,2X,5HKESN=,I2)
   99 RETURN
      END
      SUBROUTINE NVAR (NIS,I,K4,J,K)
      DIMENSION NIS(10),N(8),M(8),NM(7)
      DATA (N(L),L=1,8)/77000000000000 B, 77770000000000003,
     177777700000000008, 77777777000000008, 77777777700000008,
     277777777777700008, 77777777777770008, 77777777777777778/
      DATA (M(L),L=1,8)/1B,100B,10000B,1000000B,100000000B,
     110000000000B,1000000000000B,100000000000000B/
      DATA(NM(L),L=1,7)/77B,7777B,777777B,77777777B,7777777777B,
     177777777777B,777777777777778/
      IF(J-K4)20,20,10
   10 L1=10-J
      L2=9-L1
      K=NIS(I).AND.N(L2)
      K=K/M(L1)
      K=K.AND.NM(K4)
      GO TO 99
   20 IF(J-1)25,25,30
   25 J1=J
      GO TO 35
   30 J1=J-1
   35 L3=J1-1
      L4=K4-J+1
      L5=6*(J-1)
```

```
      L6=8-L3
      L7=9-L6
      K1=NIS(I-1).AND.NM(L4)
      K1=K1*(2**L5)
      IF(J-1)40,40,45
   40 K=K1
      GO TO 99
   45 K2=NIS(I).AND.N(L3+1)
      K2=K2/M(L6)
      K2=K2.AND.NM(L7)
      K=K1.OR.K2
   99 RETURN
      END

      SUBROUTINE MIF(NRHW,NRHO,NRHWDO,NRHODO,KESN,NERR,NIS,NX,
     1MX,NMX,NDO,NTIF,NTABIF)
      DIMENSION NRHWDO(50,7),NRHODO(50,7),NX(10),
     1MX(10),NMX(10),NTABIF(20,8),NIS(10)
      DATA BLANK,MUL,DIV,ADD,EXP,PER,SUB,LP,RP,COM/
     11R ,47B,50B,45B,47478,57B,46B,51B,52B,56B/
      TYPE INTEGER BLANK,DIV,ADD,EXP,PER,SUB,COM,RP
      IF(NDO.EQ.1)10,20
   10 NRHW=NRHW+1
      NRHWDO(NRHW,1)=NRHW
      NRHWDO(NRHW,2)=KESN
      GO TO 30
   20 NRHO=NRHO+1
      NRHODO(NRHO,1)=NRHO
      NRHODO(NRHO,2)=KESN
   30 NTABIF(NTIF,1)=NTIF
      NTABIF(NTIF,2)=KESN
      NP=K4=0
      K3=9 $ L=1 $ K5=0 $ K7=0
      DO 160 I=2,10
      IF(I.EQ.2)41,42
   41 K6=3
      GO TO 43
   42 K6=1
   43 DO 150 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      GO TO (44,48,64,80,90),L
C     LOOK FOR LEADING LEFT PARENTHESIS
   44 IF(K.EQ.LP)47,45
   45 K3=K3+1
      L=1
      GO TO 150
   46 K3=K3+1
      L=2
      GO TO 150
   47 K5=K5+1
      GO TO 46
   48 IF(K.EQ.BLANK)46,50
```

```
C       LOOK FOR A DECIMAL DIGIT IN COLUMN K3
   50 DO 52 N=1,10
      IF(K.EQ.NMX(N))46,52
   52 CONTINUE
C       LOOK FOR A SPECIAL CHARACTER IN COLUMN K3
   54 IF(K.EQ.MUL)46,55
   55 IF(K.EQ.DIV)46,56
   56 IF(K.EQ.ADD)46,57
   57 IF(K.EQ.PFR)46,58
   58 IF(K.EQ.SUB)46,59
   59 IF(K.EQ.LP)47,60
   60 IF(K.EQ.RP)110,61
   61 IF(K5-1)62,62,63
   63 K3=K3+1
      L=2
      GO TO 150
   62 K4=K4+1
      K3=K3+1
      L=3
      GO TO 150
C       LOOK FOR SPECIAL CHARACTER WHICH WOULD TERMINATE
C       VARIABLE NAME
   64 IF(K.EQ.MUL)70,65
   65 IF(K.EQ.DIV)70,66
   66 IF(K.EQ.ADD)70,67
   67 IF(K.EQ.SUB)70,68
   68 IF(K.EQ.LP) 71,69
   69 IF(K.EQ.RP)115,62
   70 CALL NVAR(NIS,I,K4,J,NV)
      NP=NP+1
      NP1=NP+3
      K4=0
      IF(NDO.EQ.1)72,74
   71 K5=K5+1
      GO TO 70
   72 NRHWDO(NRHW,3)=NP
      NRHWDO(NRHW,NP1)=NV
      GO TO 76
   74 NRHODO(NRHO,3)=NP
      NRHODO(NRHO,NP1)=NV
   76 IF(K.EQ.RP)78,46
   78 K3=K3+1
      L=4
      GO TO 150
   80 IF(K.EQ.BLANK)78,82
   82 IF(K.EQ.RP)78,84
   84 DO 86 N=1,10
      IF(K.EQ.NMX(N))88,86
   86 CONTINUE
      GO TO 46
C       SUCCESSOR OF IF STATEMENT HAS BEEN FOUND
   88 K4=K4+1
      K3=K3+1
      L=5
      GO TO 150
```

```
 90 DO 92 N=1,10
    IF(K.EQ.NMX(N))88,92
 92 CONTINUE
 94 IF(K.EQ.COM)98,96
 96 IF(K.EQ.BLANK)98,97
 97 PRINT 100,KESN,K3,K4,K6,I,J
    NERR=3
    GO TO 170
100 FORMAT(1H ,*CHARACTER OTHER THAN COMMA FOLLOWING*,
   1* FIRST OR SECOND POSSIBLE SUCCESSOR*,1H ,5HKESN=,I2,2X,
   23HK3=,I2,2X,3HK4=,I2,2X,3HK6=,I2,2X,2HI=,I2,2X,2HJ=,I2)
 98 K7=K7+1
    K8=K7+2
    CALL NVAR(NIS,I,K4,J,NV)
    K4=0
    NTABIF(NTIF,K8)=NV
    IF(K7-3)105,170,170
105 K3=K3+1
    L=5
    GO TO 150
110 K5=K5-1
    IF(K5)78,78,46
115 K5=K5-1
    GO TO 70
150 CONTINUE
160 CONTINUE
170 RETURN
    END

    SUBROUTINE MPRINT(NRHW,NRHO,NRHWDO,NRHODO,KESN,
   1NIS,NX,MX,NMX,NDO,NERR)
    DIMENSION NRHODO(50,7),NRHWDO(50,7),NIS(10),
   1NX(10),MX(10),NMX(10)
    DATA BLANK,LP,RP,COM/55B,51B,52B,56B/
    TYPE INTEGER BLANK,RP,COM
    IF(NDO.EQ.1)10,20
 10 NRHW=NRHW+1
    NRHWDO(NRHW,1)=NRHW
    NRHWDO(NRHW,2)=KESN
    GO TO 30
 20 NRHO=NRHO+1
    NRHODO(NRHO,1)=NRHO
    NRHODO(NRHO,2)=KESN
 30 K4=0 $ K3=12 $ K5=NP=0 $ L=1 $ K8=0 $ K9=0
    DO 300 I=2,10
    IF(I.EQ.2)41,42
 41 K6=6
    GO TO 43
 42 K6=1
 43 DO 200 J=K6,8
    K=NIS(I)
    K=K.AND.NX(J)
    K=K/MX(J)
    K=K.AND.NX(8)
    GO TO (44,55,85,100,110,125),L
```

```
   44 IF(K.EQ.BLANK)50,55
   50 K3=K3+1
      L=1
      GO TO 200
C     LOOKFOR PRINT FORMAT STATEMENT NUMBER
   55 DO 60 N=1,10
      IF(K.EQ.NMX(N))70,60
   60 CONTINUE
C     LOOK FOR COMMA AFTER STATEMENT NUMBER
      IF(K.EQ.COM)75,80
   70 K3=K3+1
      L=2
      GO TO 200
   75 K3=K3+1
      L=3
      GO TO 200
   80 PRINT 500,K3,K4,K5,KESN $ NERR=4
      GO TO 400
  500 FORMAT(1H ,*CHARACTER OTHER THAN COMMA AFTER LAST*,
     1* DIGIT OF FORMAT STATEMENT NUMBER*,1H ,3HK3=,I2,2X,
     23HK4=,I2,2X,3HK5=,I2,2X,5HKESN=,I2)
   85 IF(K.EQ.LP)90,95
   90 K3=K3+1
      K5=K5+1
      L=3
      GO TO 200
   95 K4=K4+1
      K3=K3+1
      L=4
      GO TO 200
  100 IF(K.EQ.LP)105,140
  105 K5=K5+1 $ K7=J $ M=1 $ K9=1
  106 K3=K3+1
      L=5
      GO TO 200
  110 IF(K.EQ.RP)115,106
  115 K5=K5-1
      IF(K5.LE.0)120,106
  120 K3=K3+1
      L=6
      GO TO 200
  125 IF(K.EQ.COM)130,135

  130 CALL NVAR(NIS,M,K4,K7,NV)

      NP=NP+1 $ NP1=NP+3
      IF(NDO.EQ.1)131,132
  131 NRHWDO(NRHW,3)=NP
      NRHWDO(NRHW,NP1)=NV
      GO TO 133
  132 NRHODO(NRHO,3)=NP
      NRHODO(NRHO,NP1)=NV
  133 IF(KA.EQ.1)400,134
  134 K3=K3+1
      K4=0
```

```
      L=3
      GO TO 200
  135 IF(K.EQ.BLANK)350,130
  140 IF(K.EQ.COM)145,150
  145 K7=J $ M=1
      GO TO 130
  150 IF(K.EQ.BLANK)155,150
  155 K8=1 $ K7=J $ M=1
      GO TO 130
  160 K3=K3+1
      K4=K4+1 $ L=4
      GO TO 200
  200 CONTINUE
  300 CONTINUE
  350 IF(K9.EQ.1)375,400
  375 K8=1 $ K9=0 $ GO TO 130
  400 RETURN
      END
      SUBROUTINE MCALL(NDO,NRHO,NLHO,NRHW,NLHW,NRHODO,
     1NLHODO,NRHWDO,NLHWDO,KESN,NX,MX,NMX,NERR,NIS)
      DIMENSION NRHODO(50,7),NLHODO(50,7),NRHWDO(50,7),
     1NLHWDO(50,7),NX(10),MX(10),NMX(10),NIS(10)
      DATA BLANK,COM,LP,RP/55B,56B,52B,52B/
      DATA NBLANK/55555555555555558/
      TYPE INTEGER BLANK,COM,RP
      IF(NDO.EQ.1)10,20
   10 NRHW=NRHW+1
      NLHW=NLHW+1
      NRHWDO(NRHW,1)=NRHW
      NRHWDO(NRHW,2)=KESN
      NLHWDO(NLHW,1)=NLHW
      NLHWDO(NLHW,2)=KESN
      GO TO 30
   20 NRHO=NRHO+1
      NLHO=NLHO+1
      NRHODO(NRHO,1)=NRHO
      NRHODO(NRHO,2)=KESN
      NLHODO(NLHO,1)=NLHO
      NLHODO(NLHO,2)=KESN
   30 K3=14 $ K4=K5=K8=0 $ NPR=NP3=0 $ L=1 $ N1=1 $ N2=4
      NMATCH=NLHO
      DO 300 I=2,10
      IF(I.EQ.2)41,42
   41 K6=8
      GO TO 43
   42 K6=1
   43 DO 200 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      GO TO (44,55,65,80,95),L
   44 IF(K.EQ.LP)50,45
   45 K3=K3+1
      L=1
```

```
      GO TO 200
   50 K5=K5+1
      K3=K3+1
      L=2
      GO TO 200
   55 IF(K.EQ.LP)50,60
   60 K4=K4+1
      K3=K3+1
      L=3
      GO TO 200
   65 IF(K.EQ.LP)70,190
   70 K7=J $ M=1 $ K5=K5+1
   75 K3=K3+1
      L=4
      GO TO 200
   80 IF(K.EQ.RP)85,75
   85 K5=K5-1
      IF(K5.GT.1)75,90
   90 K3=K3+1
      L=5
      GO TO 200
   95 IF(K.EQ.COM)100,190
  190 PRINT 500,K3,K4,K5,I,J,KESN
  500 FORMAT(1H ,*NO COMMA FOLLOWING RP WITHIN PARAMETER LIST*,
     11H ,5HK3=,I2,2X,5HK4=, I2,2X,5HK5=,I2,2X,
     22HI=,I1,2X,2HJ=, I2,2X,5HKESN=,I2)
      NERR=5
      GO TO 400
  100 CALL NVAR(NIS,M,K4,K7,NV)
  101 IF(NLHODO(NMATCH,3).EQ.NBLANK)105,110
  110 IF(NLHODO(NMATCH,N2).EQ.NV)130,103
  103 N1=N1+1 $ N2=N1+3
      IF(N1.GT.NLHODO(NMATCH,3))105,110
  105 NMATCH=NMATCH-1 $ N1=1 $ N2=4
      IF(NMATCH.GT.0)101,115
  115 N1=1 $ N2=4
      NMATCH=NLHW
  117 IF(NLHWDO(NMATCH,3).EQ.NBLANK)120,122
  122 IF(NLHWDO(NMATCH,N2).EQ.NV)130,118
  118 N1=N1+1 $ N2=N1+3
      IF(N1.GT.NLHWDO(NMATCH,3))120,122
  120 NMATCH=NMATCH-1 $ N1=1 $ N2=4
      IF(NMATCH.GT.0)117,125
  123 IF(K8.EQ.1)400,124
  124 K3=K3+1
      L=3 $ NMATCH=NLHO $ N1=1 $ N2=4
      GO TO 200
C     VARIABLE IN PARAMETER LIST IS A LEFT-HAND
C     VARIABLE OUTSIDE OF A DO
  125 NPS=NPS+1
      NP2=NPS+3
      K4=0
      IF(NDO.EQ.1)127,126
  126 NLHODO(NLHO,3)=NPS
      NLHODO(NLHO,NP2)=NV
```

```
      GO TO 123
  127 NLHWDO(NLHW,3)=NPS
      NLHWDO(NLHW,NP2)=NV
      GO TO 123
C     VARIABLE IN PARAMETER LIST IS A RIGHT-HAND VARIABLE
C     WITHIN A DO
  130 NPR=NPR+1
      NP1=NPR+3
      K4=0
      IF(NDO.EQ.1)133,132
  132 NRHODO(NRHO,3)=NPR
      NRHODO(NRHO,NP1)=NV
      GO TO 123
  133 NRHWDO(NRHW,3)=NPR
      NRHWDO(NRHW,NP1)=NV
      GO TO 123
  150 IF(K.EQ.COM)155,160
  155 K7=J $ M=1
      GO TO 100
  160 IF(K.EQ.RP)170,60
  170 K8=1 $ K7=J $ M=1
      GO TO 100
  200 CONTINUE
  300 CONTINUE
  400 RETURN
      END

      SUBROUTINE MAE(NRHO,NLHO,NRHW,NLHW,NRHODO,NLHODO,
     1NRHWDO,NLHWDO,NDO,NX,MX,NMX,NERR,NIS,KESN)
      DIMENSION NRHODO(50,7),NLHODO(50,7),NRHWDO(50,7),
     1NLHWDO(50,7),NX(10),MX(10),NMX(10),NIS(10)
      DATA BLANK,MUL,DIV,ADD,SUB,PER,CON,LP,RP,NEQ/
     155B,47B,50B,45B,46B,57B,56B,51B,52B,54B/
      TYPE INTEGER BLANK,DIV,ADD,SUB,PER,CON,RP
      IF(NDO.EQ.1)10,20
   10 NRHW=NRHW+1
      NLHW=NLHW+1
      NRHWDO(NRHW,1)=NRHW
      NRHWDO(NRHW,2)=KESN
      NLHWDO(NLHW,1)=NLHW
      NLHWDO(NLHW,2)=KESN
      GO TO 30
   20 NRHO=NRHO+1
      NLHO=NLHO+1
      NRHODO(NRHO,1)=NRHO
      NRHODO(NRHO,2)=KESN
      NLHODO(NLHO,1)=NLHO
      NLHODO(NLHO,2)=KESN
   30 K3=7 $ K4=NP=0 $ J=1 $ NI=2 $ K5=0
      DO 35 I=2,7
      K4=K4+1
      K3=K3+1
      J=J+1
      K=NIS(2)
      K=K.AND.MX(J)
```

```
      K=K/MX(J)
      K=K.AND.MX(8)
      IF(K.EQ.NEQ)40,32
   32 IF(K.EQ.LP)33,35
   33 NK4=K4 $ NJ=J  $ K5=1
   35 CONTINUE
      PRINT 500,K3,K4,K6,KESN
  500 FORMAT(1H ,*AE SEARCH FAILED TO TURN UP EQUALITY*,
     1* SIGN*,1H ,3HK3=,I2,2X,3HK4=,I2,2X,3HK6=, I2,2X,
     25HKESN=, I2)
      NERR=6
      GO TO 400
   40 IF(K5)41,41,42
   41 NK4=K4 $ NJ=J
      GO TO 43
   42 NK4=NK4 $ NJ=NJ
   43 CALL NVAR(NIS,NI,NK4,NJ,NV)
C     LEFT-HAND VARIABLE RETURNS AND WILL NOW BE STORED
      IF(NDO.EQ.1)50,45
   45 NLHODO(NLHO,3)=1
      NLHODO(NLHO,4)=NV
      GO TO 55
   50 NLHWDO(NLHW,3)=1
      NLHWDO(NLHW,4)=NV
   55 NP=0 $ L=1 $ K4=0
      K3=K3+1
      DO 300 I=2,10
      IF(I.EQ.2)60,65
   60 K6=J+1
      GO TO 70
   65 K6=1
   70 DO 200 J=K6,6
      K=NIS(I)
      K=K.AND.MX(J)
      K=K/MX(J)
      K=K.AND.MX(8)
      GO TO (75,115),L
C     LOOK FOR DECIMAL DIGIT
   75 DO 80 N=1,10
      IF(K.EQ.NMX(N))85,80
   80 CONTINUE
      GO TO 90
   85 K3=K3+1
      L=1
      GO TO 200
C     LOOK FOR SPECIAL CHARACTER
   90 IF(K.EQ.MUL)85,92
   92 IF(K.EQ.DIV)85,94
   94 IF(K.EQ.ADD)85,96
   96 IF(K.EQ.SUB)85,98
   98 IF(K.EQ.PER)85,100
  100 IF(K.EQ.COM)85,102
  102 IF(K.EQ.LP)85,104
  104 IF(K.EQ.RP)85,106
  106 IF(K.EQ.BLANK)400,110
```

```
  110 K4=K4+1
      K3=K3+1
      L=2
      GO TO 200
C     LOOK FOR TERMINATING SPECIAL CHARACTER
  115 IF(K.EQ.MUL)130,117
  117 IF(K.EQ.DIV)130,119
  119 IF(K.EQ.ADD)130,121
  121 IF(K.EQ.SUB)130,123
  123 IF(K.EQ.LP)130,125
  125 IF(K.EQ.RP)130,127
  127 IF(K.EQ.BLANK)130,110
  130 CALL NVAR(NIS,I,K4,J,NV)
C     A RIGHT-HAND VARIABLE IS RETURNED AND WILL NOW BE STORED
      NP=NP+1
      NP1=NP+3
      K4=0
      IF(NDO.EQ.1)140,135
  135 NRHODO(NRHO,3)=NP
      NRHODO(NRHO,NP1)=NV
      GO TO 145
  140 NRHWDO(NRHW,3)=NP
      NRHWDO(NRHW,NP1)=NV
  145 IF(K.EQ.BLANK)400,85
  200 CONTINUE
  300 CONTINUE
  400 RETURN
      END
```

APPENDIX B

An Example of the Recognition Process

This Appendix will illustrate the theory of operation of the FORTRAN parallel task recognizer. This will be accomplished by applying the recognition techniques to a sample source program. A listing of the sample program follows. The numbers to the left of the executable statements are the numbers assigned by the recognizer during Phase I.

```
      C     THIS IS A TEST PROGRAM DESIGNED TO CHECK PPS
            DIMENSION A1(10),A2(10),A3(10)
            INTEGER A1,A2,ABC,A2,X1,X2,B,C,D
 1          READ 100, (A1(I),I=1,10),B,C,D
 2          READ 100, (A2(I),I=1,10),NS,NST,NSTU
 3          DO 10 I=1,10
 4          IF(A1(I)-A2(I))20,30,40
 5       20 X1=(A1(I))**(B-C)
 6          X2=D**(B/C)
 7          A3(I)=X1*X2
 8       10 CONTINUE
      C     THIS IS A TEST COMMENT
 9       30 PRINT 200,B,C,D
10       40 CALL ALPHA(A1,A2,ABC,B4,B5)
11          PRINT 3057,X1,X2,(A3(I),I=1,10)
12          CALL BETA(X1,X2,A3,B5)
13          IF(B4-B5)50,50,60
14       50 READ 315,E,F,G,H
15          X3=(E*F)+(G-H)
16          X4=B6+G
17          X5=X3-X4
18          X6=(B4+B5)*X5
19       60 PRINT 4,X3,X4,X5
20          PRINT 52,(A1(I),I=1,10),ABC,C,(A3(I),I=1,10)
          100 FORMAT(10I2,3I3)
          200 FORMAT(1HO,8  B  C  D**,/,3I3)
         3057 FORMAT(1H ,2I3,10F7.1)
          315 FORMAT(4F7.4)
            4 FORMAT(3F7.4)
           52 FORMAT(12I3,10F7.1)
21          END
```

A listing follows of the tables generated during Phase I, the connectivity matrices generated during Phase II, and the precedence partitions accomplished in Phase III.

NTABIF

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
1	4	20	30	40	5	9	10

NTABDO

(1)	(2)	(3)	(4)
1	3	1	8

NLHODO

(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	1	4	A1	B	C	D
2	2	4	A2	NS	NST	NSTU
3	10	3	ABC	B4	B5	
4	12	1	B6			
5	14	4	E	F	G	H
6	15	1	X3			
7	16	1	X4			
8	17	1	X5			
9	18	1	X6			

NRHODO

(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	9	3	B	C	D	
2	10	2	A1	A2		
3	11	3	X1	X2	A3	
4	12	3	X1	X2	A3	
5	13	2	B4	B5		
6	15	4	E	F	G	H
7	16	2	B6	G		
8	17	2	X3	X4		
9	18	3	B4	B5	X5	
10	19	3	X3	X4	X5	
11	20	4	A1	ABC	C	A3

NLHWDO

(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	5	1	X1			
2	6	1	X2			
3	7	1	A3			

NRHWDO

(1)	(2)	(3)	(4)	(5)	(6)	(7)
1	4	2	A1	A2		
2	5	4	A1	I	B	C
3	6	3	D	B	C	
4	7	2	X1	X2		

THE BLOCKS CONTAINING THE STATEMENT NUMBERS OF THE INPUT PROGRAM WHICH CAN BE EXECUTED IN PARALLEL ARE GIVEN BELOW.

(1,2)
(3)
(9,10,11,12)
(13)
(14)
(15,16)
(17)
(18,19,20)

NC

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1
2	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

NC1

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 24. Connectivity Matrices for the Example Program.

B I B L I O G R A P H Y

[1] Richards, R. K. "Electronic Digital Systems," John Wiley & Sons, Inc., 267.

[2] Radin, George and Rogoway, Paul. "NPL: Highlights of a New Programming Language," Communications of the ACM, Vol. 8 (1), 9-17, (Jan. 1965).

[3] Bernstein, A. J. "Analysis of Programs for Parallel Processing," IEEE Transactions on Electronic Computers, Vol. EC-15 (5), 757-763, (Oct. 1966).

[4] Abel, Norma E.; Budnik, Paul P.; Kuck, David J.; Muraoka, Yoichi; Northcote, Robert S.; and Wilhelmson, Robert B. "TRANQUIL: A Language for an Array Processing Computer," Proc. Spring Joint Computer Conference, 57-68, (1969).

[5] Dennis, J. B. "Programming Generality, Parallelism and Computer Architecture," Proc. IFIPS Congress 68, C1-C7.

[6] Sutherland, W. R. "On-Line Graphical Specification of Computer Procedures," Technical Report No. 405, Lincoln Laboratory, M. I. T., 1966.

[7] Tesler, L. G. and Enea, H. J. "A Language Design for Concurrent Processes," Proc. Spring Joint Computer Conference, Vol. 32, 403-408, (1968).

[8] Conway, Melvin E. "A Multiprocessor System Design," Proc. Fall Joint Computer Conference, Vol. 23, 139-146, (1963).

[9] Gosden, J. A. "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers," Proc. Fall Joint Computer Conference, Vol. 29, 651-660, (1966).

[10] Ramamoorthy, C. V. "A Dynamic Look Ahead & Program Segmentation System for Multi-Programmed Computers," Proc. National Meeting ACM, 1966.

[11] Martin, D. E. and Estrin, G. "Models of Computational Systems," IEEE Transactions, Vol. EC-16 (1), (Feb. 1967).

[12] Opler, Ascher. "Procedure-Oriented Statements to Facilitate Parallel Processing," Communications of the ACM, Vol. 8 (5), 306-307, (May 1965).

[13] Dennis, Jack B. and Van Horn, Earl C. "Programming Semantics for Multi-Programmed Computations," Communications of the ACM, Vol. 9 (3), 143-155, (Mar. 1966).

[14] Djkstra, E. W. "Solution of a Problem in Concurrent Programming Control," Communications of the ACM, Vol. 8 (9), 569, (Sept. 1965).

[15] Knuth, Donald. "Additional Comments on a Problem in Concurrent Programming Control," Communications of the ACM, Vol. 9 (5), 321-322, (May 1966).

[16] Coffman, E. G. and Munts, R. R. "Models of Pure Time Sharing Disciplines for Research Allocation," Proc. 1969 National ACM Conference.

[17] Fisher, D. A. "Program Analysis for Multiprocessing," Burroughs Corporation, (May 1967).

[18] Ramamoorthy, C. V. "Analysis of Graphs by Connectivity Considerations," Journal of the ACM, Vol. 13 (2), 211-222, (Apr. 1966).

[19] Ramamoorthy, C. V. "A Structural Theory of Machine Diagnosis," Proc. Spring Joint Computer Conference, 743-756, (1967).

[20] Hellerman, H. "Parallel Processing of Algebraic Expressions," IEEE Transactions on Electronic Computers, Vol. EC-15 (1), (Feb. 1966).

[21] Stone, Harold S. "One-Pass Compilation of Arithmetic Expressions for a Parallel Processor," Communications of the ACM, Vol. 10 (4), 220-223, (Apr. 1967).

[22] Squire, J. S. "A Translation Algorithm for a Multiprocessor Computer," Proc. 18th ACM National Conference, 1963.

[23] Baer, J. L. and Bovet, D. P. "Compilation of Arithmetic Expressions for Parallel Computation," Proc. IFIPS, '68, B4-B10, (1968).

[24] Knuth, Don. "The Art of Computer Programming, Vol. 1, Fundamental Algorithms," Addison-Wesley, 316.

[25] Russell, E. C. and Estrin, G. "Measurement Based Automatic Analysis of FORTRAN Programs," Proc. Spring Joint Computer Conference, 1969.