DECEMBER, 1965

REPORT ESL-R-253
M.I.T. PROJECT DSR 9948
NASA Research Grant NsG-496(Part)

CADD: ON-LINE SYNTHESIS OF LOGIC CIRCUITS

Michael L. Dertouzos
Paul J. Santos, Jr.

Electronic Systems Laboratory

MASSACHUSETTS INSTITUTE OF TECHNOLOGY, CAMBRIDGE, MASSACHUSETTS 02139

Department of Electrical Engineering

December, 1965 Report ESL-R-253 Copy 32

CADD: ON-LINE SYNTHESIS OF LOGIC CIRCUITS

by

M. L. Dertouzos and P.J. Santos, Jr.

The preparation and publication of this report, including the research on which it is based, was sponsored under a grant to the Electronic Systems Laboratory, Massachusetts Institute of Technology, Project DSR No. 9948. This grant is being administered as part of the National Aeronautics and Space Administration Research Grant No. NsG-496 (Part). This report is published for information purposes only and does not represent recommendations or conclusions of the sponsoring agency. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Electronic Systems Laboratory
Department of Electrical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

N66-21659

The central aim of logical design is the synthesis of any given switching function in terms of given sets of elementary building blocks, for the optimization of some performance index in the presence of constraints. Although the present "state of the art" yields algorithmic methods for the solution of certain specific instances of the problem (such as minimization of building-block inputs with a two-level, AND-OR realization), no fully algorithmic method exists for the solution of the more general problem.

The system described in this paper gives solutions to the general problem by use of an online process (using Project MAC at M.I.T.), where the machine accomplishes those computational tasks which can be algorithmically specified, and where the user provides those decisions which he is better qualified to make. The machine portion of the system is based on a set of heuristic procedures which guarantee convergence of the process and give better results than conventional, sub-optimal brute-force techniques. The machine thus behaves in an "intelligent" fashion using successive local-optimization procedures and does not depend on impractical (and usually impossible) exhaustive searches through all possible solutions. When coupled with the flexible human decision process, these procedures give results of practical significance.

Author

CONTENTS

CONTENTS (Cont.)

LIST OF FIGURES

be used in this case. This is an instance where fan-in limitations prevent the use of the standard minimization algorithms. Moreover, synthesis limitations are present when there exist no standard synthesis method that can handle a given set of logic gates. An example of such a case is synthesis restricted to EXCLUSIVE-OR and AND gates, even when these gates have no fan-in restrictions.

B. METHOD OF SOLUTION

1. General Concepts

The method of solution described in this report is basically heuristic; a fully algorithmic and rigorous treatment of this problem does not exist presently and its future development seems unlikely in view of our current knowledge. Fundamental to this heuristic approach is the use of a digital computer on an interactive basis with the human designer. Another aspect of this approach which is a basis for its heuristic structure is a dependence on local, rather than global, optimization algorithms. To illustrate and clarify the often abused term "heuristic", the following description of system operation is given. A discussion of convergence of the method is given in Chapter II, Section D.

The designer, located at a remote terminal of a digital computer, communicates with it typically via a keyboard, and perhaps through a graphical display. A computer program governs the man-machine interaction, and provides the necessary calculating power. The designer provides the program with the necessary data about the Boolean function to be synthesized and the set of blocks which are to be used in the realization of that function. The man-machine combination then carries out a recursive decomposition which operates first on the given function and then on each subfunction into which the given function is decomposed. Decomposition is thus continued until each non-decomposed subfunction becomes either an input variable or a constant. A simple example of such a decomposition process is shown in Fig. 1.1 where the allowable gates are two-input ANDs and ORs. Each step of the process corresponds to a subfigure, and the subfigures are numbered in increasing order of complexity of decomposition. Note the multiple use of function BD. This example

CHAPTER I

INTRODUCTION

A. DESCRIPTION OF THE PROBLEM

The synthesis of a given switching function using logical building blocks (gates) is a task which has traditionally been approached with emphasis on some optimality criterion. At present, there exist methods of synthesis such as two-level AND-OR, NOR-NOR, and NAND-NAND, which guarantee an optimal realization of any switching function. Common features of such methods are:

a) a restriction on the type and method of interconnection of building blocks, and

b) a lack of restriction on the number of inputs to these blocks.

For example, two-level AND-OR synthesis of arbitrary (four-variable) switching functions may require the use of AND gates with two to four inputs and an OR gate with two to eight inputs.

It is a matter of definition that any set of building blocks which is "logically complete"* can be used exclusively in the synthesis of any given switching function. It is desirable, then, to have a general method which synthesizes any given Boolean function using any given set of logically complete blocks. Such a general synthesis approach is the objective set forth in this report.

Before outlining the foregoing generalized method, it seems appropriate to give a clearer and more specific picture of the limitations which generally confront the user of conventional optimal techniques. In the case of AND-OR synthesis, for example, a perfectly valid, logically complete set might be a two-input AND gate and a two-input OR gate. Since the well-known optimal technique requires either or both blocks to have more than two inputs, it cannot

* At no point in the method to be discussed is the logical completeness of a given set questioned. The proof of completeness is, in general, a difficult task and has therefore been assumed to be already shown.

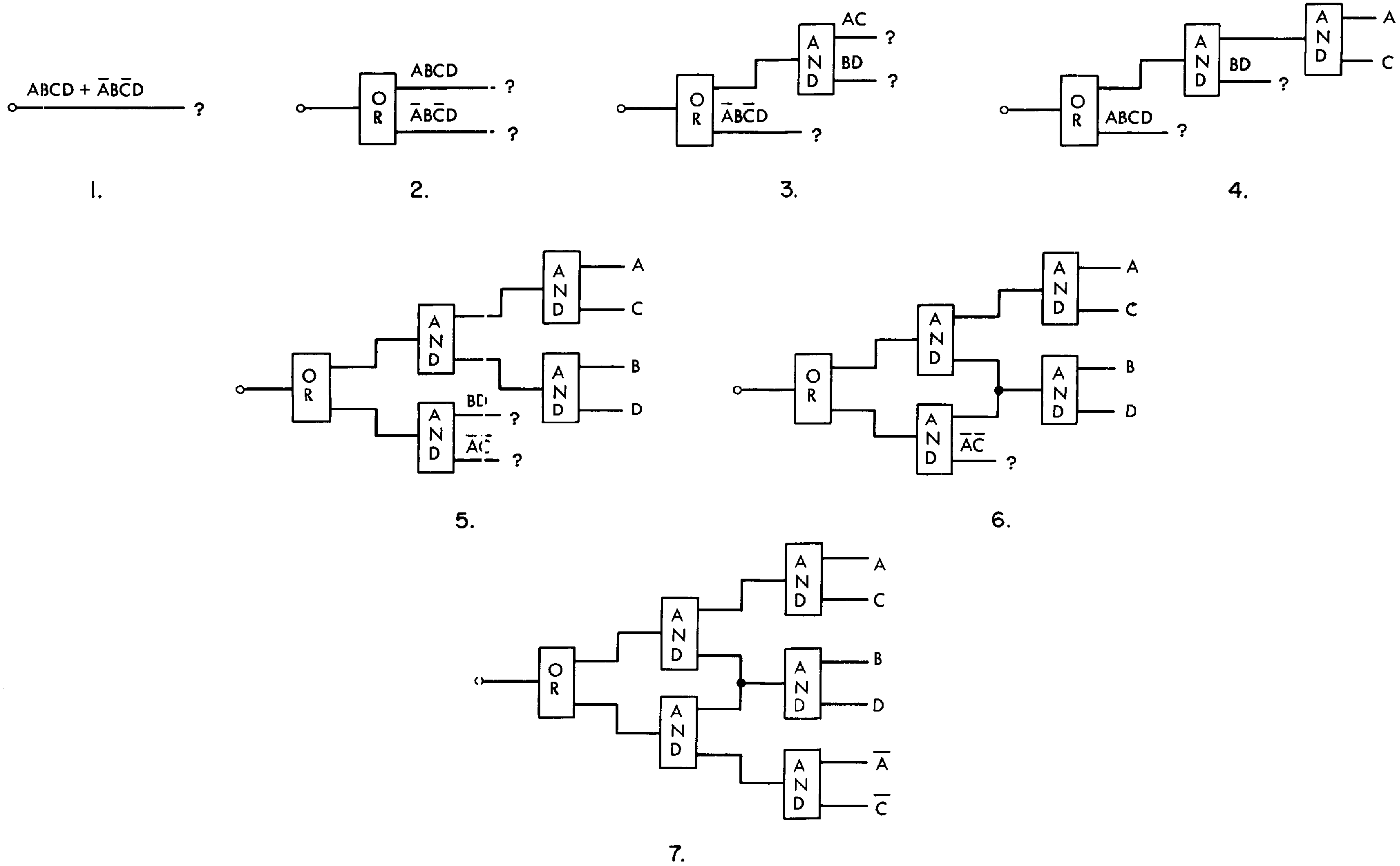


Fig. 1.1 Illustration of Recursive Decomposition Process

is given to illustrate principles of operation and does not represent a CADD-generated solution.

As a general rule, the system of programs which implement the method and control the interaction do not allow the operator to make a decision which is inconsistent with the current state of the decomposition, both by checking each decision and by limiting his choices. The final result of the synthesis is a block diagram which contains only building blocks belonging to the originally specified complete set, arranged in some arbitrary tree structure and realizing the given function.

A completely mechanized heuristic approach to a problem of this magnitude would involve a large amount of programming to account for all possible circumstances, and would be, moreover, inflexible to change. By using an interactive system, however, all or part of the decision mechanism can be delegated to the human designer. Thus, programming time and the amount of computer memory occupied by the programs are reduced. Furthermore, the interactive system is flexible and permits an easier development of heuristics to meet the many unexpected situations which inevitably arise.

Because of the foregoing, the system started out in a highly experimental form, with most of the decision-making burden assigned to the human. Gradually, however, modifications were introduced to balance the apportionment of decision-making and computing. The modifications were derived from the observation of common situations and patterns which were amenable to algorithmic solution.

Global optimization usually involves an unmanageable growth of data space, and severely limits the size of the problem which can be attacked. Local optimization, on the other hand, although it generally gives results which are not globally optimal, has the advantage of a reasonably bounded data space and is capable of handling larger problems with an attendant increase in computer time only.

This report concerns two basic programs referred to as CADD-1 and CADD-2 which are the original and modified versions of the method under discussion.

2. CADD-1 and CADD-2

CADD-1 refers to the system completed in June, 1965. This system is free of program errors and is capable of attacking the type of problem described in the foregoing. It is subject, however, to several deficiencies which can be separated into three areas.

The first of these deficiencies concerns the inclusion of the function library* (q.v.) and its attendant need for folding, rotating, etc. In retrospect, it was found that the function library contributed little to the synthesis process and used a large amount of program space.

The other two deficiencies are on the implementation level and concern the speed and amount of man-machine interaction. CADD-1 uses a typewriter for all interaction, and hence a large amount of real (human) time is consumed in the typing of decomposition tables and block diagrams for the information of the designer. Also, since CADD-1 is a highly experimental system, a large amount of interaction replaces unknown algorithmic tasks. Experience with CADD-1 eventually made much of this interaction unnecessary.

The foregoing deficiencies make CADD-1 a rather slow system in terms of real time (six hours of interaction may be required for a difficult six-variable function), even though computer time usage is small (about two minutes for the same six-variable function).

To overcome these shortcomings of CADD-1, a new system, CADD-2, was created. Major modifications consisted of removing the function library and its associated machinery, using a graphical display rather than a typewriter to speed up the rate of interaction, and eliminating certain areas of interaction. The graphical display used in CADD-2 is the Electronic Systems Laboratory Display Console, and the digital computer used for both CADD-1 and CADD-2 is the Project MAC time sharing system using a modified IBM 7094 processor.

C. BASIS FOR EVALUATION OF RESULTS

Because the generalized synthesis procedure deals generally with problems to which there are no other known methods of solution

*A library composed of functions generated by permuting and negating inputs of all the available building blocks.

except for so-called "brute-force" methods (to be discussed), it becomes difficult to judge the "goodness" of a particular circuit realization developed by CADD-1 or CADD-2. In a sense, it is "good" that even a single solution has been achieved. Furthermore, how is one to judge the "goodness" of a certain block-diagram configuration, which was purposely generated in that form by the user for reasons of his own? Such a configuration may be better than another configuration which perhaps contains fewer blocks but fails to satisfy criteria of greater importance to the user.

Since there exist conflicting or unknown measures of "goodness," the only comparison used here is based on the relative number of building blocks used by the generalized versus the (a priori known) "brute-force" techniques.

The "brute-force" technique discussed in the foregoing consists of the following steps

1. Since the given set of building blocks is logically complete, use it to generate {AND, OR}, {NOR} or {NAND}. That is, construct each member of this new set from members of the given set and from the constants 1 and 0.

2. Carry out the classical two-level minimal AND-OR, OR-AND, NOR-NOR, or NAND-NAND synthesis.*1

3. Substitute members of the classical realization by members of the given set in accordance with step 1, above. Observe that step 1 may be invoked several times, such as, for example, when a particular type of block required by the classical realization contains a different number of inputs than a block already generated. Such a case would be the building of a four-input AND gate from two-input AND gates.

4. Retain the realization generated from step 3 above which uses the least number of building blocks belonging to given sets. This realization will be then compared to realizations obtained using generalized CADD techniques.

It may be true that this comparison is somewhat arbitrary and unfair. Nevertheless, it is the only known method that can be consistently used, since it is independent of the type of logic blocks to be used and of the function to be synthesized.

* Superscripts refer to numbered items in the Bibliography.

CHAPTER II

DESCRIPTION OF THE SYNTHESIS METHOD

A. OUTLINE

A flow diagram which outlines the generalized synthesis method is given in Fig. 2.1. Operation can be considered in three phases: Phase I, where the program accepts as input the function to be synthesized and the building blocks to be used, and, in the case of CADD-1, generates from each building block a "library" of functions to be used in Phase III; Phase II, where the program provides the mechanism for associating one of the given building blocks with the current function to be realized;[*] and Phase II, where the mechanism is provided for properly decomposing the current function, under the restriction of the given building blocks, into subfunctions. These subfunctions are either constants, variables, negated variables, building-block functions of variables from the library generated in Phase I (CADD-1 only), functions already realized, the fan-out of which has not yet been exceeded (CADD-2 only), or functions which in turn have to be decomposed later.

Phase I is described more fully in Section B of this chapter, Phases II and III in Section C, and particular aspects of Phase III in Section D.

B. PRELIMINARY PROCESSING

Prior to beginning the actual synthesis, a certain amount of processing of the given set of building blocks must take place. This preprocessing consists of the creation of subelements in the building block directory and, in the case of CADD-1, the generation of a library of functions, for each building block, to be used in a manner described in Section C. For each given block, the preprocessing operation is as follows:

[*] The current function may be the result of successive decompositions of the original function, or the original function itself.

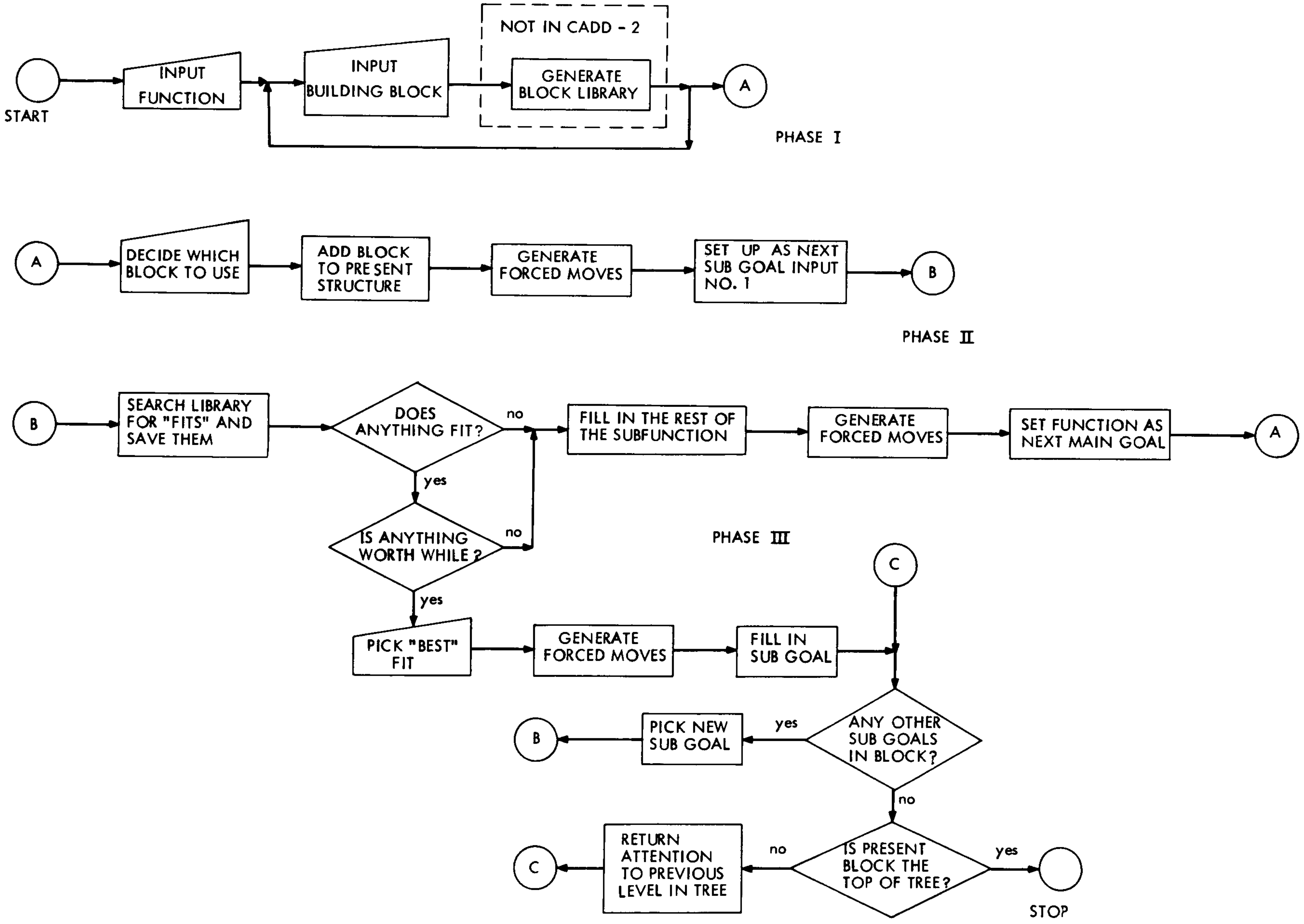

START
INPUT FUNCTION
INPUT BUILDING BLOCK
NOT IN CADD - 2
GENERATE BLOCK LIBRARY
A
PHASE I
A
DECIDE WHICH BLOCK TO USE
ADD BLOCK TO PRESENT STRUCTURE
GENERATE FORCED MOVES
SET UP AS NEXT SUB GOAL INPUT NO. 1
B
PHASE II
B
SEARCH LIBRARY FOR "FITS" AND SAVE THEM
DOES ANYTHING FIT?
no
yes
IS ANYTHING WORTH WHILE ?
no
yes
FILL IN THE REST OF THE SUBFUNCTION
GENERATE FORCED MOVES
SET FUNCTION AS NEXT MAIN GOAL
A
PHASE III
PICK "BEST" FIT
GENERATE FORCED MOVES
FILL IN SUB GOAL
C
ANY OTHER SUB GOALS IN BLOCK?
yes
PICK NEW SUB GOAL
B
no
IS PRESENT BLOCK THE TOP OF TREE?
yes
STOP
no
RETURN ATTENTION TO PREVIOUS LEVEL IN TREE
C

Fig. 2.1 Outline of Synthesis Method

1. Create a new subelement in the block directory which contains all the pertinent information about the block, such as its truth table, the number of inputs, its name, and a usable specification of its function. This function specification takes the form of two lists which indicate for what combinations of input values the block generates ZEROS and ONES, respectively. To generate these lists, first roughly list all possible input combinations which yield, say, a ZERO, and then refine this list in a manner similar to the Quine-McCluskey[2] procedure. Thus, any element of the list which is independent of a particular input, carries a DON'T CARE entry under that input.

2.* Detect all the sets of inputs about which the function is symmetric, since the function is invariant under permutations of these inputs. This is done by choosing the smaller of the two (zero or one) function specification lists and considering all possible interchanges of inputs within the lists in order to see if, indeed, the function remains invariant under that particular interchange of inputs. Clearly, inputs which are symmetric to the same input are symmetric to each other.

3.* Generate all unique permutations of the inputs by using the detected symmetries to eliminate hidden duplicates.

4.* Initialize a library list and append to it items consisting of a truth-table specification of a block function and an indication of the permutations and negations of the input variables which generate this function. For each permutation, and for each possible combination of negations, an item is added to the list if its truth table does not duplicate that of an item already in the list. When the creation of the library list is finished, it contains all possible functions which can be generated by permuting and negating the inputs to the given block.

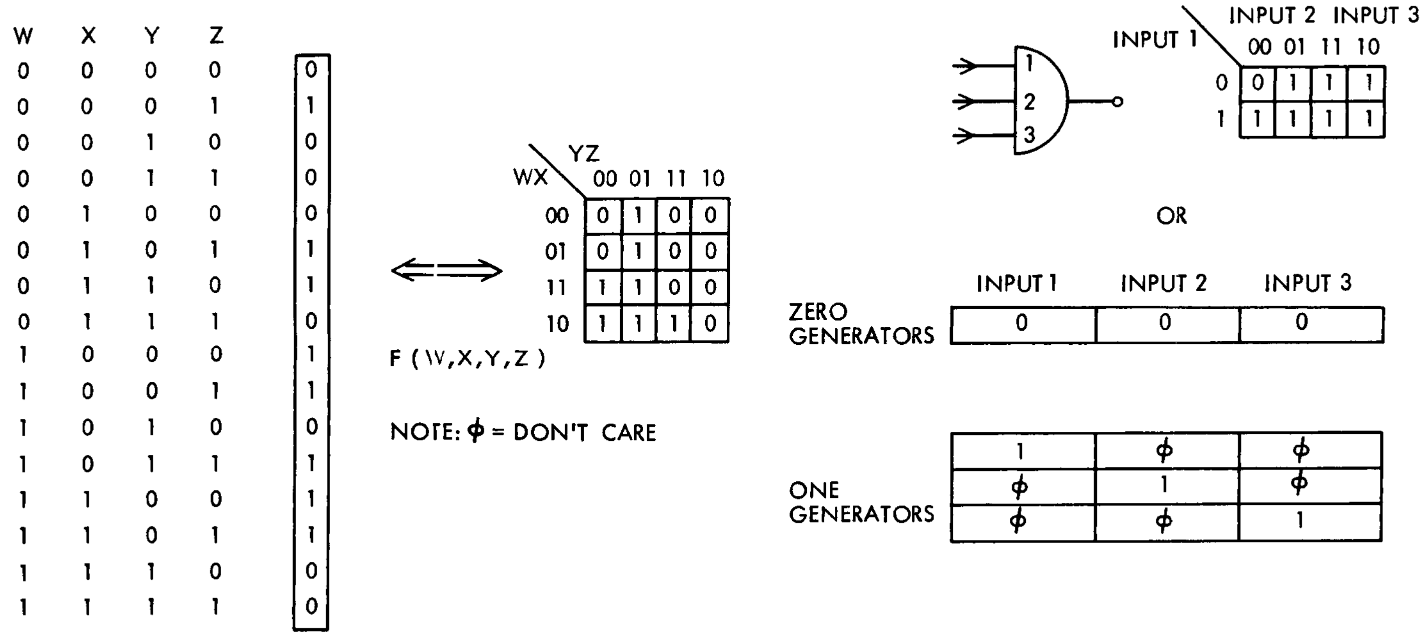
C. DECOMPOSITION

Classical synthesis methods generally build up the circuit realization by successive combinations of simple functions starting from the input variables until the desired output function is produced. The synthesis method presented here does the opposite, i.e., it works from the output function back toward the input variables, decomposing each function into several "simpler" functions which may, in turn, have immediate realizations or may need to be further decomposed.

* Sections 2 through 4 apply to CADD-1 only.

Before proceeding with a detailed description of the process, two conventions used for representing an n-variable Boolean function $F(x_1, \ldots x_n)$ will be explained. The first convention consists of entering in a 2^n-element array the values of F in an ordering which corresponds to the binary natural code formed by $x_1, \ldots x_n$. Thus, for example, the value F(W, X, Y, Z) = 1 at W = 0, X = 1, Y = 0, Z = 1 is entered in the sixth element of the array since 0101 is the sixth binary number counting zero. The remaining values of F(W, X, Y, Z) are similarly entered as shown in the example of Fig. 2.2a. The second convention involves two sets of n-element arrays, corresponding to the minimum sum of products and to the minimum product of sums. Each of these products (sums) is placed in an n-element array by entering ONES (ZEROS) for those literals of the product (sum) which are present and entering DON'T CARES for the remaining literals. For example, a three-input OR is represented in this convention as shown in Fig. 2.2b.

Returning to the description of decomposition, let us assume that we are given a function (using the first convention described above) which is to be realized. It is desired that this function be realized at the output of one of the given building blocks. First, a block is chosen, in a manner to be described. The problem is, then, to find a set of subfunctions associated one-to-one with the inputs of the block, and satisfying the following condition. for each combination of the pertinent input variables, these subfunctions give a set of values which when applied to the block produce the correct output value. Each subfunction is dependent on the same variables as the original function or some subset of those variables. For example, suppose the function to be realized has a ZERO value for the fourth element and the block realizing this function is a two-input OR. Then subfunctions F and G, corresponding to inputs 1 and 2 respectively, must both have a ZERO value for their fourth element, since an OR generates a ZERO only if both inputs are ZERO. The method for solving this problem and assigning subfunctions to inputs is, for the most part, also described below. although certain aspects of the process, such as convergence and the encounter of "dead ends", are discussed in later sections.



W	X	Y	Z	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

⟺

WX \ YZ	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	1	1	0	0
10	1	1	1	0

F (W,X,Y,Z)

NOTE: ϕ = DON'T CARE

INPUT 1 \ INPUT 2 INPUT 3	00	01	11	10
0	0	1	1	1
1	1	1	1	1

OR

	INPUT 1	INPUT 2	INPUT 3
ZERO GENERATORS	0	0	0
ONE GENERATORS	1	ϕ	ϕ
	ϕ	1	ϕ
	ϕ	ϕ	1

(a) Function Represɜntation

(b) Function Specification Representation for a three-input OR

Fig. 2.2 Illustration of Conventions

In selecting a block to realize a given function, the number of ONES and ZEROS of the function are first obtained, and the DON'T CAREs ignored. Then an operator is applied to each block to determine which is the "best" block to use under the circumstances. The operator contains four sections, each of which is given a weight (variable throughout the course of the process) commensurate with the relative importance of that section. The sections are concerned with:

(a) The number of constants (ZEROs and ONEs) appearing in the Building block function specification.

(b) The number of DON'T CAREs appearing in the same specification.

(c) The number of possible ways to generate each of the desired output values.

(d) The number of block inputs.

The operator uses the information about the number of ONEs or ZEROs of the function to weigh the significance of the ONE generators or ZERO generators of the building block function specification, respectively. Operations performed by Sections (a) and (b) are normalized in order to achieve independence from Sections (c) and (d) above. The reason for making the weight adjustable is because the meaning of "best block" changes from one point in the process to another. For example, given two blocks which are identical in "goodness" except that one has more inputs than the other, the one with more inputs is "better" at the beginning of the decomposition because it tends to simplify the problem more rapidly; whereas, toward the end of the decomposition, the one with fewer inputs may be "better".

Once a block has been chosen, it is added to the existing structure of the block diagram and a decomposition table for that block is created. The table is a rectangular array which has a column for each input to the block, a column for the function to be decomposed, and a row for each element of the function array. The total number of rows is 2^n, where n is the number of arguments of the function. It may be possible to fill in certain entries in the decomposition table immediately. For example, if the function has a DON'T CARE in a certain row, then all other entries of this row will be DON'T CAREs Moreover, if the type of block chosen is such that it can generate, say, ZEROs, in only one or a few ways so that certain inputs must have certain values and no others, then every row in which the function has a ZERO must be filled in according to these input restrictions. Figure 2.3a shows how initial restrictions are filled in. Note, in particular, how all rows for which F_0 is ONE contains ZEROs under all subfunctions. since a NOR can generate a ONE only if all inputs are ZERO

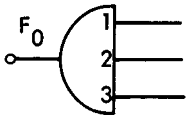
The next step in the process is to try to choose an immediate realization[*] for one of the subfunctions, usually the one corresponding to the first input, F_1. In the example of Fig. 2.3a a search of available immediate realizations is made, rejecting all those which are incompatible with having a ZERO as their fourth and seventh element Those realizations which are compatible are said to "fit"; for instance, the variable X does not fit because it had a ONE in its seventh element, whereas the variable $\overline{Y}$ fits, since it contains ZEROs in its fourth and seventh elements. From all those realizations which fit, the "best" one is chosen and used to fill in the particular column (subfunction) under consideration. The "best" subfunction depends upon the type of block being used. In the case of a NOR gate the subfunction which row by row yields the highest [illegible]correlation[**] with the function is "best", since it generates the greatest number of DON'T CAREs in succeeding subfunctions, i.e., if the output of a NOR is ZERO, then a ONE on any of its inputs allows the other inputs to be arbitrarily assigned. In the case of an AND gate, correlation rather than anticorrelation is the "goodness" criterion, for the same reason, i.e., the maximum generation of DON'T CAREs. Once a goodness criterion has been chosen for a given type of building block the same criterion can be used throughout the synthesis.

In Fig. 2.3b, F_1 has been filled in with the variable $\overline{Y}$ (recall that both variables <u>and</u> their negations are available), since it was found to be the "best" subfunction. Variable $\overline{Y}$ is attached to the first

[*] A constant, variable, negated variable, library function (CADD-1 only), or a previously realized function (CADD-2 only).

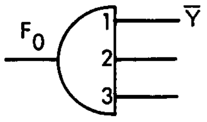
[**] Anticorrelation is the negative of the correlation.

X	Y	Z	F_0	F_1	F_2	F_3
0	0	0	0			
0	0	1	0			
0	1	0	0			
0	1	1	1	0	0	0
1	0	0	0			
1	0	1	ϕ	ϕ	ϕ	ϕ
1	1	0	1	0	0	0
1	1	1	0			



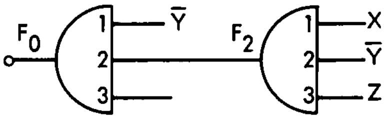
(a)

X	Y	Z	F_0	F_1	F_2	F_3
0	0	0	0	1	ϕ	ϕ
0	0	1	0	1	ϕ	ϕ
0	1	0	0	0		
0	1	1	1	0	0	0
1	0	0	0	1	ϕ	ϕ
1	0	1	ϕ	ϕ	ϕ	ϕ
1	1	0	1	0	0	0
1	1	1	0	0		



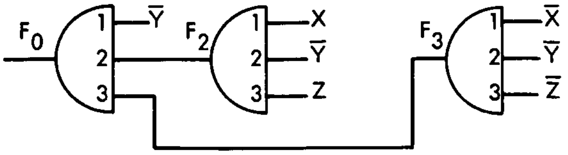
(b)

X	Y	Z	F_0	F_1	F_2	F_3
0	0	0	0	1	ϕ	ϕ
0	0	1	0	1	ϕ	ϕ
0	1	0	0	0	1	ϕ
0	1	1	1	0	0	0
1	0	0	0	1	ϕ	ϕ
1	0	1	ϕ	ϕ	ϕ	ϕ
1	1	0	1	0	0	0
1	1	1	0	0	0	



(c)

X	Y	Z	F_0	F_1	F_2	F_3
0	0	0	0	1	ϕ	ϕ
0	0	1	0	1	ϕ	ϕ
0	1	0	0	0	1	ϕ
0	1	1	1	0	0	0
1	0	0	0	1	ϕ	ϕ
1	0	1	ϕ	ϕ	ϕ	ϕ
1	1	0	1	0	0	0
1	1	1	0	0	0	1



(d)

NOTE:
ϕ = DON'T CARE

Fig. 2.3 Typical Decomposition Step Using Three-Input NOR Gates

input of the NOR in the block diagram, and further entries in the table may now be filled in. In the present example, Fig. 2.3b shows how DON'T CAREs are generated in the remainder of the first, second, and fifth rows due to the ONEs of the first subfunction.

The above procedures are repeated for each of the remaining inputs to the block and each time a subfunction is chosen and filled in, it further restricts the behavior of the remaining inputs. A point may be reached sooner or later when no immediate realization "fits". In this case, remaining blank entries are filled in a manner left to the discretion of the operator, but subject to some of the points to be mentioned in Section D. The resulting subfunction is then treated as a new function to be decomposed. Upon successfully completing this further decomposition, the process moves on to the next input until all inputs have been filled, at which point the original function has been successfully decomposed and attention is then returned to the preceding level of decomposition. In the example of Fig. 2.3, all inputs have immediate realizations; the NOR of X, $\overline{Y}$, and Z for the second input (Fig. 2.3c) and the NOR of $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ for the third input (Fig. 2.3d), so that no further work is necessary and the given function has been decomposed.

To summarize, the decomposition process is accomplished as follows:

1. Take the given function to be synthesized and apply to it the block selection and decomposition techniques illustrated above.

2. Apply these techniques to all generated subfunctions which cannot be immediately realized and iterate until a final realization of the original function is reached.

The path followed by the process in performing Steps 1 and 2 above will form a tree-like structure which is one-to-one with the block diagram representing the current state of the synthesis. The process terminates when all the inputs of the first block in the tree (block diagram) have been filled and realized.

D. CONVERGENCE AND SPECIAL CASES

So far we have not discussed convergence, i.e., the termination of the entire process in a successful realization. The question of whether convergence can be achieved or not will be called the convergence problem. The convergence problem arises whenever a given subfunction is not immediately realizable, but must be further decomposed. In the following subfunction G is said to be a convergent subfunction of function F if either

A. min (ONEs (F), ZEROs (F)) > min (ONEs (G), ZEROs (G))

or

B. DON'T CAREs (G) > DON'T CAREs (F)

If for every generated subfunction one of these convergence criteria is obeyed, then repeated decomposition will yield final functions which are either an input variable, a constant, or a DON'T CARE.

For any arbitrary set of logic gates, it is not certain that this convergence criterion will be satisfied. On the other hand, it is possible to show that convergence can always be satisfied for certain given sets of building blocks. For example, proof of convergence when the given set consists of n-input NOR gates follows:

PROOF: Let the function to be realized have N_0 ZEROs, N_1 ONEs, and N_2 DON'T CAREs. Then each subfunction which is not immediately realizable contains exactly $N_0' = N_1$ ZEROs (due to the ONEs of the function), $N_1' < N_0$ ONEs and $N_2' > N_2$ DON'T CAREs, which assures convergence since $\min (N_0, N_1) \geq \min (N_0', N_1')$ and $N_2' > N_2$. The reason that N_1' is less than N_0 (and therefore by mutual exclusion $N_2' > N_2$) is as follows:

a. Assume that in the worst case, no immediately realizable function fits any of the n inputs. Then assign a single ONE in each row where the function is ZERO, distributing these ONEs as evenly as possible among the inputs.* It follows that

$$N_1' \leq [N_0/n] + 1 \text{ and } N_2' = N_2 + N_0 - N_1'$$

thereby confirming the assertion.

b. Assume one or more immediately realizable subfunctions fit some of the inputs. Since these subfunctions must contain at least one ONE (otherwise they would be trivial and serve no purpose in the decomposition), and since this ONE

* This results in at most [[illegible]] + 1 ONEs per input, where [X] means the integer part of X.

must occupy a row which corresponds to a ZERO in the function, all other unfilled inputs are assigned DON'T CAREs in that row and hence $N_2' \geq N_2 + 1$ and $N_1' \leq N_1 - 1$ for the remaining subfunctions, thus confirming the assertion.

Dually, use of the set of n-input NAND gates renders the method convergent. Likewise, convergence has been shown when the given set consists of minority gates. In addition to the above set of gates, it is expected that convergence can be shown for a number of logically complete sets. No special effort was spent in attempting to show convergence for other sets of gates, since primary attention was directed on the development of the method.

In explaining the generalized synthesis method of Section C, certain special situations were not discussed in order that the basic ideas be made as clear as possible. These special situations can be grouped into cases requiring "folding", permutation of variables, and "backing up", the first two being present in CADD-1, and last being present in both. The remaining discussion in this section, with the exception of the "backing up" issue, applied mainly to CADD-1.

In the example of Fig. 2.3 the function to be realized depends on three variables. Furthermore, the only building block in the function library has three inputs. Hence, both function vectors (linear arrays) are of equal length. The natural question arises as to what policy should be taken if the length of the function vector of the function to be realized were unequal to that of any or all of the immediately realizable functions arising from the building blocks in the function library. The answer to this question depends on the relevant circumstances outlined below:

1. The number of variables, on which the subfunction to be realized depends, is less than the number of inputs of <u>any</u> of the blocks in the library. Consequently, the function vector of each block is longer than the vector of the function to be realized. Therefore, of all available immediately realizable subfunctions, only variables and constants can be used.

2. The number of variables, n, on which the function to be realized depends, is greater than or equal to the number of inputs m of some or all of the building blocks. Those blocks for which m = n are treated as before, that is they are tested for "fitting" and "goodness" along with the variables, and constants. Blocks

for which $m < n$ can be used, provided that their function vectors are "unfolded"* to accommodate the greater length of the subfunction vector or, conversely, the subfunction is "folded"* to accommodate the lesser length of the building block function vector. Those blocks for which $m > n$ are not used at this decomposition stage.

A subfunction vector can be "folded" about its highest-order variable if the first half of the vector is consistent with the second half in the following sense:

1. Let $2p$ be the length of subfunction F, and p the length of folded subfunction F'.
2. For $i = 0$ to $i = p - 1$:
   - a) If the ith element of F, F_i, is a DON'T CARE or an unfilled entry, $F_i' = F_{p+i}$
   - b) If $F_i = 0$ and $F_{p+i} = 1$, then folding is impossible, and the procedure stops, otherwise element $F_i' = 0$
   - c) If element $F_i = 1$ and $F_{p+i} = 0$ folding is impossible and the procedure stops. Otherwise element $F_i' = 1$.

Conversely, "unfolding" a subfunction consists of doubling the length of its function vector by repeating it.

When decomposing a function of n variables, and attempting to check the fit and "goodness" of an immediately realizable subfunction derived from a block having $m < n$ inputs, the subfunction is repeatedly "unfolded" n-m times and is then treated as any other immediately realizable subfunction. The effect of the unfolding process is to make the subfunction independent of the highest-order n-m variables. Clearly the other subfunctions assume the burden of this dependence. Moreover, the extent to which a given, partially specified subfunction can be folded places a lower bound on the number of inputs a usable block may have. That is, if the maximum folded length of the function vector of a subfunction is greater than that of a block, then that block, when unfolded, will not fit, since otherwise that subfunction could have been further folded.

* The process of folding and unfolding is discussed in the next paragraph.

Since folding, in general, implied independence from one or more variables, all non-immediately realizable subfunctions, before being in turn decomposed, are folded as far as their completely specified entries permit. This technique, which also applied to CADD-2, allows the subfunction to be dependent on as few variables as possible and therefore makes the function easier to decompose. Many times these non-immediately realizable subfunctions can be folded by a judicious assignment of blank entries in the decomposition table. Since folding a subfunction satisfies convergence (a function can only be folded a finite number of times), a good heuristic technique consists of using up as many restrictions* as possible, under one subfunction, provided they do not interfere with the folding of that subfunction. The reason for this approach is that other subfunctions may then receive the benefits of DON'T CAREs in the rows where restrictions have been satisfied. The next most desirable alternative to finding an immediate realization for a subfunction is the ability to fold it, particularly if some restrictions can be absorbed at the same time. Several examples of folding appear in Fig. 2.4 with Fig. 2.4a pertaining to CADD-1 only and with Figs. 2.4b and 2.4c pertaining to both CADD-1 and CADD-2.

Folding was explained in terms of the highest-order variable since it is easier to see physically than folding about some other variable. Clearly, it can be extended to folding about any variable through an available mechanism. The number of permutations that are tried at each level of the decomposition is arbitrary, and will affect the local optimality of the solution, rather than the actual finding of a solution It should be noted that for a given n-variable function and an m-input block at most (n-m)! permutations of the variables (starting with the highest-order one) need be tried since the least significant m-variables, corresponding to the m-inputs of the block, have already been exhaustively permuted in the function library. Examples of this permuting process are shown in Fig. 2.4.

* Restrictions are any situations which limit decomposition operations such as folding, fitting, etc.

W	X	Y	Z	F1	F2	F3	F4
0	0	0	0	0	1	ϕ	ϕ
0	0	0	1	ϕ	ϕ	ϕ	ϕ
0	0	1	0	1	0	0	0
0	0	1	1	0	0		
0	1	0	0	1	0	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0		
0	1	1	1	0	0		
1	0	0	0	0	1	ϕ	ϕ
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	0	0		
1	1	0	0	0	0		
1	1	0	1	0	0		
1	1	1	0	1	0	0	0
1	1	1	1	ϕ	ϕ	ϕ	ϕ

(a) Folding F2 about W and fitting NOR(X, Y, Z)

X	Y	Z	W	F1	F2	F3	F4
0	0	0	0	0	1	ϕ	ϕ
0	0	0	1	0	1	ϕ	ϕ
0	0	1	0	ϕ	ϕ	ϕ	ϕ
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	ϕ
0	1	1	1	0	0	1	ϕ
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	ϕ
1	0	1	0	1	0	0	0
1	0	1	1	0	0	0	
1	1	0	0	0	0	0	
1	1	0	1	1	0	0	0
1	1	1	0	0	0	1	ϕ
1	1	1	1	ϕ	ϕ	ϕ	ϕ

(b) Folding F3 about X and generating a maximum of DON'T CARES in F4

Z	W	X	Y	F1	F2	F3	F4
0	0	0	0	0	1	ϕ	ϕ
0	0	0	1	1	0	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	1
0	1	0	0	0	1	ϕ	ϕ
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	ϕ
0	1	1	1	1	0	0	0
1	0	0	0	ϕ	ϕ	ϕ	ϕ
1	0	0	1	0	0	1	ϕ
1	0	1	0	1	0	0	0
1	0	1	1	0	0	1	ϕ
1	1	0	0	1	0	0	0
1	1	0	1	0	0	1	ϕ
1	1	1	0	0	0	0	1
1	1	1	1	ϕ	ϕ	ϕ	ϕ

(c) Folding F4 about Z

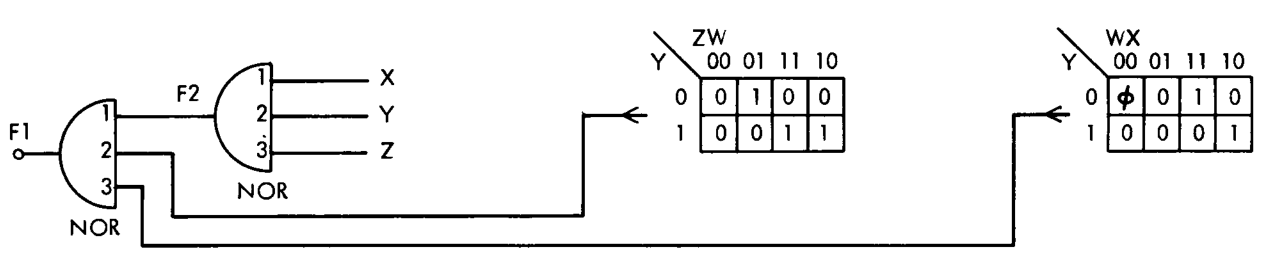


Fig. 2.4 Sample Decomposition Using Folding Techniques

Finally, a feature is provided for "backing-up", i.e., undoing a certain amount of previous work. When first writing the program to implement the method, the ability to "back-up" was considered necessary because it could not be a priori shown that the synthesis would always converge. Although convergence is guaranteed for NOR, NAND, and minority gates, thus making use of the backing-up feature unnecessary, the feature can be nevertheless used in striving to improve the optimality of the resulting configuration. Thus, from any given decomposition stage, the backing-up feature can be used to modify an earlier decomposition stage so as to obtain better results.

CHAPTER III

IMPLEMENTATION

A. PROGRAMMING CONSIDERATIONS

The reason for implementing the method described in Chapter II was to test the validity of the method, rather than the relative efficiency with which it could be carried out. Time limitations imposed restrictions on the amount of programming effort. Consequently, it was decided to use the AED-0[3] programming language for the bulk of the program, with selected short subroutines written in FAP[4] (IBM 7094 Assembly Language). AED-0 is an ALGOL-like symbolic programming language which has additional facilities (in the form of systems of special subroutines) for computer-aided design, such as easily-programmable free-format input-output and dynamic storage allocation. AED-0 programs can be written with the same ease as FORTRAN or MAD (other symbolic compiler languages), and AED-0 was chosen instead of the latter two because of its superior facilities. No claim is made as to the efficiency of the AED programs, however, since the prime consideration was to achieve a working program. Certain purely logical (bit-manipulating) tasks were written as subroutines in FAP, a far more natural language to use in those cases.

Time limitations, together with initial uncertainties about the extent of interaction necessary for the method to be properly implemented, dictated that considerable emphasis be placed on the interaction aspect of the system. From the standpoint of trying to write a program as rapidly as possible, a decision that would be very involved to program would be better left to the discretion of the operator, especially if it were based on intuitive, rather than computable factors. From the standpoint of a priori uncertainties in implementing the method, it was deemed expedient to give control to the operator whenever a sound algorithm could not be devised for the computer.

It should be borne in mind, therefore, that the programs described in the remainder of this chapter do not represent highly efficient programming or interaction systems, but are a rather expedient implementation aimed at testing the validity of the synthesis method. Because

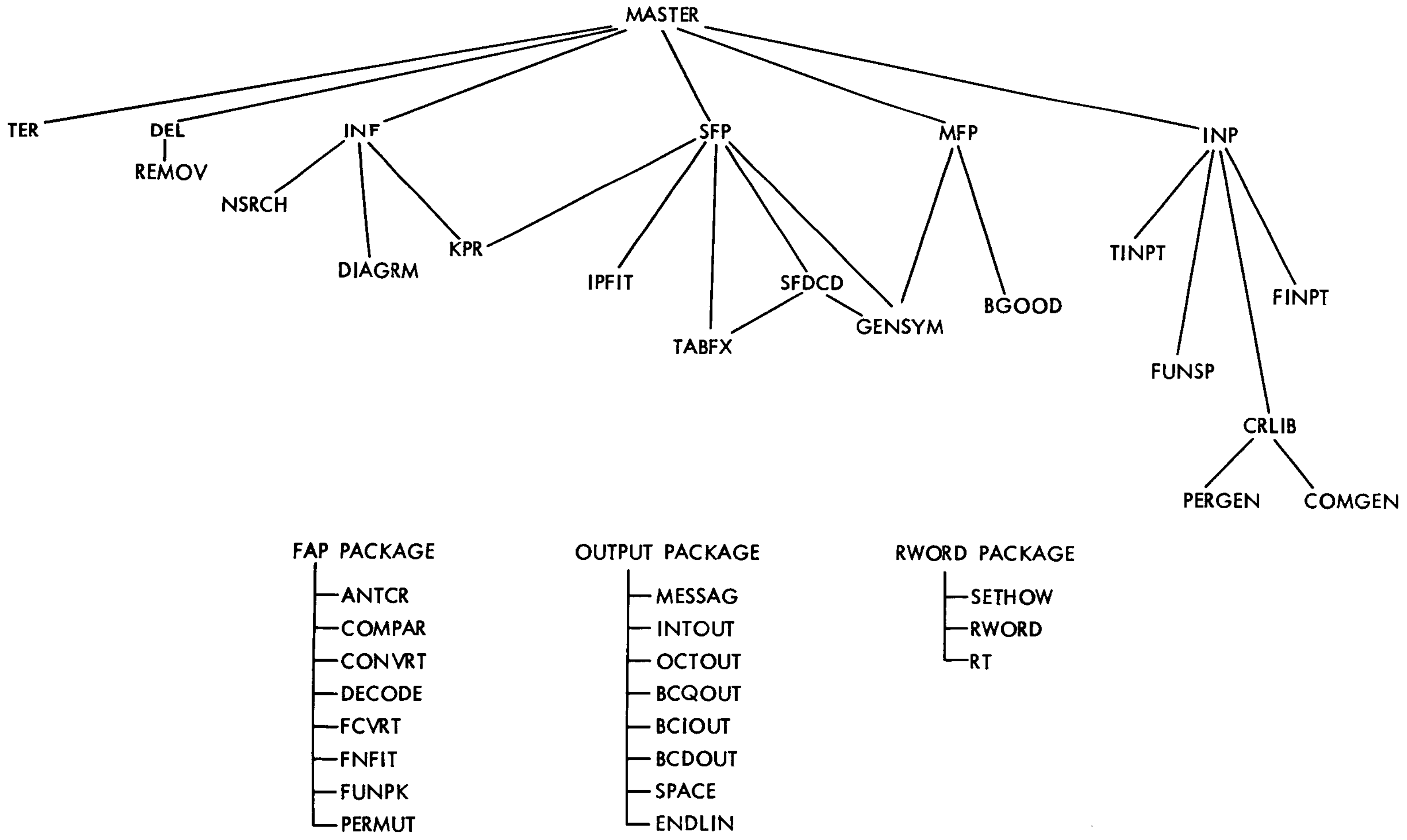


Fig. 3.1 General Program Structure for CADD-1

of this reason, no attempt is made to explain the fine details of each program. Instead, a more general description of all the programs and their interconnections is given here. Important data structures used and referred to in the remainder of this chapter are illustrated in Appendix I, and detailed flowcharts of each important program in CADD-1 are given in Appendix II. Flowcharts for the programs in CADD-2 are not given, due to time limitations and their resemblance to CADD-1 flowcharts.*

B. GENERAL PROGRAM STRUCTURE

1. CADD-1

The general program structure and hierarchy is given in Fig. 3.1. It should be noted that only the AED programs are included in the tree-like structure. This structure indicates the origin of calls on each subroutine, MASTER being the main program. The FAP package is the group of subroutines written in FAP for bit-manipulating purposes and each subroutine is called from within one or more of the AED programs. The OUTPUT package is a set of routines, provided as part of the AED programming system, for free-format output. The chief advantage of the OUTPUT package over the standard FORTRAN format statement is that printed output can be specified character by character, rather than a line at a time, considerably simplifying the programming of the machine-to-man interaction.

The RWORD package is analogous to the OUTPUT package, except that it is for free-format input, allowing the operator to type commands and data in a form most convenient to him. In using the RWORD package, initially SETHOW is called to establish the source of the input data (keyboard, tape, disc file, etc.) and subsequently whenever it is desirable to clear out the input buffer. Each call on RWORD gets a new "item" from the input buffer, stores the "item" in BCD form in a temporary location, and returns a pointer to this

*Copies of all programs are available from Paul J. Santos, Jr. at the Electronic Systems Laboratory.

location. An "item" is defined by parsing the left end of the input buffer according to the character table RT. This table indicates the type of each BCD character and further indicates with which other characters it can be grouped to form an item. An item is a sequence of characters which fit together, with delimiters on either side. Since one or more consecutive blanks or a carriage return are considered single delimiters, a satisfactory manner of inputing all items is to type them one after the other, separated by blanks, and on consecutive lines if necessary. All items will be in BCD form, which is suitable for interpretation of commands and of BCD data. If numerical data is expected, the BCD form is converted to an integer number by use of the subroutine DECODE.

The programs MASTER, INP, INF, MFP, SFP, and DEL, within which all the interaction takes place, are each equipped with a separate command structure. This structure enables the program to ask for and interpret a command from the operator and then branch to the appropriate executive subsection. Within the subsection there may be further requests for commands and /or data, and when the necessary processing is completed, control is returned to the main section which requests another command.

Dynamic storage allocation is handled by means of three subroutines, FREZ, FREC, and FRET, supplied also as part of the AED system. FREZ sets aside a block of consecutive computer words from free storage and returns a pointer to them so that they can be used to hold newly generated data. FREC is similar except that the block set aside from free storage is made identical to an already existing block. FRET returns blocks which are no longer needed to available free storage. A "pointer" is a variable whose value is some absolute location in core memory which is the address of the first word of a block.

All reference to blocks of free storage, for both storage and retrieval purposes, is made through pointers to the blocks. This referencing is further aided by the AED "bead structure" facility, which allows a component of a free-storage block, specified by the position of the word within the block and the position of the component within the word, to be declared and used on any pointer.

All data which is referenced from more than one program is assigned a location in COMMON storage, in order to eliminate the need for transmitting it as arguments in subroutine calls. The program variables so used are as follows:

1. STATUS - indicates the present status of the synthesis with respect to specification of function to be realized and blocks to be used. STATUS takes on four values: 0 - beginning; 1 - function specified, but no blocks specified; 2 - function and some blocks specified; 3 - decomposition begun.
2. INVARS - number of input variables.
3. INPTVARBS - pointer to block of storage containing variable names.
4. MFN - pushdown stack containing pointer to current function specification.
5. CBLK - pushdown stack containing pointer to building block (in tree) under consideration.
6. BBLK - pointer to building block directory.
7. LIBR - pointer to building block library.
8. NIL - pointer which indicates termination, either bottoms of stacks or ends of string-pointer lists.
9. PSTATE - indicates the present state of the decomposition: 0 - decomposition has not begun; 1X - select block to realize function on input X; 2X - select subfunction on input X; 30 - decomposition done.
10. TRUNK - pointer to head of block diagram (block that realizes output function).

Execution of the program begins with MASTER requesting a command. MASTER will accept seven commands, six of which cause it to call subroutines INP, INF, MFP, SFP, DEL, and TER, and one which prints out these commands in case the operator has forgotten them.* INP is the input subroutine, MFP and SFP are the subroutines which carry on the decomposition, INF and DEL serve to support the synthesis effort, TER ends the execution of the program. MASTER

*A typical feature of all the command subroutines.

serves as a junction point for the transfer of control from one of the above mentioned subroutines to another during the course of synthesis.

2. CADD-2

The general program structure for CADD-2, similar to that of CADD-1, is given in Fig. 3.2. CADD-2 makes use of a large number

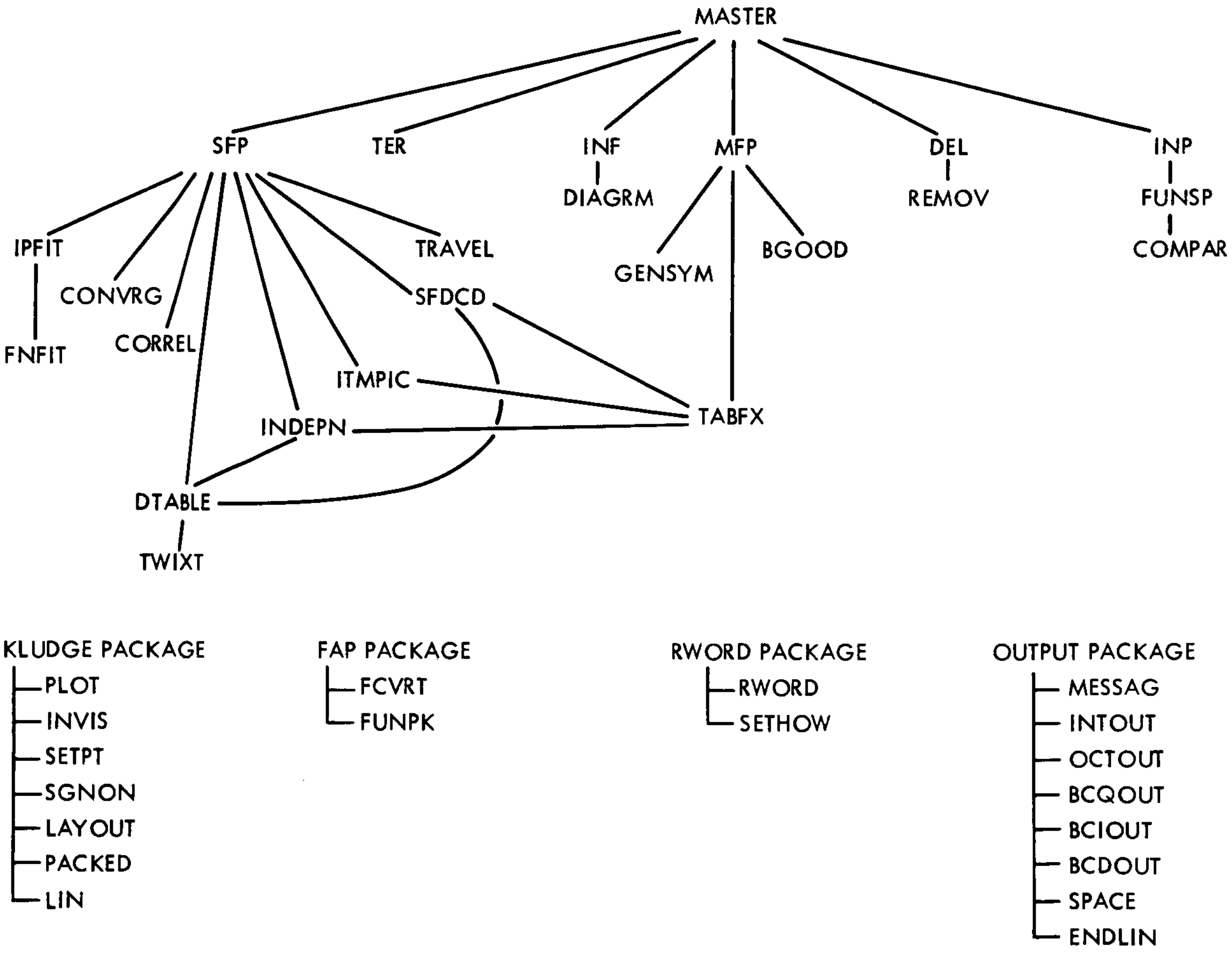


Fig. 3.2 General Program Structure for CADD-2

of utility subroutines not mentioned in the figure which are supplied along with all the other AED-0 programming packages. The discussion of the CADD-1 structure applies to CADD-2 as well, with the following exceptions.

The RWORD package was reduced in size (and flexibility) in order to accommodate the needs of CADD-2 without excessive program length. The latter version of RWORD reads "items" one at a time from the input buffer. An "item" is any grouping of non-blank characters delineated by blanks; thus the character table is made trivial and the processing considerably simpler.

In common storage, the older INVARS and INPTVARBS are incorporated into new INVARS, LIBR is deleted, and FNS, which is a string list of all previously realized functions, is added. PAGE and FITLIST are also added to common, the former to indicate the current page of the decomposition table displayed on the cathode ray tube, and the latter to point to the top of a list of "fits" for the current subfunction.

The "KLUDGE" package mentioned in Fig. 3.2 is the set of routines which enables the programming of a visual display. The display is used as a "fast typewriter" in order to reduce interaction time. The two AED programs that generate displays are DTABLE and DIAGRM, displaying respectively the up-to-date decomposition table and the circuit block diagram.

C. PROGRAM OPERATION

1. CADD-1

a. Input Phase The first subroutine to which MASTER transfers is normally INP. INP permits the input and editing of both the function to be synthesized and the set of building blocks to be used in the synthesis. The function must be specified before the blocks, and may be edited at any time thereafter until the actual decomposition process begins. Blocks are specified one at a time and may be edited at any time provided they haven't been used in the decomposition.

The function is specified by giving the number and names of the variables and the values of all the terms (rows) in the function truth table. The latter can be specified in two ways, depending upon the nature of the function. One way is to first set all rows to the same value, and then indicate which rows have different values. Rows are coded in natural binary sequence. Thus, for example, F(W,X,Y,Z) = WYX + XYZ is specified by first setting all rows to ZERO and then setting rows 7, 14 and 15 to ONE. The other way is to indicate, row by row, the values of the function. This information is first stored in the unpacked form (UNPACKED FN) shown in Fig. A1.3 of Appendix I, and when completed, it is converted to the packed form shown in the same figure. Packing and unpacking of functions is carried out by the use of the FCVRT and FUNPK subroutines, respectively. Finally, all the information concerning the function is put into the form of a FUNCT (also shown in Fig. A1.3) structure, and the pointer is stacked onto MFN to initialize the state of the decomposition. A FUNCT block is of length n + 2, where n is the number of variables on which the function depends. The first word of the block contains n, the second word contains a pointer to the packed function description, and the remaining n words contain numbers which indicate what the variables are and in what order of significance they appear, (the further down in the block, the higher the order). To find the variable occupying a certain position, the number in that position is added to the pointer INPTVARBS and the new pointer becomes the location of the BCD variable name. FUNCT structure is used through the decomposition process for storing all information concerning a particular function (or subfunction).

A block is specified by first giving its name and number of inputs, and then its truth table, in a manner similar to that for function above. The data structure for the block directory and the associated block library is shown in Fig. A1.2 (Appendix I). Every time a new block is specified, the following occurs:

1. Increase BLOCKNUM (total number of blocks) by 1.

2. Change MIP or MAP (minimum or maximum number of inputs of any block) if new block affects them.

3. Create a new entry at the beginning of both the block directory and the library. PDKFN points to the packed function representing the truth table for the block.

4. Create a generator list in the form given by FSPEC in Fig. A1.3 and insert the pointer in the FSPEC component of the block entry.

5. Find all symmetries of inputs, putting this information in the form of the list SYMLIST (Phase 1) given in Fig. A1.1 (of Appendix I). This form is used since it facilitates the incorporation of subsequent symmetries without affecting the already existing structure.

6. Convert the SYMLIST into the form (Phase 2) of the same figure.

7. Using the new SYMLIST, generate all possible nonredundant permutations of inputs and classify them in the form of the tree-list to which PERMUTATIONS of Fig. A1.1 points.

8. Create the list COMBLIST which uses PERMUTATIONS to generate all permutations and their associated packed functions.

9. For each permutation, generate the functions associated with all possible combinations of negations of the inputs, and add them to the list of functions under the library entry for the block if they are new functions. Thus, the library is subdivided according to blocks, and each block points to a list representing all possible functions realizable with that block. Furthermore, each element of the list contains an input code (indicating permutations and negations) and a pointer to a packed function.

The above steps make use of the following additional subroutines:

FUNSP - creates the generator list.

CRLIB - creates the block library.

PERGEN - generates PERMUTATIONS in CRLIB.

COMGEN - generates COMBLIST in CRLIB;

COMPAR - compares and merges terms for use in FUNSP.

PERMUT - permutes values of inputs within a generator term for use in generating SYMLIST in CRLIB.

CONVRT - converts terms from one- to two-bit mode for use by FUNSP.

INP - can be re-entered at any time during the decomposition for the purpose of specifying new blocks or editing old ones which have not yet been used; further decomposition will then be based on the new block directory and library.

b. <u>Decomposition Phase</u> After the function to be realized and the set of blocks to be used have been specified within INP, control is returned to MASTER. The main course of the decomposition and synthesis then involves alternate calls on the subroutines MFP and SFP, with occasional calls on the support routines discussed in Section C.

MFP is called whenever the state of the decomposition process requires that a choice of block be made to realize the current function specification. SFP is called to handle all matters pertaining to the decomposition of a function using a block selected by MFP. SFP and MFP operate in such a way (using the common variable PSTATE) that only the proper one can be entered at any given time. At the beginning of the decomposition process, MFP is called to decide on a block to realize the given (original) function. Then, SFP is called to decompose that function into subfunctions, one or more of which may require a new call on MFP and consequently on SFP. This process continues until SFP realizes all subelements of the block diagram tree, without further calls on MFP, at which time the process is backed up to the level of the output block with no more inputs left unspecified.

MFP contains commands which allow the user to find out the present objective (i.e., the input and block associated with the current function), to find out the theoretically best block to use, and to specify which block to actually use.

The best block to use is found by applying the subroutine BGOOD to every block in the directory and noting the one which produces the highest value. The program gives BGOOD the number of ONEs and ZEROs of the function, as well as information concerning the generators for each block; the user gives it four weights (DCWT, CONSWT, VARWT, and INPWT) which are used to weigh the average number of DON'T CAREs in the generator lists, the average number of constraints in the generator lists, the elements of that list and the number of inputs of each block, respectively.

When a decision is finally reached concerning the block to be used, a new element is (a) added onto the previous tree structure, (b) connected to the input entry in the previous block (specified by PSTATE), (c) given a unique name generated by the subroutine GENSYM, and (d) filled in with all the proper initial information, such as number of inputs, type of block, and output function. A decomposition table is created and filled in with all initial restrictions and the generator table is initialized to contain all generators. Finally, control is returned to MASTER with PSTATE set to begin decomposing the function on input No. 1.

At this point it seems appropriate to explain in detail the data structure needed to contain the growing tree of the block diagram and intermediate decomposition results. A typical element in this structure is illustrated in Fig. A1.4 of Appendix I. The main block consists of n + 2 words, n being the number of inputs of the block. In the first word, SPEC indicates whether or not the entire block has been fully realized through all of its inputs, OUTFN is a pointer to a FUNCT type structure specifying the function realized at the output of the block, INPUTS is the number of inputs to the block, and NAME is a pointer to the BCD name of the element (of the form AND004, OR021, NOR015, etc.). In the second word, TBL is a pointer to the decomposition table and BLKTY is a pointer to the building block directory entry of that type of building block. The third through n + 2 words correspond to the first through n inputs, and consist of: SPEC, which indicates if the subfunction on that input has been realized; INPFN, which, if the subfunction is realized by another block, is a pointer to a FUNCT type structure specifying the subfunction; INPCODE, which indicates whether the subfunction is another block (4), a negated variable (3), a variable (2), or a constant (1); and NEXTBLK, which is (a) a pointer to a similar element of the structure if the subfunction is realized by a block, (b) a pointer to the BCD name of a variable if the subfunction is a variable or its negation, and (c) 0 or 1 if the subfunction is constant ZERO or ONE. An element block which has been completely specified is stripped of its decomposition table, leaving only the structural skeleton in finished portions of the block diagram. The decomposition table consists of n + 1 words. In the first word, GEN is a pointer to the generator table, and FN is a pointer to an UPKDFN-type structure containing the function specification for the block. The remaining n words correspond to inputs 1 through n and each contain SPEC, which indicates if the input is filled (note: the input may not be realized yet), and COLL, which is a pointer to another UPKDFN-type structure containing the partially specified or complete subfunction. The generator table is divided into two parts to accomodate the case when there are a large number of ZERO- or ONE-generators for a certain type of block, but normally only the first half, to which GEN points, is used.

Each half is of the same length as the UPKDFN structures, thus giving a generator specification for each term of the function. SPEC indicates when only one generator remains for a given term of the function, thus fixing all subfunctions in that row (term). A 1 in the leftmost (33rd) bit of GENCODE indicates that the function has a DON'T CARE in that term, whereas a 1 in any other bit position (counting from right) of GENCODE means that that particular ZERO or ONE generator is still valid for the function term. Thus, a term containg ZERO (or ONE) initially contains g 1's in its rightmost g bit positions, corresponding to the g ZERO (or ONE) generators in the block specification; furthermore, initial restrictions on certain columns (inputs) are determined from these g generators. As the decomposition table is gradually filled in, added restrictions limit the choice of generators (which are erased bit-by-bit from GENCODE), until only one generator is left and the row is completely filled in.

SFP has a large number of commands designed to handle all aspects of decomposition and to provide some aid to the user in making decisions concerning the decomposition. These commands accomplish the following tasks: indicate the present objective (which input of what block) of the decomposition; change the objective of the decomposition; find all possible immediate realizations which fit the present partially specified subfunction; find which one (s) has (have) the highest correlation or anticorrelation factor with the main function; decide which of these to use; fill in an item (row) of the present subfunction; travel on to further decomposition once an input is completely filled; give the state of convergence of the present subfunction; rotate the variables of the function in order to better detect some foldings; and print out the Karnaugh map of any library function.

The information concerning which input (hence, which subfunction) is currently decomposed is contained in PSTATE: Once an input has been completely realized, it cannot be made an object of SFP. Whenever the subfunction being decomposed changes, FITLIST, which is the string-pointer list indicating the immediately realizable functions, variables, and constants that fit into the present subfunction, is erased. Correlation, anticorrelation, and the decision as to which immediately realizable subfunction must be chosen, work only with a non-empty FITLIST. These details, and a number of others, are ommitted from the present discussion so as not to confuse the main thoughts.

A typical mode of operation in decomposition, for each new input is as follows:

1. Find all fits. If nothing fits, go to Step 5 below.
2. Correlate or anticorrelate.
3. If not satisfied,* permute variables and go back to Step 1. All permutations of variables can be achieved.
4. If still not satisfied, go to Step 5. Otherwise, specify which immediately realizable function is desired. The input is then filled with the selected subfunction and new restrictions are filled in. The decomposition process may be repeated next for a new input. If no inputs remain to be realized, and if the top of the tree has been reached, then the decomposition is over; otherwise, return to the higher level and look for unrealized inputs. This process is repeated until an unfilled input is found at some higher level or the process terminates.
5. Fill in by hand all remaining blank items of the subfunction using previously mentioned techniques.
6. Fold the subfunction (via the command TRAVEL) wherever possible and change PSTATE so that when control is returned to MASTER, MFP will be called next.

Subroutines used during the above process and their descriptions follows:

IPFIT - performs the task of generating the FITLIST.

SFDCD - performs the task of choosing an immediately realizable function and carrying out the consequences.

KPR - prints out the Karnaugh map of a function.

GENSYM - generates a new, unique name for blocks in the tree each time it is called.

BGOOD - evaluates the "goodness" of a block to be used to realize a function.

*Satisfaction rests with the operator and involves fitting, along with the fullfilment of certain criteria of goodness.

ANTCR - used by MFP in correlation and anticorrelation.
FNFIT - used by IPFIT in finding fits.

Many times it is difficult for the user to be fully aware of all the details of the process, especially after a permutation of variables or in trying to fill in a subfunction by hand. Moreover, the user may wish to "back up" in order to achieve a better realization. Both these cases are considered as parts of the support phase for the decomposition process rather than parts of the decomposition and are handled by INF and DEL, respectively, to be discussed in the next section.

c. Support Phase. The support subroutines INF and DEL supply additional information concerning the various aspects of the decomposition and provide a means of retreating from a situation which is considered unsuitable by the operator. INF can be called at any time from MASTER, whereas DEL can be sucessfully called only when PSTATE indicates the "subfunction" (2X) mode. The reason for the latter restriction is that there will be no need to "back up" while trying to decide what type of building block should be used in realizing the current function. DEL enables the operator to do one of two things: (a) Erase the entire present element of the tree, and all the structure dependent on it, putting the state of the decomposition back to where a call on MFN to re-realize the present function is appropriate, or (b) retreat yet one step further, and place the process in a subfunction-picking mode with reference to the tree element from which the original element was derived. In the first case, CBLK is unstacked and the entire present element (including its decomposition table), together with the portions of the tree connected to its inputs, are deleted. One input alone cannot be deleted since it normally affects the decomposition of all other inputs. Control is then returned to MASTER with PSTATE indicating that MFP should be called next. In the second case, both CBLK and MFN are unstacked, thus undoing the effect of a previous "travel" command. PSTATE is set to a mode indicating subfunction selection on the particular input of the higher-level block which was previously connected to the recently deleted element. Control remains in DEL in case further retreat is desired. In both cases, special provision is made for treating the process when it is backed up to the top of the tree.

INF is a completely passive subroutine in the sense that its only purpose is printout of information. INF permits the operator to ask for the following information: Status of decomposition; number of input variables; names of the input variables; Karnaugh map of current main function; number and names of building blocks; members of block library corresponding to a given building block; state of decomposition; condition of present tree element (block); condition of any element in tree; Karnaugh map of any completely specified input function to present element; decomposition table, including generators for present element; and current block diagram (tree). By far, the most frequently requested information concerns the decomposition table, since it changes every time a new item of a subfunction is filled in, or whenever the function is rotated. Normally, there are several repeated transfers of control from SFP to MASTER in the course of decomposition of a single function, since when filling in a subfunction by hand the precise state of the table must be known.

Subroutines used by INF and DEL and their descriptions follow:

NSRCH - searches the entire block diagram tree for an element with a given name. Used by INF to give the condition of any element in tree.

DIAGRM - prints out current block diagram. It is a good illustration of the superiority of the OUTPUT package over FORMAT statements, since several parts of the same line may be printed out by successive recursive levels of DIAGRM, which is a recursive procedure.

REMOV - deletes an element and all its subelements from the tree, deleting also the decomposition table of the top element.

When the synthesis process has been completed, the circuit realization can be obtained via a call on INF to print out the final block diagram.

2. CADD-2

a. Input Phase. The input phase of CADD-2 operates much in the same way as that of CADD-1, although certain details are different. An alternate method of inputing which saves interaction time is available for both function and block specification. This method consists of writing a small program in a standard format which specifies completely a function or building block, and compiling this program prior to execution of CADD-2. Then, whenever a specification is called for within CADD-2, the particular program which is desired is loaded into core memory. This method is particularly helpful for blocks and function which are going to be used frequently (such as standard logic gates) since the effort expended in writing the program is small compared to the effort expended in re-specifying the block or function for every execution of CADD-2 in which it is used.

A new parameter which has been added to the block specification is fan-out. This parameter restricts the number of times a specific subfunction can be used within the block diagram by restricting the fan-out of the block which realizes that subfunction.

The "function specification" format for CADD-2 differs in various respects from its predecessor (see Fig. A1.5). Besides the fan-out restrictions, all functions are kept in canonical form, i.e., with all the variables in the same order, and with the component VCODE to indicate the variables on which the function depends. Under the arrangement of the decomposition process of CADD-2, all rotation, folding, etc., is performed dynamically, so that the functions are stored in canonical form only, therefore making easier the testing for fits.

All of the programming which generates the block library in CADD-1 is absent from CADD-2, since no such library is now used. This approach greatly simplifies the concepts, computing time and storage used in decomposition.

All subfunctions that become fully realized are placed in an ordered list to which FNS points. When a subfunction is being treated for "fits", elements of the FNS list posessing the same variable dependence are checked, hence, it is now possible to fit an already existing function and to permit a fan-out of more than one for blocks picked from FNS.

b. Decomposition Phase. The main differences between CADD-1 and CADD-2 decomposition lie in the subroutine SFP. MFP is virtually unchanged, except to accomodate the new programming details, and to speed up the interaction, such as an automatic block selection if there is only one block in the directory. SFP for CADD-2 is a self-contained working unit, with need to transfer to INF only for displaying the current block diagram. This is done by having the current decomposition table constantly displayed on the CRT, with any occuring changes immediately updating it. Most of the CADD-1 SFP commands remain in CADD-2, unchanged in intent, but somewhat changed in content. The subfunction fitting command has improved interaction abilities and no longer checks a function library, but rather checks the FNS list (along with constants and variables). There are many interaction time improvements in the (anti) correlation, subfunction selection, manual table filling, traveling, and convergence information command, as well as changes in programming due to the new data structure. Two new commands were added, (a) to give the degrees of freedom (lack of dependence on input variables) of the current subfunction, and to fill in the table in such a way as to preserve this independence, and (b) to turn the pages of the decomposition table on the CRT if the table is too long to be displayed at once.

A slightly revised technique for decomposing a subfunction within SFP proceeds as follows:

1. Find all fits. If nothing fits, go to Step 4 below.
2. Correlate or anticorrelate.
3. If nothing is satisfactory, go to Step 4. Otherwise, fill in subfunctions that gave best factor in (2) above. Return to Step 1, for the next subfunction or go the next higher level.
4. Find independences. If none exist, go to Step 5, otherwise, make the subfunction independent of one or more of its variables.
5. Fill in any remaining entries in the table as judiciously as possible and "travel".

A number of other details of CADD-2 are ommitted, since they only differ slightly from CADD-1.

c. Support Phase. The DEL Section of CAD-2 is identical to that of CADD-1, except for details concerning the new data structure. The INF Section is no longer the same as that of CADD-1, since it only need display the current block diagram of the combinational system under synthesis.

CHAPTER IV

RESULTS AND CONCLUSIONS

A. CADD-1

1. Limitations

The CADD-1 implementation of the generalized synthesis method given in Chapter III suffers from a few limitations. These limitations, along with methods for eliminating them, are discussed in the present section. A large part of these suggestions are incorporated in CADD-2.

One major limitation is that the synthesis process consumes far too much real time to be commensurate with practical computer-aided design. The extra time is due mostly to waiting while the console (typewriter) types out advice and data, and to a smaller extent due to certain tedious input tasks, such as filling in a long subfunction by hand. The problem can be traced to two distinct sources. The first is that there is too much interaction (overused in CADD-1 so as to allow the operator to intervene in all critical tasks). Experimental use of CADD-1 indicated that much of this freedom was unnecessary, and should be eliminated by programming, rather than by interaction. Furthermore, much of the CADD-1 printout was not usually required for the decomposition process. The second source of unnecessary delay is typewriter speed. For example, the decomposition table for a six-variable function covers a complete page and consumes five minutes of typing.

In order to reduce interaction amount and time, the following changes were deemed appropriate.

1. Program many of the choices now left to the operator. Experience has shown where this can be done with safety.

2. Abbreviate much of the printout, and include options to eliminate printout completely at the operator's discretion.

3. Include ability to chain many commands. To make the process even faster, a decomposition strategy which is considered successful can be included in the input phase along with the building blocks. The program can then simply follow the strategy under normal conditions, resorting to interaction only when special circumstances arise.

4. Add features which enable a short command to accomplish the same objectives as a previously long and tedious input operation.

5. Use a cathode-ray tube graphical display instead of the typewriter for unavoidably long outputs, such as decomposition tables and block diagrams. This feature alone can cut the real-time usage by almost fifty percent.

Another major limitation of the CADD-1 implementation is that it is not thorough enough in checking possible moves. Instead, it puts the burden on the operator, who can either go through the tedious process of permuting variables, folding, etc., or simply make quick but arbitrary decisions thus probably missing a "better" solution. Many times the operator is forced to do some tedious processing himself since no command will give him this information. Such processing might involve the determination of dependence of a function on certain variables, or the configuration of an immediately realizable library function within a subfunction.

Since CADD-1 uses a very small amount of computer time (about fifteen seconds for a four-variable function), some greater searching and processing capability can be delegated to the computer.

Most of the above discussion had indicated a probably increase in complexity of the program. Since the CADD-1 program is quite large (about two thirds of core memory), simplifications and use of essential features should be considered. One possible simplification is to eliminate the library functions, since they reduce the decomposition by one level, i.e., they eliminate the need for an extra call on MFP and SFP and the filling in of variables. Thus INP can be considerably simplified, and CRLIB and its tributaries can be eliminated. SFP would then be free to do a more perceptive analysis of each function and its possible decompositions, no longer having the added task brought about by library functions.

Finally, many improvements of a minor nature can be made in the implementation, such as the standardization of packed components, variable names, data structures, and procedures which vary slightly from each other and the rewriting of many logical subprocesses in FAP rather than AED.

2. Comparison with other Methods

As mentioned in Section C, Chapter I, a universal basis for comparison and evaluation of results of the general synthesis method is the "brute force" method of converting the synthesis problem to the form of a two-level classical synthesis realization. Nine test cases and their results are shown in Table I. In each case, CADD-1 gives a more optimal solution than the "brute force" techniques.*

Several things should be noted about the results of Table I. First, the cases in which CADD-1 is superior to other methods of synthesis are those which deal with "unusual" sets of building blocks, such as Cases III and IV. The reason for this is that these sets of blocks lend themselves less easily to the (Boolean) algebraic manipulations (which underlie the "brute-force" method) than the more standard AND-OR, NOR and NAND gates which have a simpler Boolean algebraic structure. Second, in cases which deal with more conventional blocks, CADD-1 gains advantage from its capability to arrive at more than two-level realizations. Thus, in Case VI, the superiority of CADD-1 lies in being able to construct a symmetrical, four level OR-AND-OR-AND tree, whereas the "brute-force" method needs two extra ANDs in the necessity of maintaining only two levels in the tree. A similar though less symmetrical situation occurs in CASE V, where the total number of block inputs decides the more optimal solution.

On balance, it can be said that CADD-1 represents a reasonable initial step to the solution of the generalized synthesis problem for combinatorial digital networks.

3. Effect of Human Operator

One final question that must be asked is: What is the dependence of the system upon the operator? Or, similarly, what is the effect of the skill of the operator upon the result? There is no doubt that operator skill affects the results in a very positive way; this skill, however, can be acquired after some use of CADD-1 because of the great adaptability of the human brain.

*Where it is assumed that both complemented and uncomplemented variables are available.

B. CADD-2

At the time of the writing of this report, the programming system to implement CADD-2 still contains several program errors which inhibit its full operation. Few test cases can be run without using the areas of the program which contain these errors. Consequently a complete list of long examples is not included in this report. On the other hand, real-time used for CADD-2 based on a small number of simple examples shows a five-to-one reduction over the time used by CADD-1 for the same examples. A sample run of CADD-2, using the same example as that used in CADD-1, is given in Appendix D.

TABLE 1

Results of Test Cases for CADD-1

	NO. OF		BUILDING	REALIZATIONS	
CASE	VARIABLES	FUNCTION	BLOCKS	CADD	"BRUTE-FORCE"
I	6	Σ(0,5,9, 13,14,24,26, 32,33,34,35, 40,51,60)	3-INPUT NOR	40 NORs	48 NORs
II	5	Σ(0,5,9 13,14,24,26)	3-INPUT NAND	16 NANDs	28 NANDs
III	4	Σ(0,3,4 5,6,8,10,15)	3-INPUT MINORITY	13 MINs	48 MINs
IV	4	same	3-INPUT EXCLUSIVE OR, 3-INPUT AND	5 ANDs 5 XORs	11 ANDs 11 XORs
V	4	π(0,1,2, 4,8)	2,3-INPUT ANDs, 2,3-INPUT ORs	1 3-OR, 1 3-AND, 2 2-ORs, 2 2-ANDs (14 inputs)	1 3-OR, 1 2-OR, 4 3-ANDs (17 inputs)
VI	3	Σ(1,2,4, 7) (3-input XOR)	2-INPUT AND, 2-INPUT OR	3 ORs 6 ANDs	3 ORs 8 ANDs
VII	3	same	3-INPUT NAND	6 NANDs	8 NANDs
VIII	3	same	2-INPUT NAND	9 NANDs	18 NANDs
IX	3	Σ(0,2,5) (see Sample Run, APPENDICES C and D)	2-INPUT	5 NORs	7 NORs

APPENDIX A

DATA STRUCTURES

Figure Page

CADD-1

CADD-2

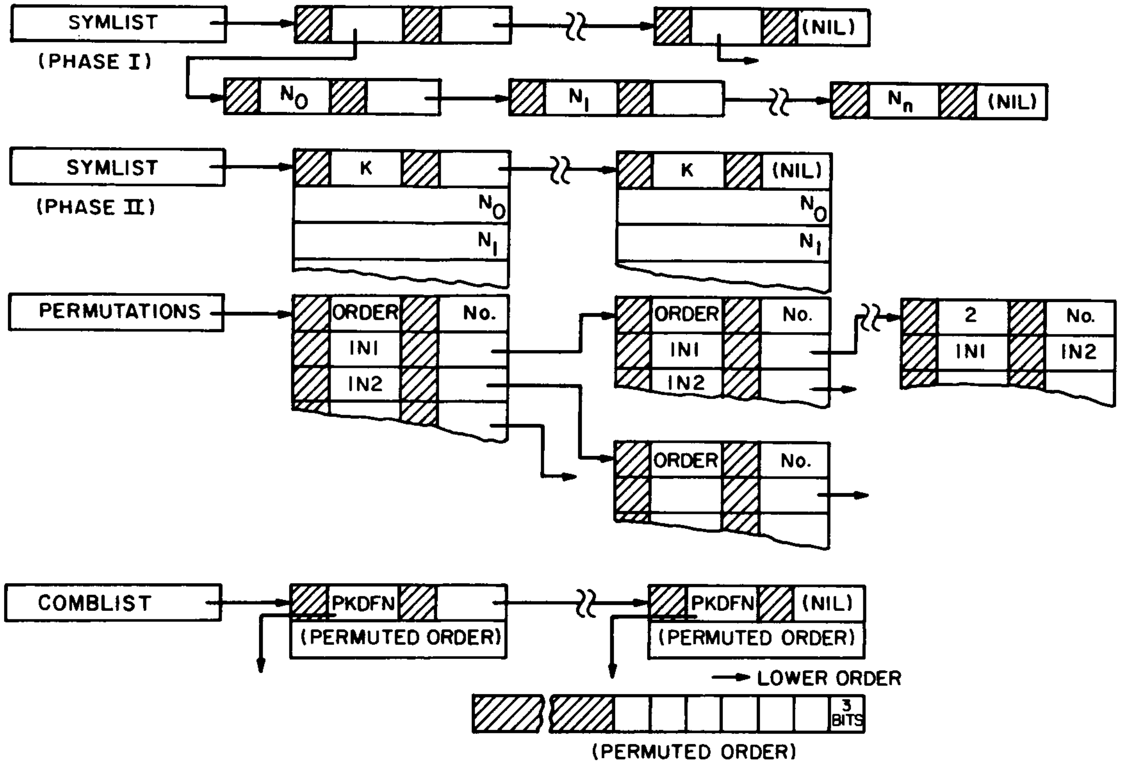


Fig. A1.1 Structures Used in Library Generation

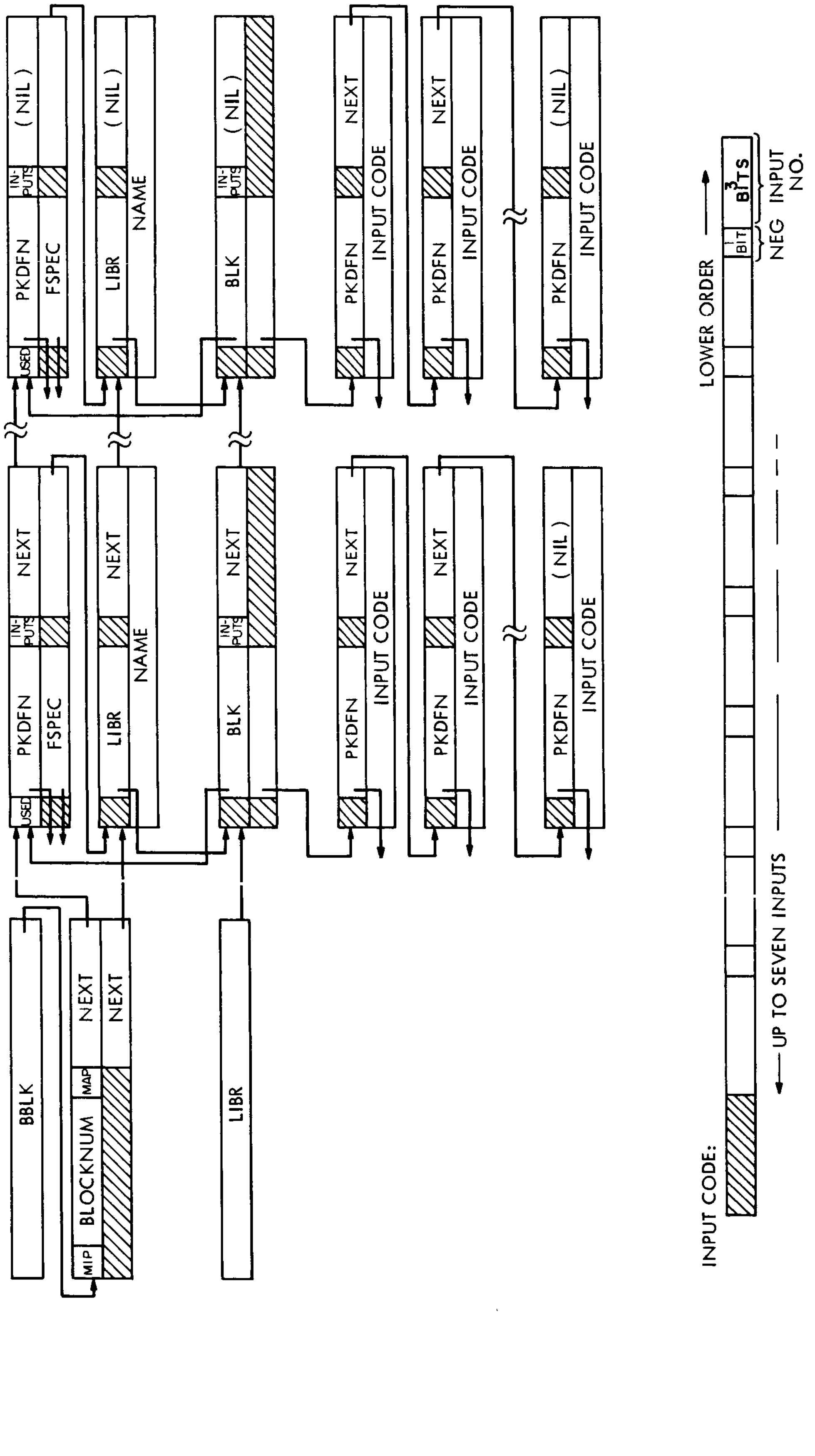


Fig. A1.2 Building Block Directory and Library

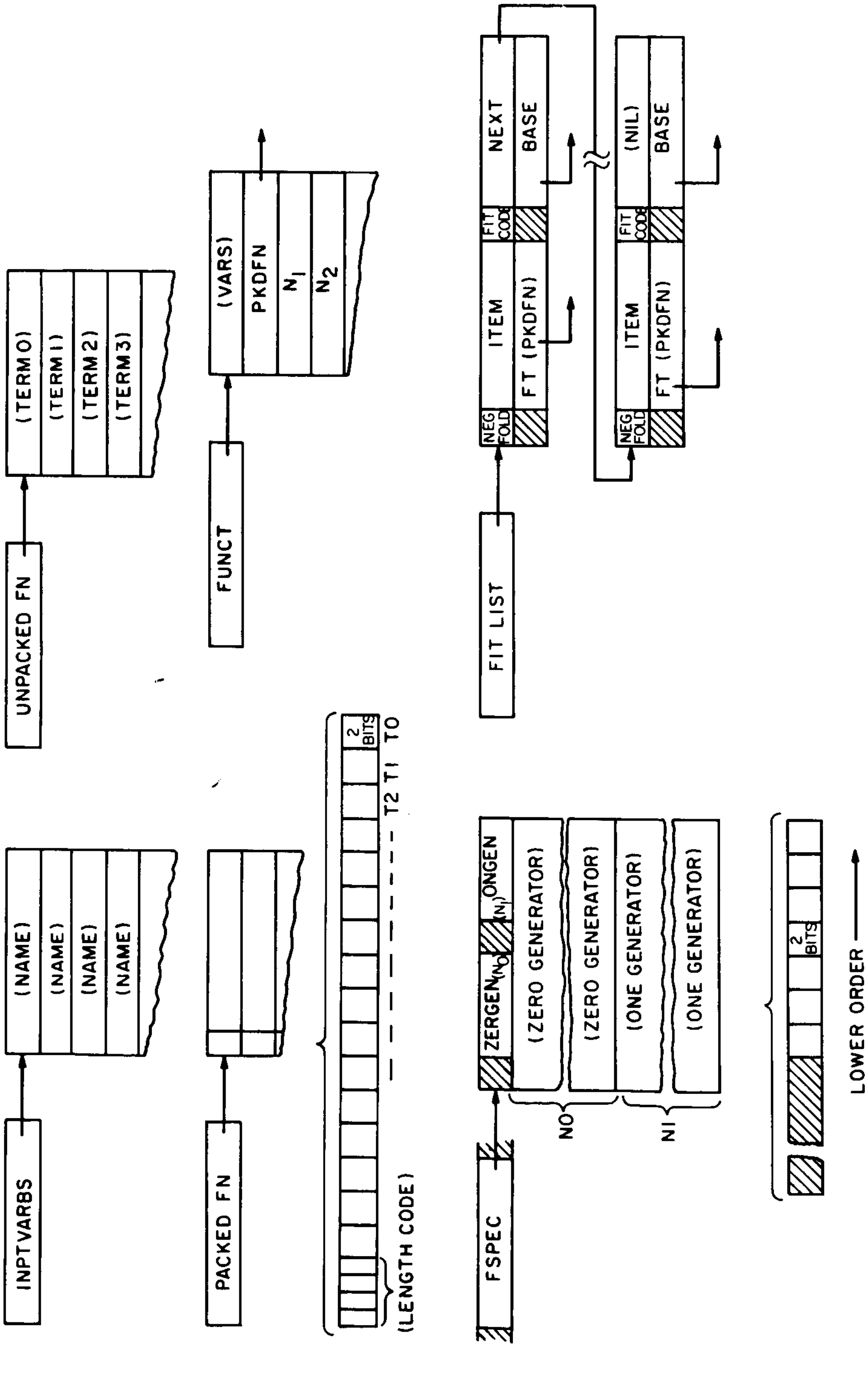


Fig. A1.3 Common Structures Used During Decomposition

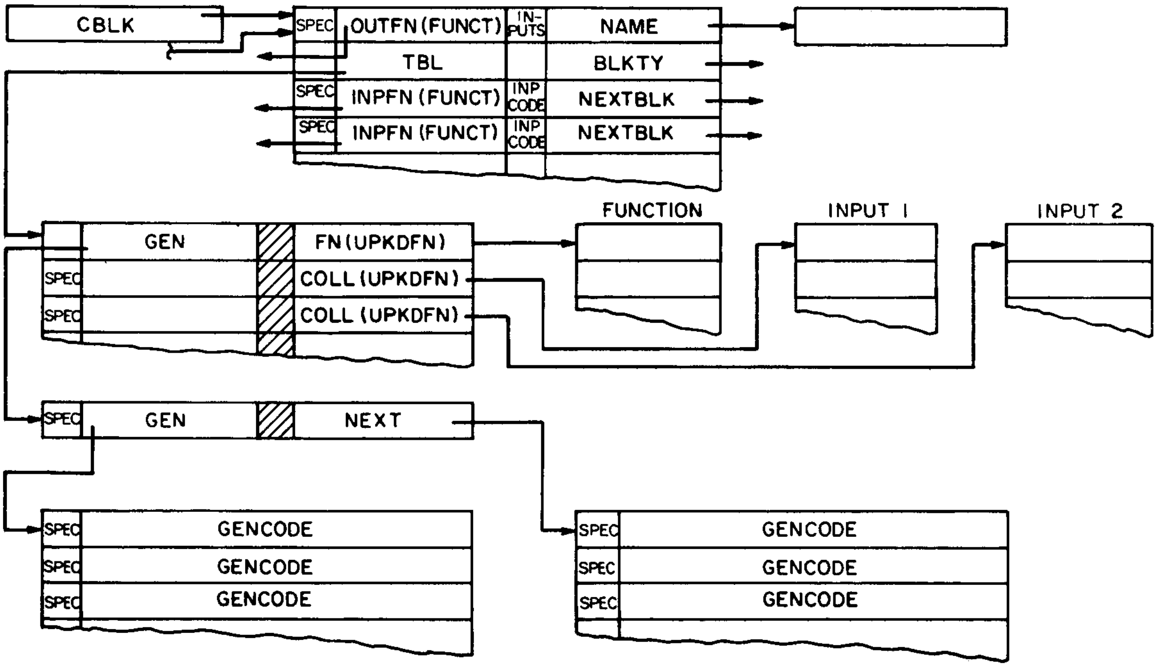


Fig. A1.4 Typical Tree Element

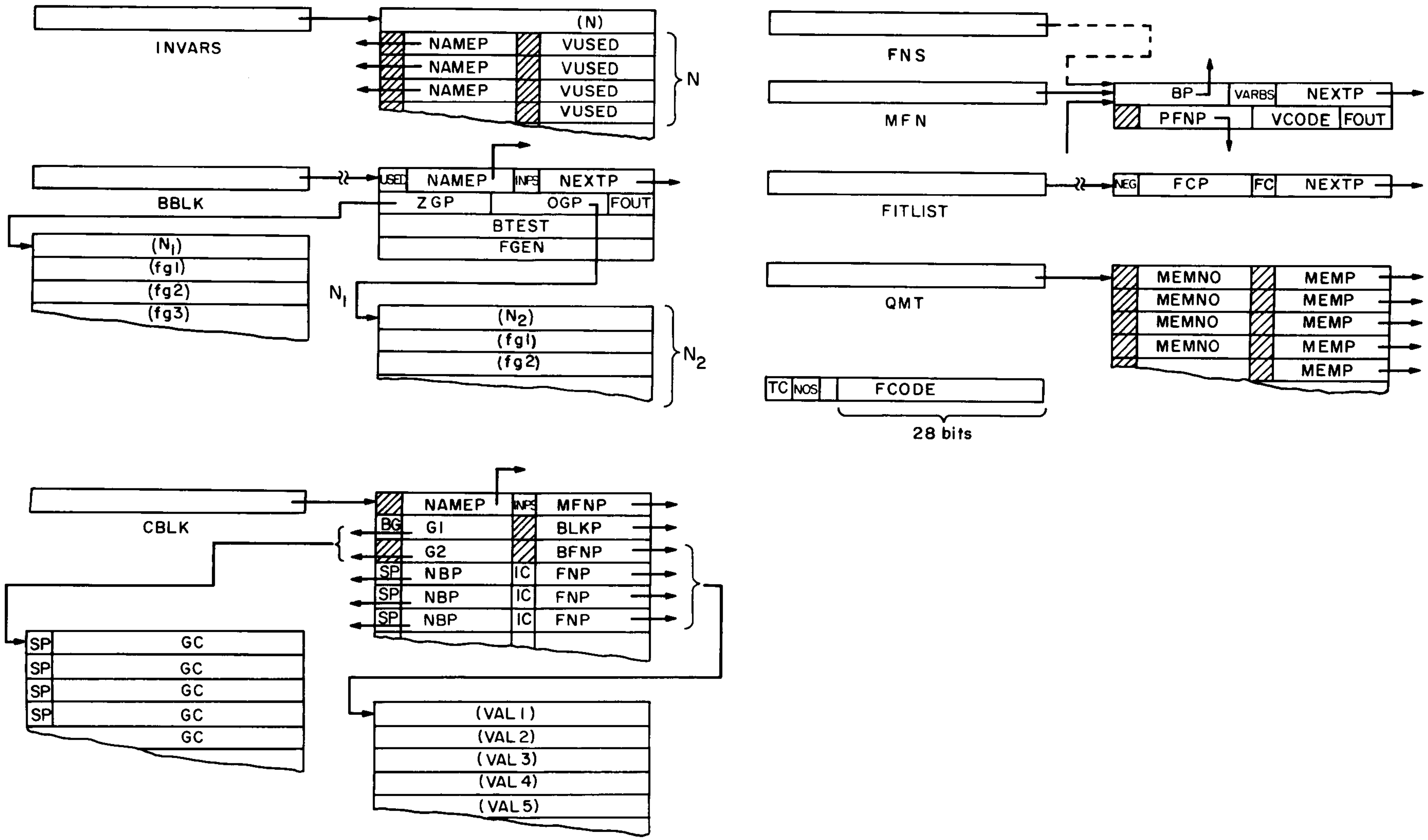
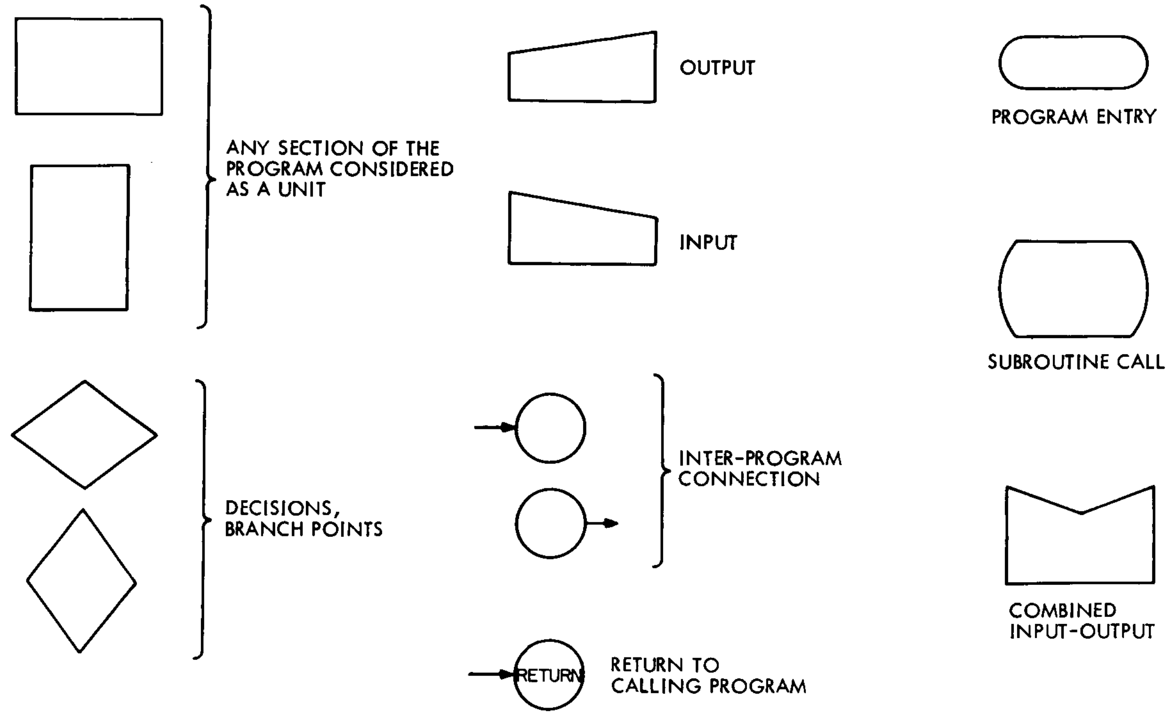


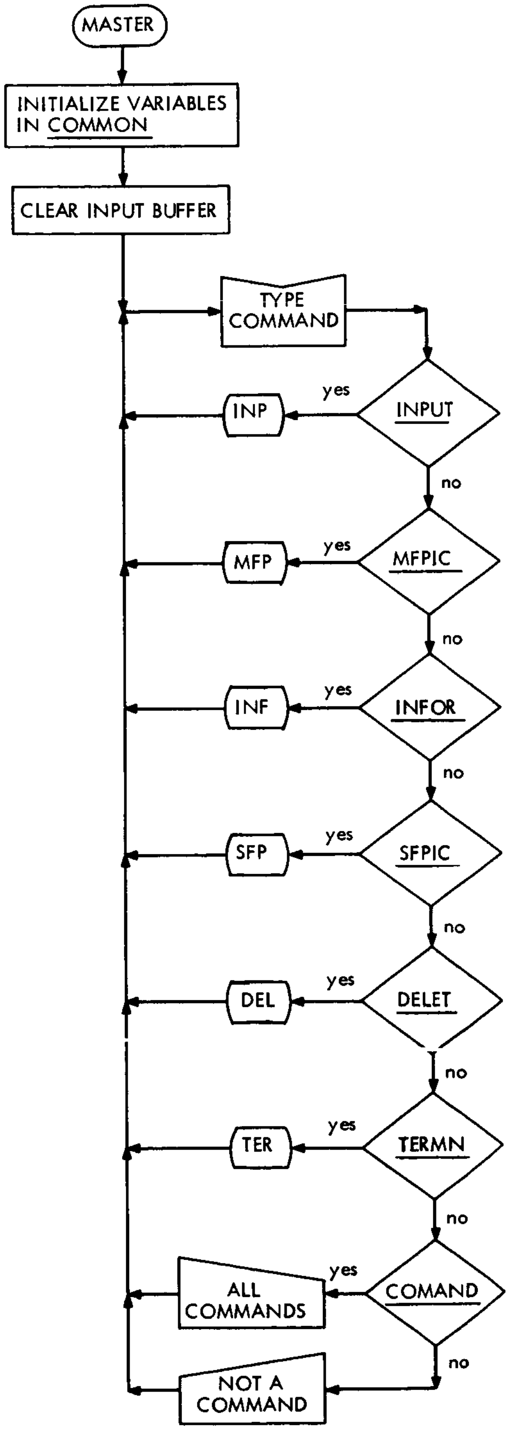
Fig. A1.5 CADD-2 Data Structures

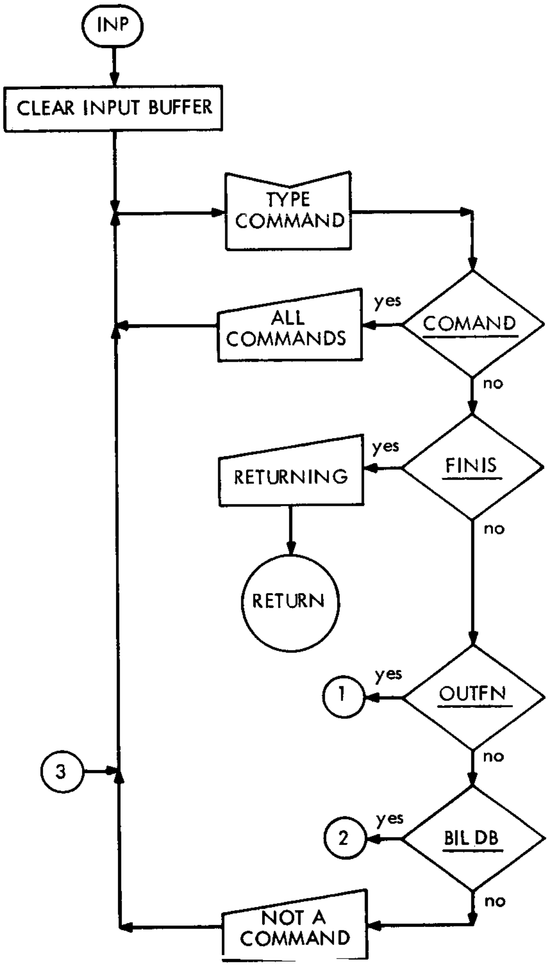
APPENDIX B

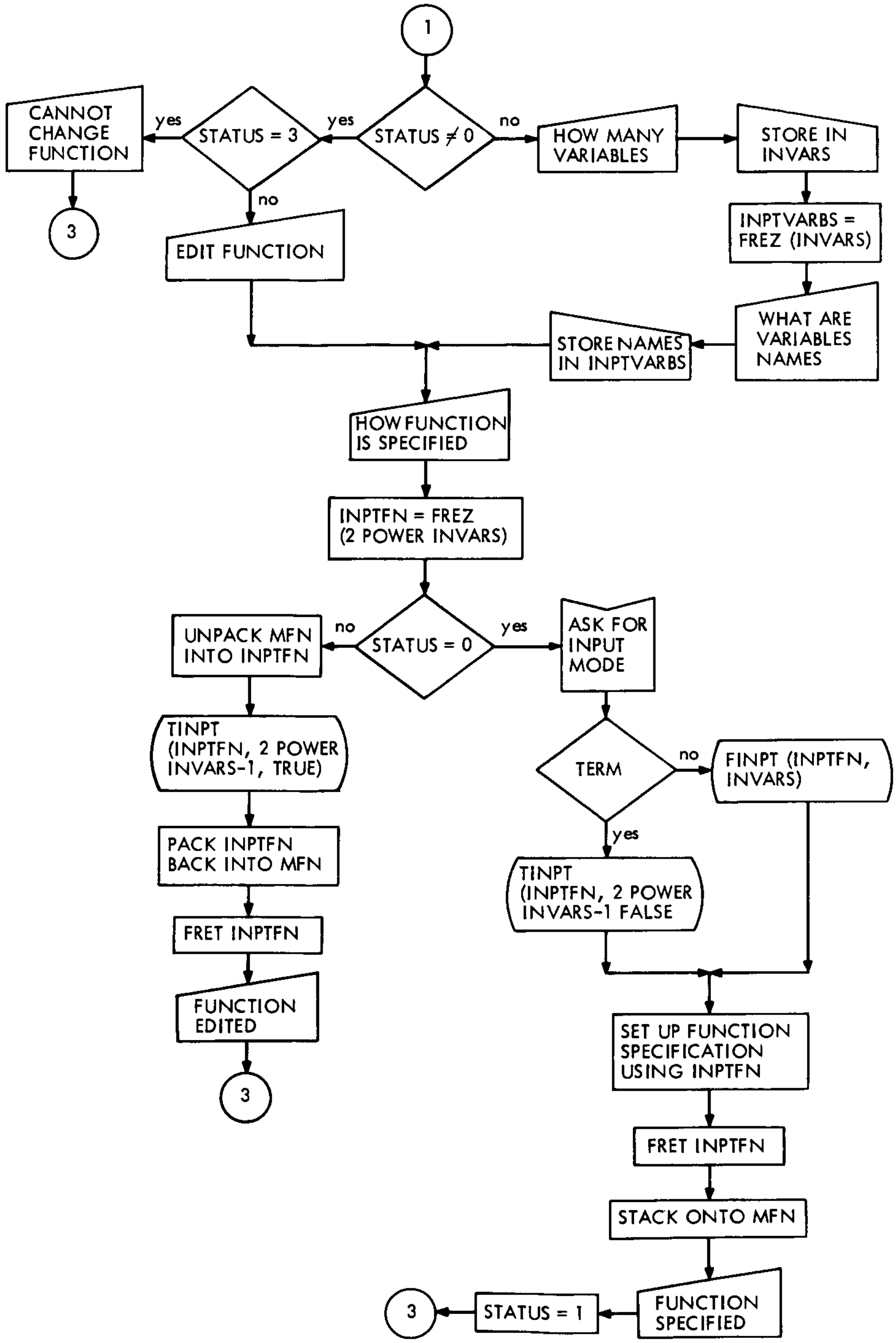
DETAILED FLOW CHARTS

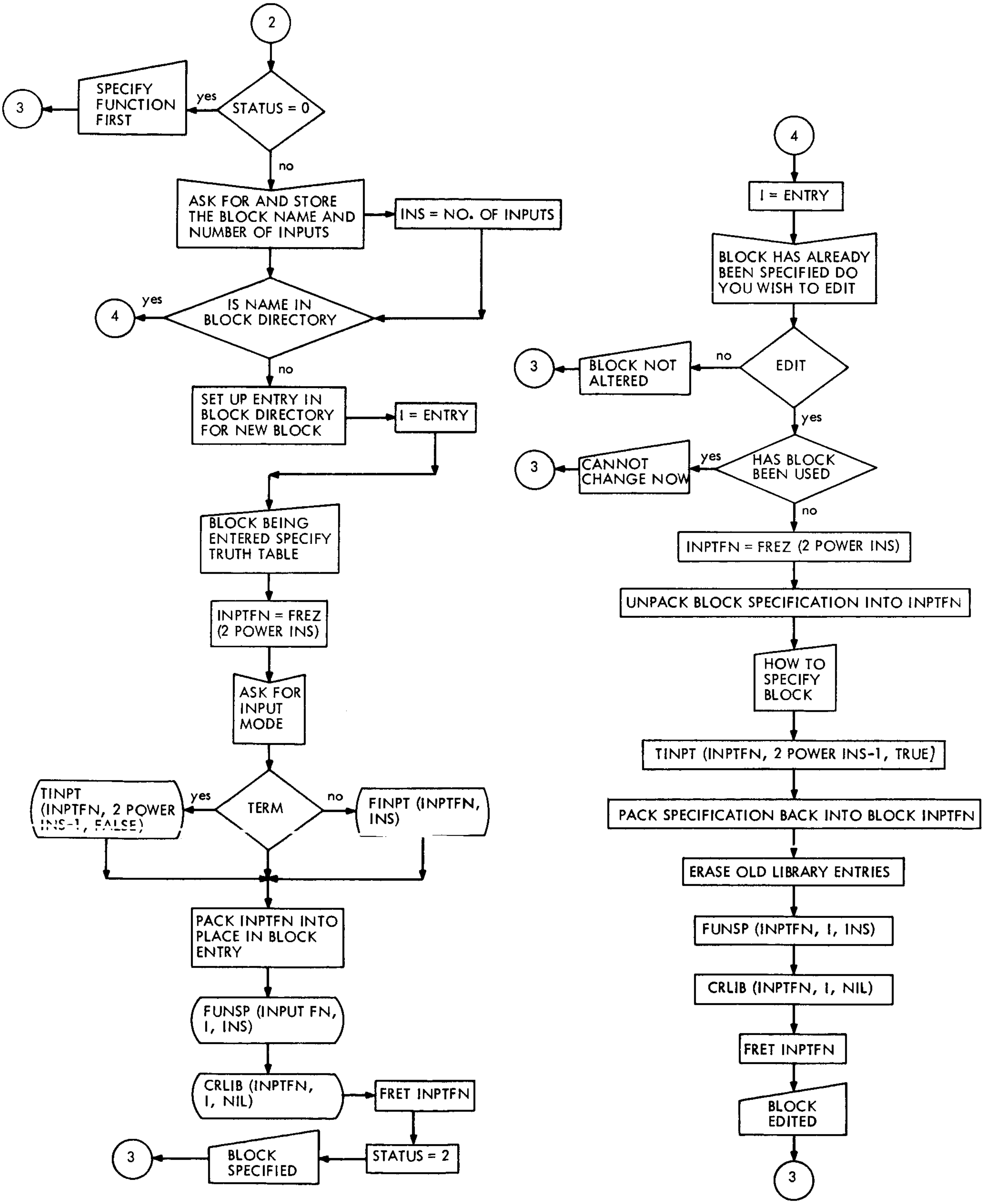
CONVENTIONS

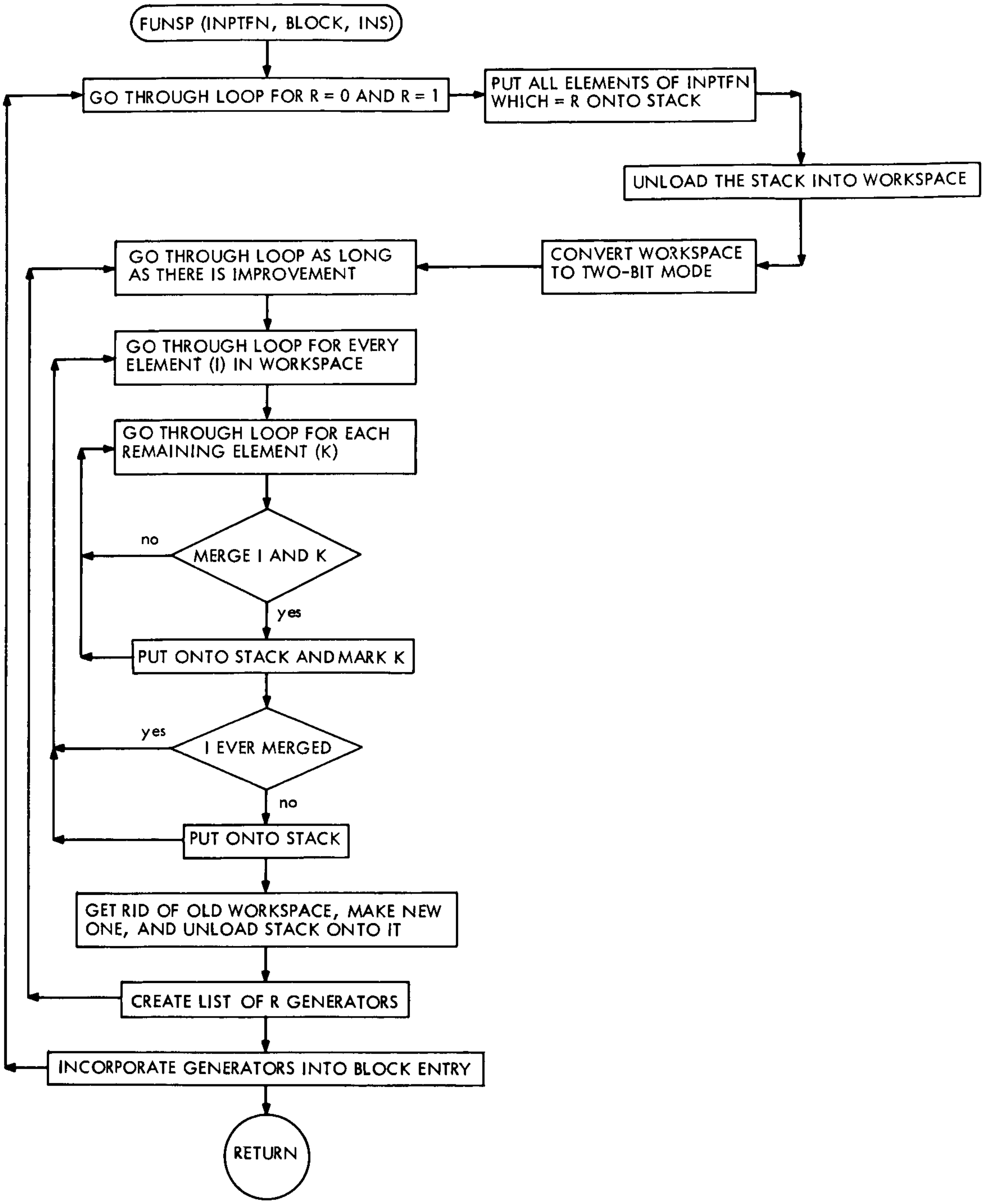


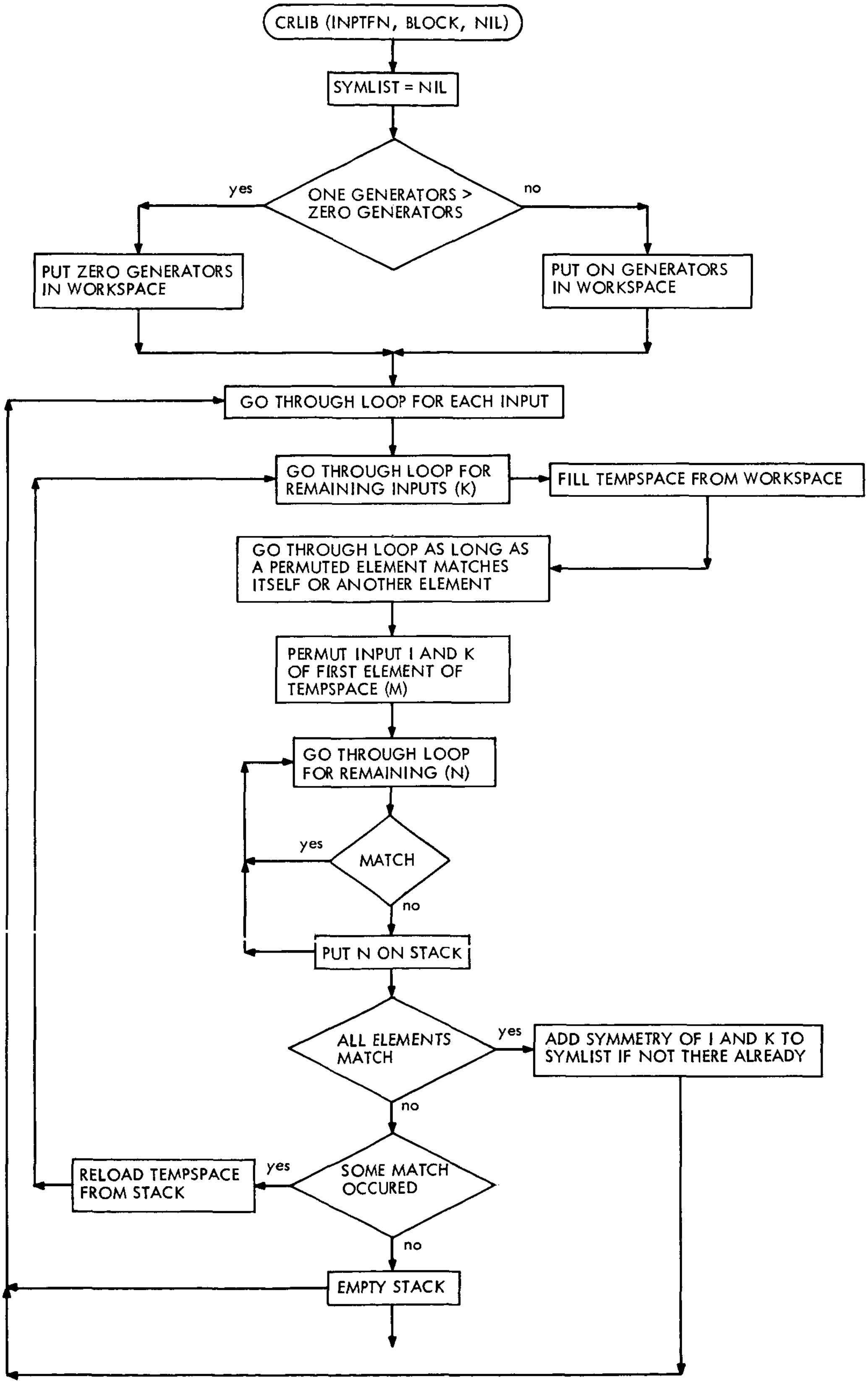

MASTER
INITIALIZE VARIABLES IN COMMON
CLEAR INPUT BUFFER
TYPE COMMAND
INPUT
yes
INP
no
MFPIC
yes
MFP
no
INFOR
yes
INF
no
SFPIC
yes
SFP
no
DELET
yes
DEL
no
TERMN
yes
TER
no
COMAND
yes
ALL COMMANDS
no
NOT A COMMAND

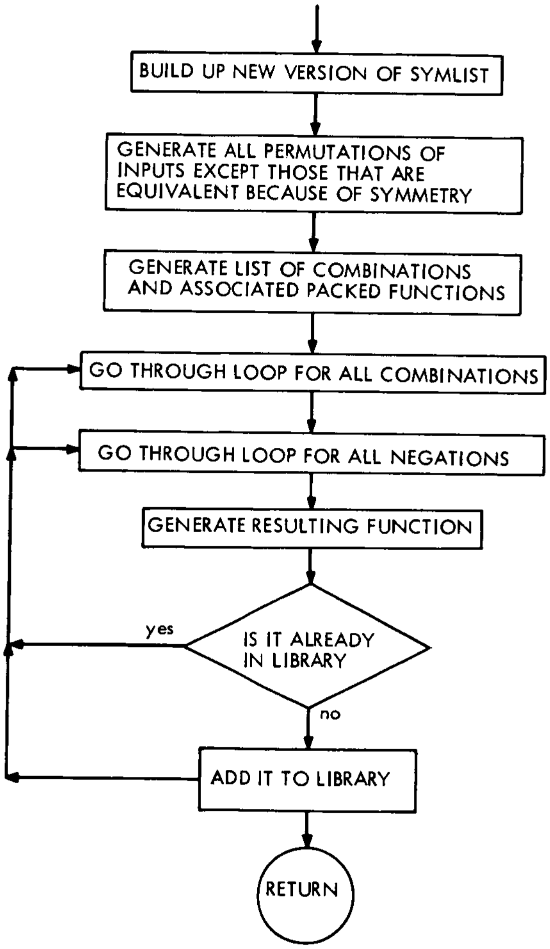

INP
CLEAR INPUT BUFFER
TYPE COMMAND
COMAND
yes
ALL COMMANDS
no
FINIS
yes
RETURNING
RETURN
no
OUTFN
yes
1
no
3
BIL DB
yes
2
no
NOT A COMMAND

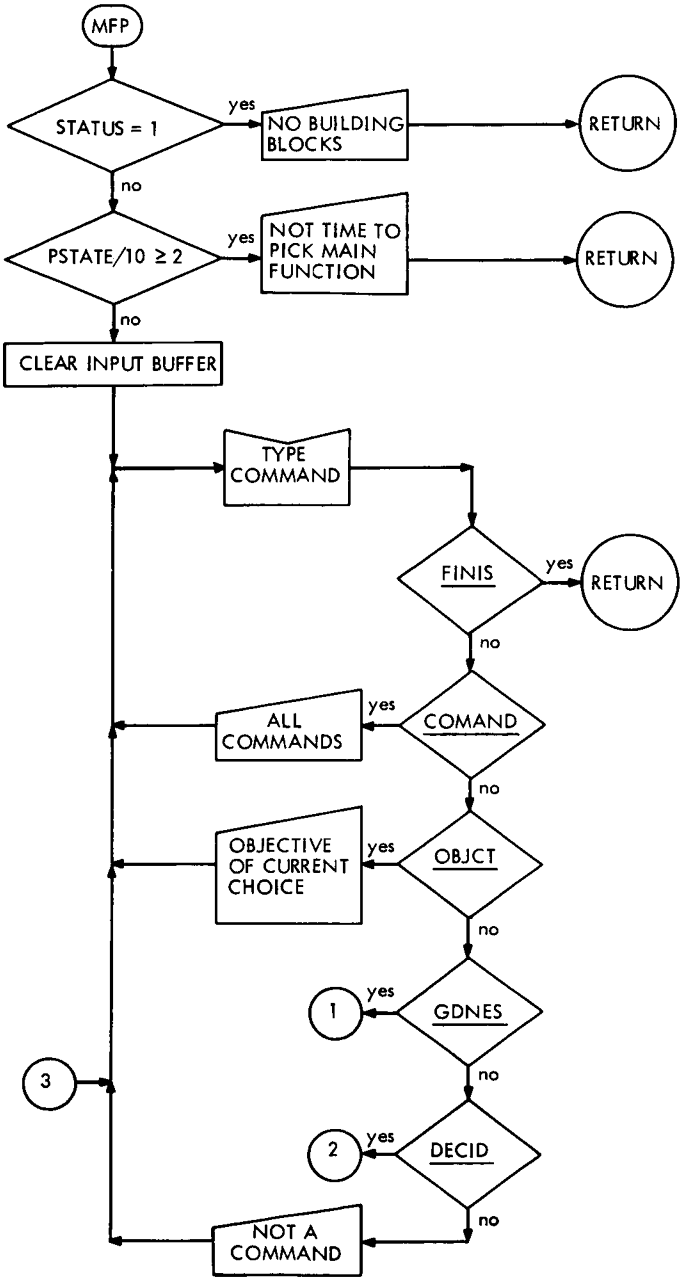

1
STATUS ≠ 0
yes
no
STATUS = 3
yes
no
CANNOT CHANGE FUNCTION
3
EDIT FUNCTION
HOW MANY VARIABLES
STORE IN INVARS
INPTVARBS = FREZ (INVARS)
WHAT ARE VARIABLES NAMES
STORE NAMES IN INPTVARBS
HOW FUNCTION IS SPECIFIED
INPTFN = FREZ (2 POWER INVARS)
STATUS = 0
no
yes
UNPACK MFN INTO INPTFN
TINPT (INPTFN, 2 POWER INVARS-1, TRUE)
PACK INPTFN BACK INTO MFN
FRET INPTFN
FUNCTION EDITED
3
ASK FOR INPUT MODE
TERM
no
yes
FINPT (INPTFN, INVARS)
TINPT (INPTFN, 2 POWER INVARS-1 FALSE
SET UP FUNCTION SPECIFICATION USING INPTFN
FRET INPTFN
STACK ONTO MFN
FUNCTION SPECIFIED
STATUS = 1
3

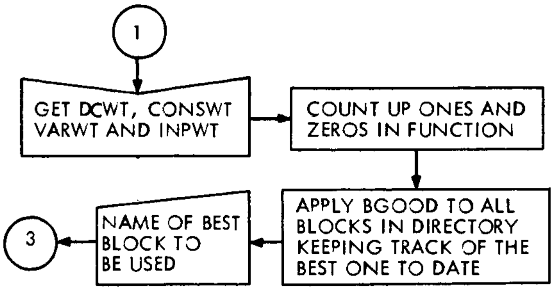

2
STATUS = 0
yes
SPECIFY FUNCTION FIRST
3
no
ASK FOR AND STORE THE BLOCK NAME AND NUMBER OF INPUTS
INS = NO. OF INPUTS
IS NAME IN BLOCK DIRECTORY
yes
4
no
SET UP ENTRY IN BLOCK DIRECTORY FOR NEW BLOCK
I = ENTRY
BLOCK BEING ENTERED SPECIFY TRUTH TABLE
INPTFN = FREZ (2 POWER INS)
ASK FOR INPUT MODE
TERM
yes
TINPT (INPTFN, 2 POWER INS-1, FALSE)
no
FINPT (INPTFN, INS)
PACK INPTFN INTO PLACE IN BLOCK ENTRY
FUNSP (INPUT FN, I, INS)
CRLIB (INPTFN, I, NIL)
FRET INPTFN
STATUS = 2
BLOCK SPECIFIED
3
4
I = ENTRY
BLOCK HAS ALREADY BEEN SPECIFIED DO YOU WISH TO EDIT
EDIT
no
BLOCK NOT ALTERED
3
yes
HAS BLOCK BEEN USED
yes
CANNOT CHANGE NOW
3
no
INPTFN = FREZ (2 POWER INS)
UNPACK BLOCK SPECIFICATION INTO INPTFN
HOW TO SPECIFY BLOCK
TINPT (INPTFN, 2 POWER INS-1, TRUE)
PACK SPECIFICATION BACK INTO BLOCK INPTFN
ERASE OLD LIBRARY ENTRIES
FUNSP (INPTFN, I, INS)
CRLIB (INPTFN, I, NIL)
FRET INPTFN
BLOCK EDITED
3

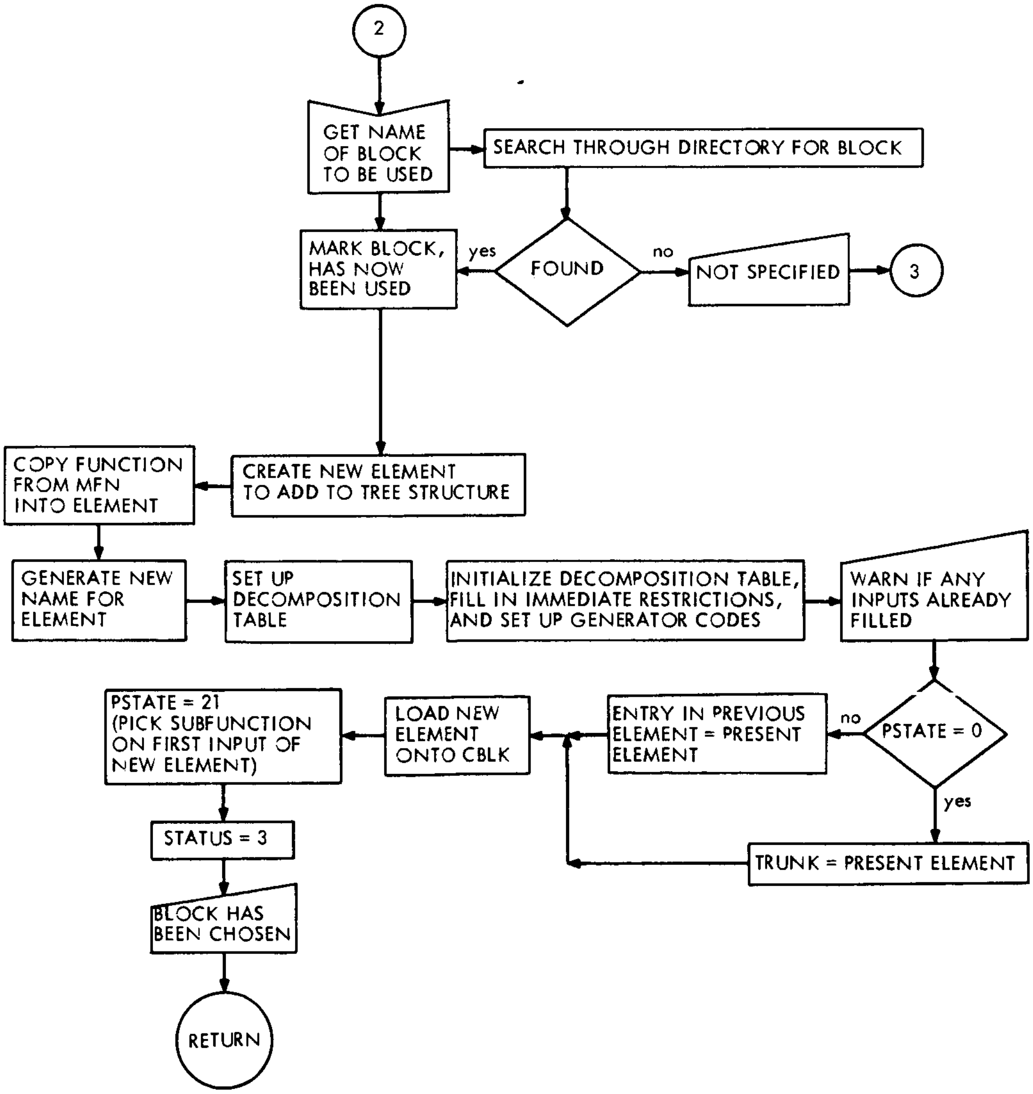

FUNSP (INPTFN, BLOCK, INS)
GO THROUGH LOOP FOR R = 0 AND R = 1
PUT ALL ELEMENTS OF INPTFN WHICH = R ONTO STACK
UNLOAD THE STACK INTO WORKSPACE
CONVERT WORKSPACE TO TWO-BIT MODE
GO THROUGH LOOP AS LONG AS THERE IS IMPROVEMENT
GO THROUGH LOOP FOR EVERY ELEMENT (I) IN WORKSPACE
GO THROUGH LOOP FOR EACH REMAINING ELEMENT (K)
MERGE I AND K
no
yes
PUT ONTO STACK AND MARK K
I EVER MERGED
yes
no
PUT ONTO STACK
GET RID OF OLD WORKSPACE, MAKE NEW ONE, AND UNLOAD STACK ONTO IT
CREATE LIST OF R GENERATORS
INCORPORATE GENERATORS INTO BLOCK ENTRY
RETURN

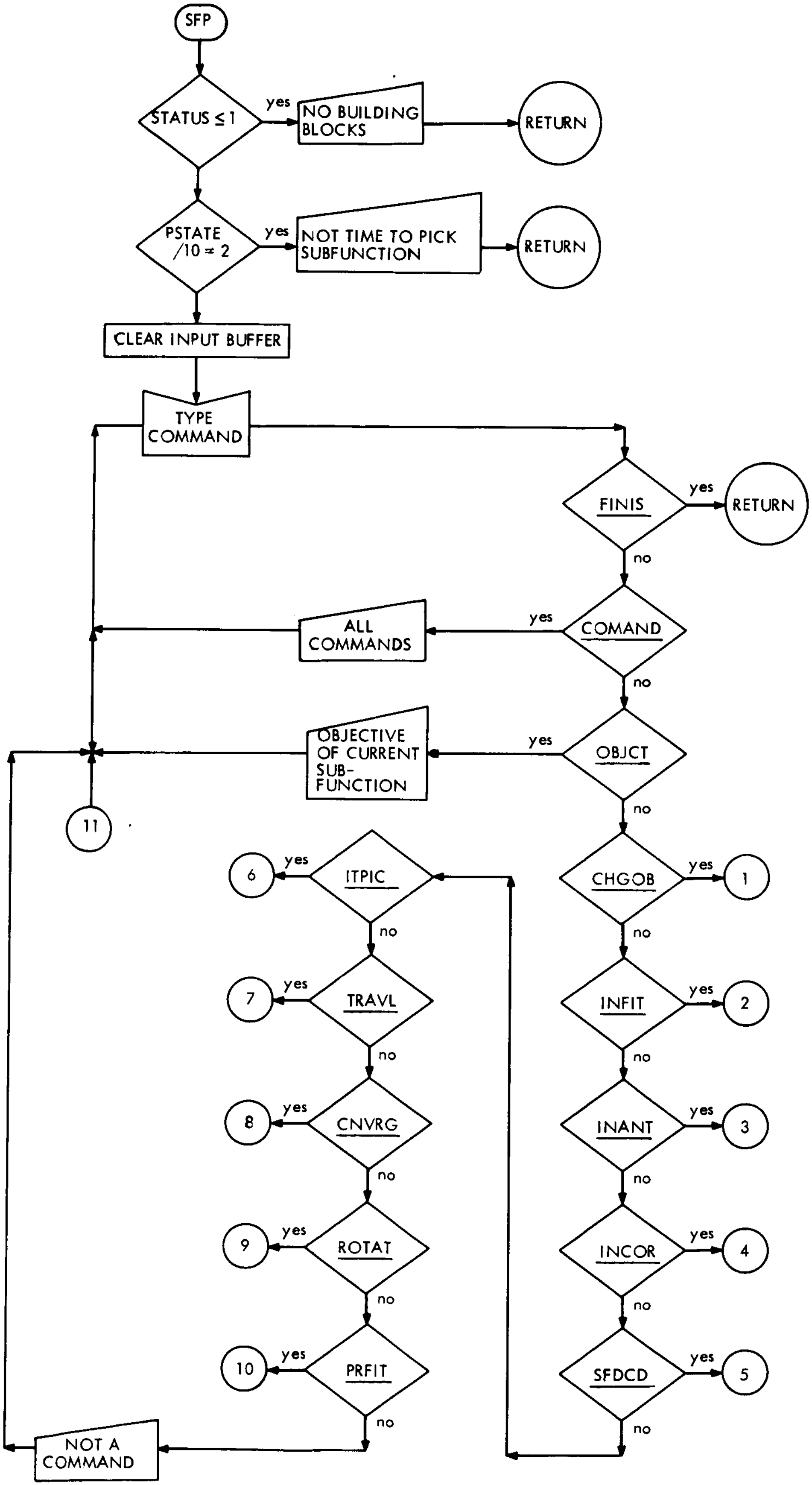

CRLIB (INPTFN, BLOCK, NIL)
SYMLIST = NIL
ONE GENERATORS > ZERO GENERATORS
yes
no
PUT ZERO GENERATORS IN WORKSPACE
PUT ON GENERATORS IN WORKSPACE
GO THROUGH LOOP FOR EACH INPUT
GO THROUGH LOOP FOR REMAINING INPUTS (K)
FILL TEMPSPACE FROM WORKSPACE
GO THROUGH LOOP AS LONG AS A PERMUTED ELEMENT MATCHES ITSELF OR ANOTHER ELEMENT
PERMUT INPUT I AND K OF FIRST ELEMENT OF TEMPSPACE (M)
GO THROUGH LOOP FOR REMAINING (N)
yes
MATCH
no
PUT N ON STACK
ALL ELEMENTS MATCH
yes
ADD SYMMETRY OF I AND K TO SYMLIST IF NOT THERE ALREADY
no
RELOAD TEMPSPACE FROM STACK
yes
SOME MATCH OCCURED
no
EMPTY STACK

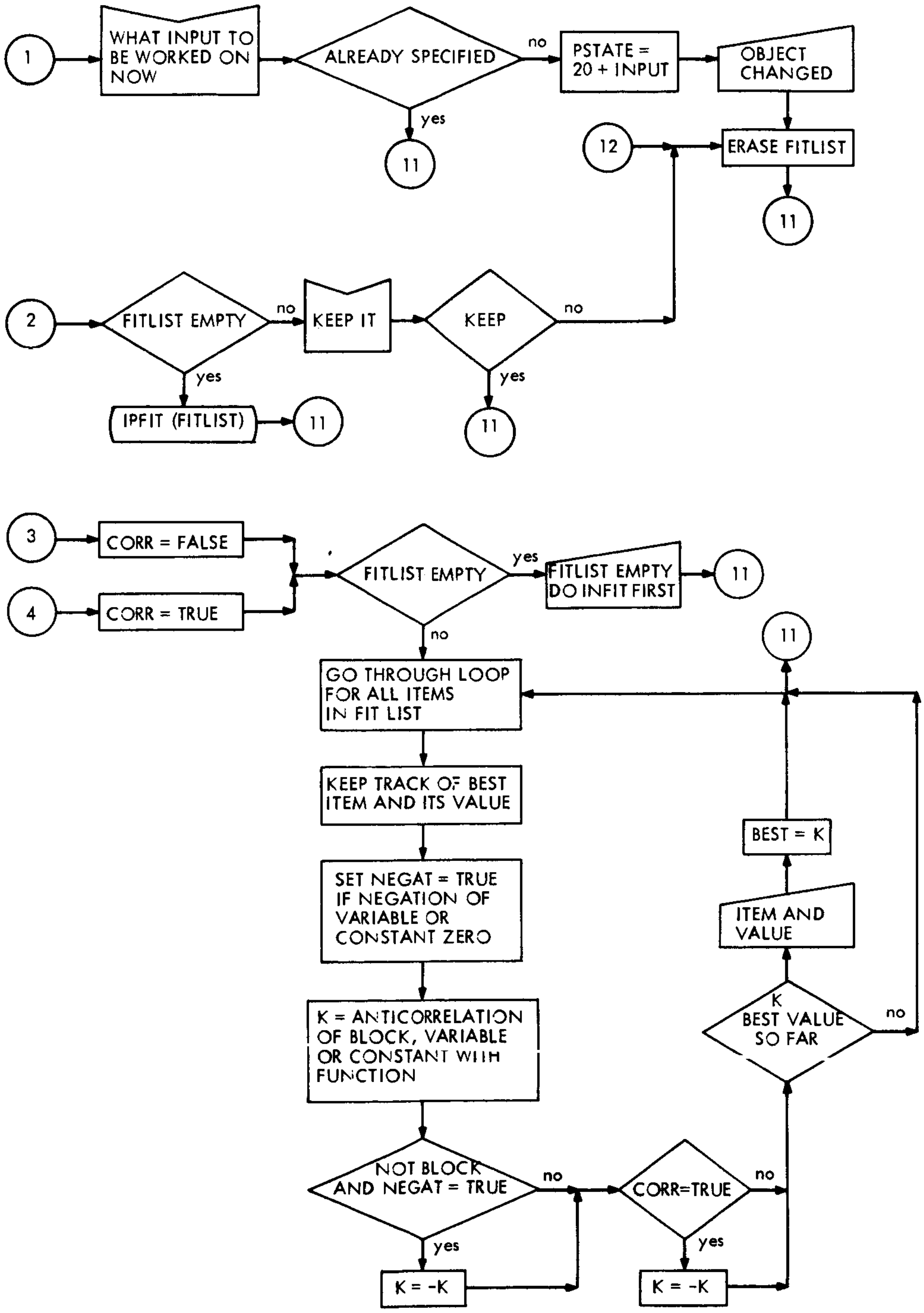

BUILD UP NEW VERSION OF SYMLIST
GENERATE ALL PERMUTATIONS OF INPUTS EXCEPT THOSE THAT ARE EQUIVALENT BECAUSE OF SYMMETRY
GENERATE LIST OF COMBINATIONS AND ASSOCIATED PACKED FUNCTIONS
GO THROUGH LOOP FOR ALL COMBINATIONS
GO THROUGH LOOP FOR ALL NEGATIONS
GENERATE RESULTING FUNCTION
IS IT ALREADY IN LIBRARY
yes
no
ADD IT TO LIBRARY
RETURN

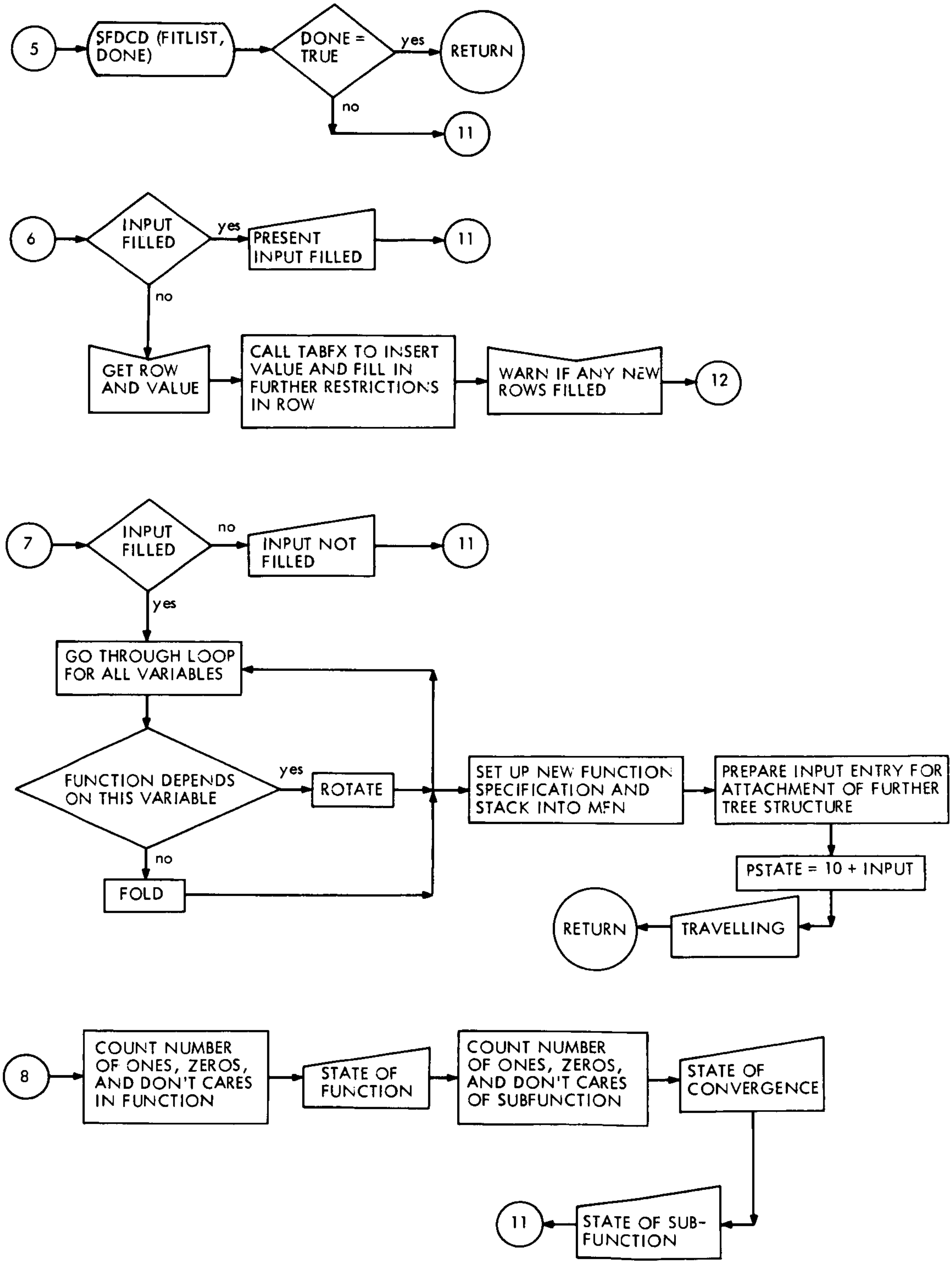

MFP
STATUS = 1
yes
NO BUILDING BLOCKS
RETURN
no
PSTATE/10 ≥ 2
yes
NOT TIME TO PICK MAIN FUNCTION
RETURN
no
CLEAR INPUT BUFFER
TYPE COMMAND
FINIS
yes
RETURN
no
COMAND
yes
ALL COMMANDS
no
OBJCT
yes
OBJECTIVE OF CURRENT CHOICE
no
GDNES
yes
1
no
3
DECID
yes
2
no
NOT A COMMAND

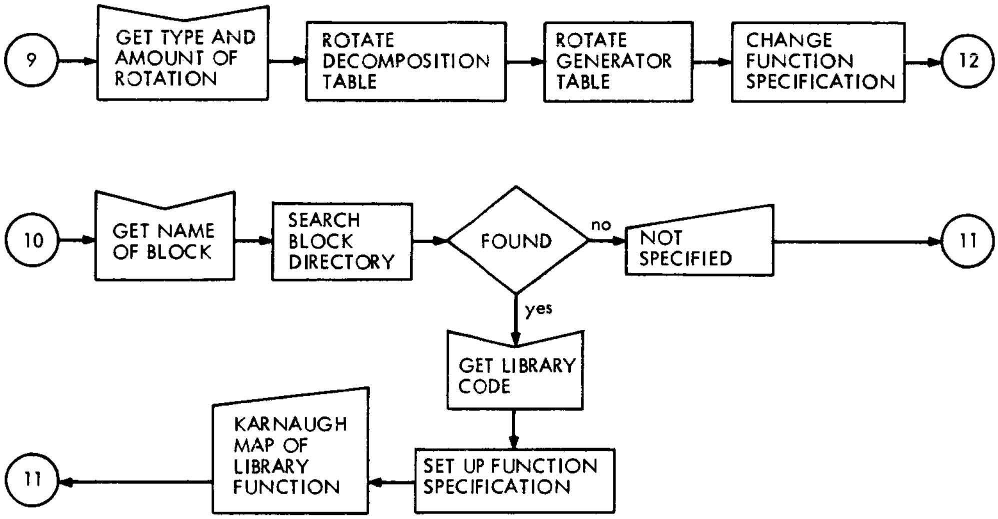

1
GET DCWT, CONSWT VARWT AND INPWT
COUNT UP ONES AND ZEROS IN FUNCTION
APPLY BGOOD TO ALL BLOCKS IN DIRECTORY KEEPING TRACK OF THE BEST ONE TO DATE
NAME OF BEST BLOCK TO BE USED
3

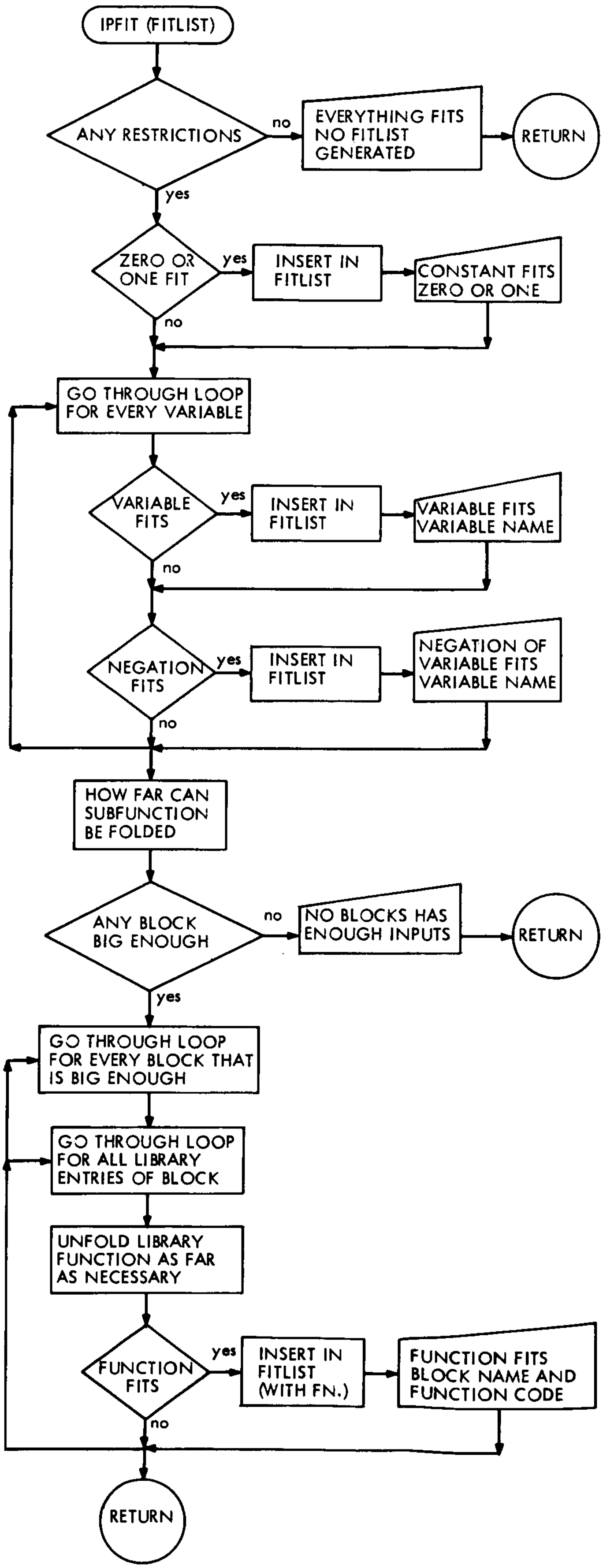

2
GET NAME OF BLOCK TO BE USED
SEARCH THROUGH DIRECTORY FOR BLOCK
FOUND
yes
no
MARK BLOCK, HAS NOW BEEN USED
NOT SPECIFIED
3
CREATE NEW ELEMENT TO ADD TO TREE STRUCTURE
COPY FUNCTION FROM MFN INTO ELEMENT
GENERATE NEW NAME FOR ELEMENT
SET UP DECOMPOSITION TABLE
INITIALIZE DECOMPOSITION TABLE, FILL IN IMMEDIATE RESTRICTIONS, AND SET UP GENERATOR CODES
WARN IF ANY INPUTS ALREADY FILLED
PSTATE = 0
no
yes
ENTRY IN PREVIOUS ELEMENT = PRESENT ELEMENT
TRUNK = PRESENT ELEMENT
LOAD NEW ELEMENT ONTO CBLK
PSTATE = 21 (PICK SUBFUNCTION ON FIRST INPUT OF NEW ELEMENT)
STATUS = 3
BLOCK HAS BEEN CHOSEN
RETURN

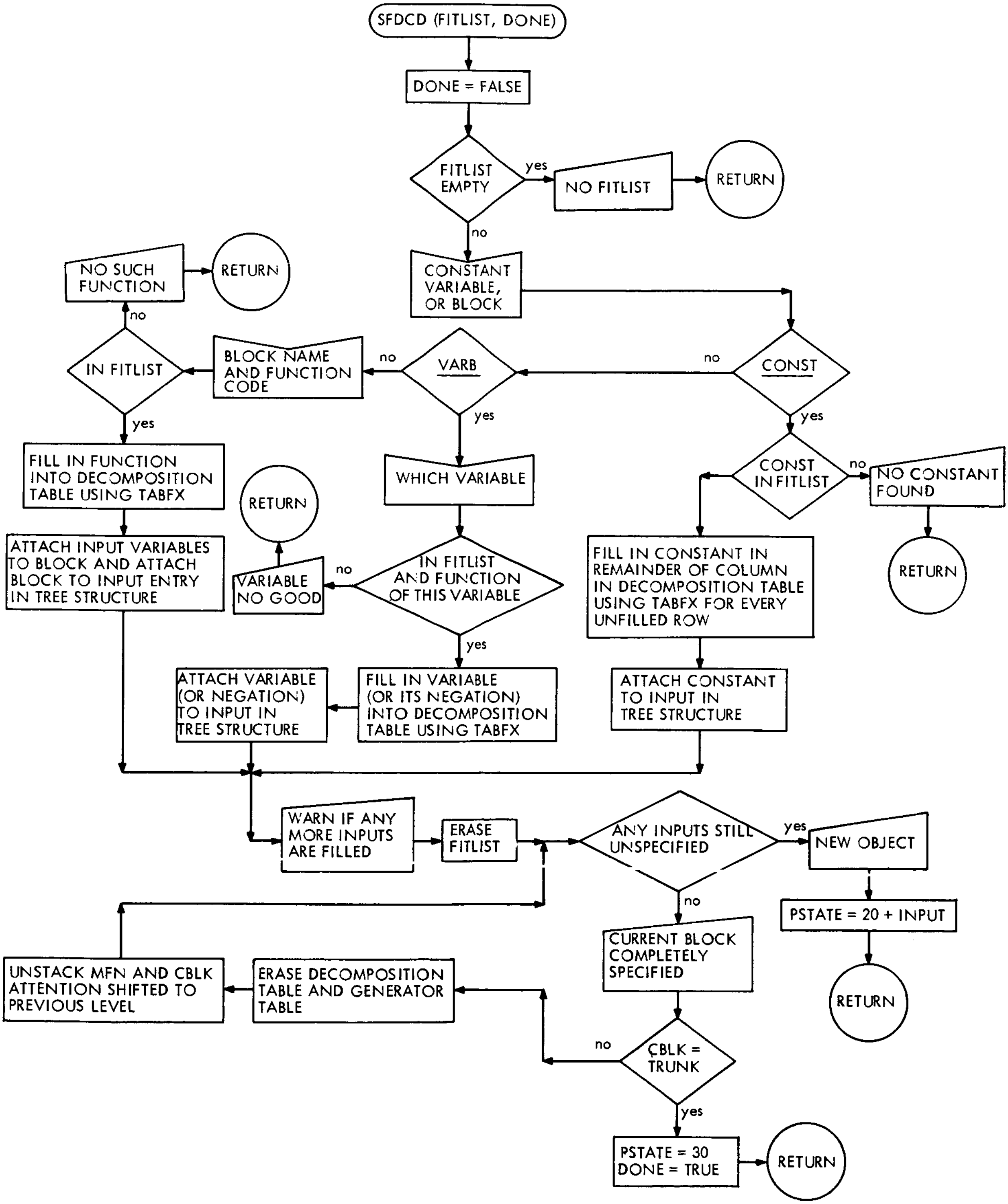

SFP
STATUS ≤ 1
yes
NO BUILDING BLOCKS
RETURN
PSTATE /10 = 2
yes
NOT TIME TO PICK SUBFUNCTION
RETURN
CLEAR INPUT BUFFER
TYPE COMMAND
FINIS
yes
RETURN
no
COMAND
yes
ALL COMMANDS
no
OBJCT
yes
OBJECTIVE OF CURRENT SUB-FUNCTION
no
11
CHGOB
yes
1
no
INFIT
yes
2
no
INANT
yes
3
no
INCOR
yes
4
no
SFDCD
yes
5
no
ITPIC
yes
6
no
TRAVL
yes
7
no
CNVRG
yes
8
no
ROTAT
yes
9
no
PRFIT
yes
10
no
NOT A COMMAND

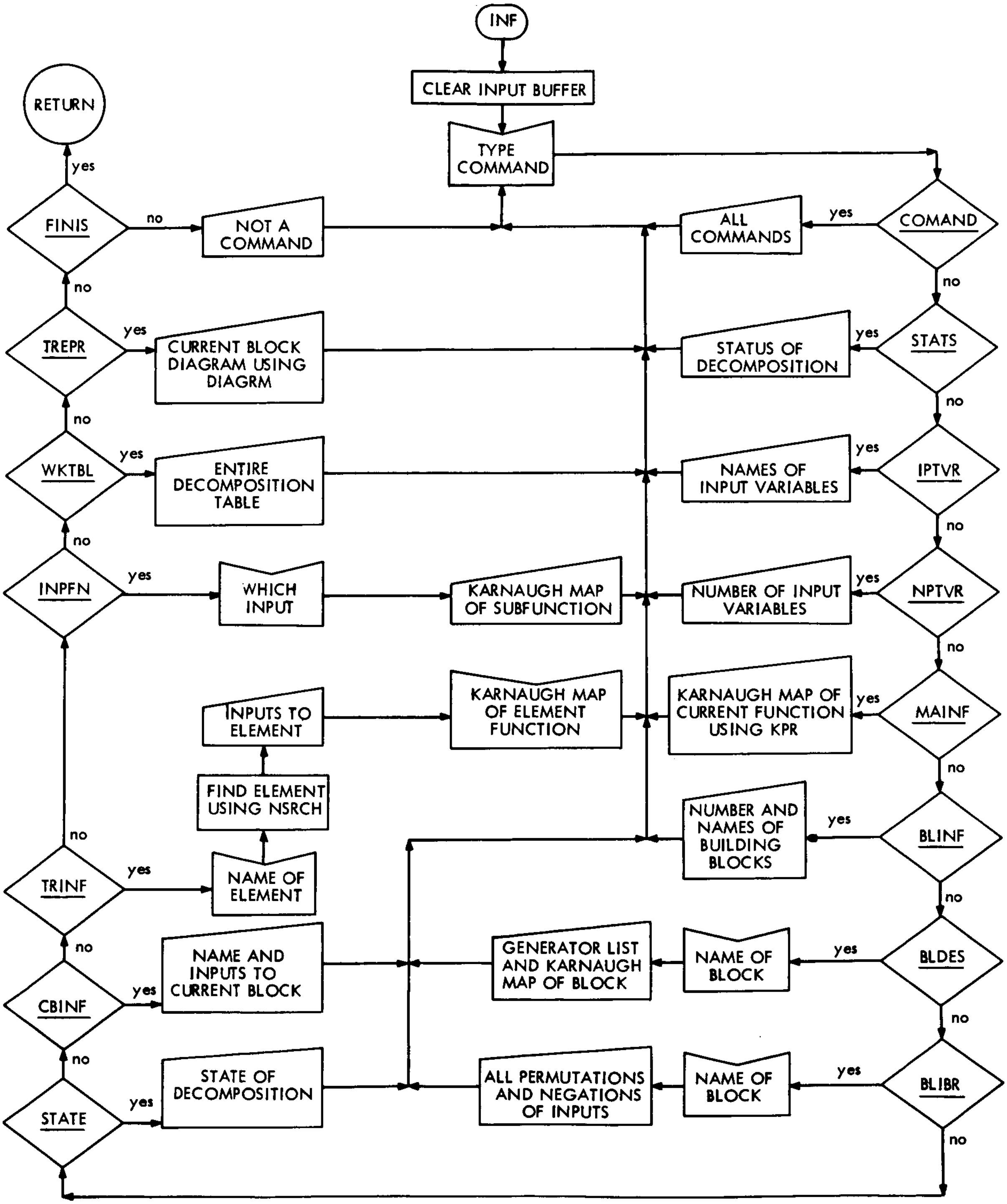

1
WHAT INPUT TO BE WORKED ON NOW
ALREADY SPECIFIED
no
PSTATE = 20 + INPUT
OBJECT CHANGED
yes
11
12
ERASE FITLIST
11
2
FITLIST EMPTY
no
KEEP IT
KEEP
no
yes
IPFIT (FITLIST)
11
yes
11
3
CORR = FALSE
4
CORR = TRUE
FITLIST EMPTY
yes
FITLIST EMPTY DO INFIT FIRST
11
no
11
GO THROUGH LOOP FOR ALL ITEMS IN FIT LIST
KEEP TRACK OF BEST ITEM AND ITS VALUE
SET NEGAT = TRUE IF NEGATION OF VARIABLE OR CONSTANT ZERO
K = ANTICORRELATION OF BLOCK, VARIABLE OR CONSTANT WITH FUNCTION
BEST = K
ITEM AND VALUE
K BEST VALUE SO FAR
no
NOT BLOCK AND NEGAT = TRUE
no
CORR=TRUE
no
yes
K = -K
yes
K = -K

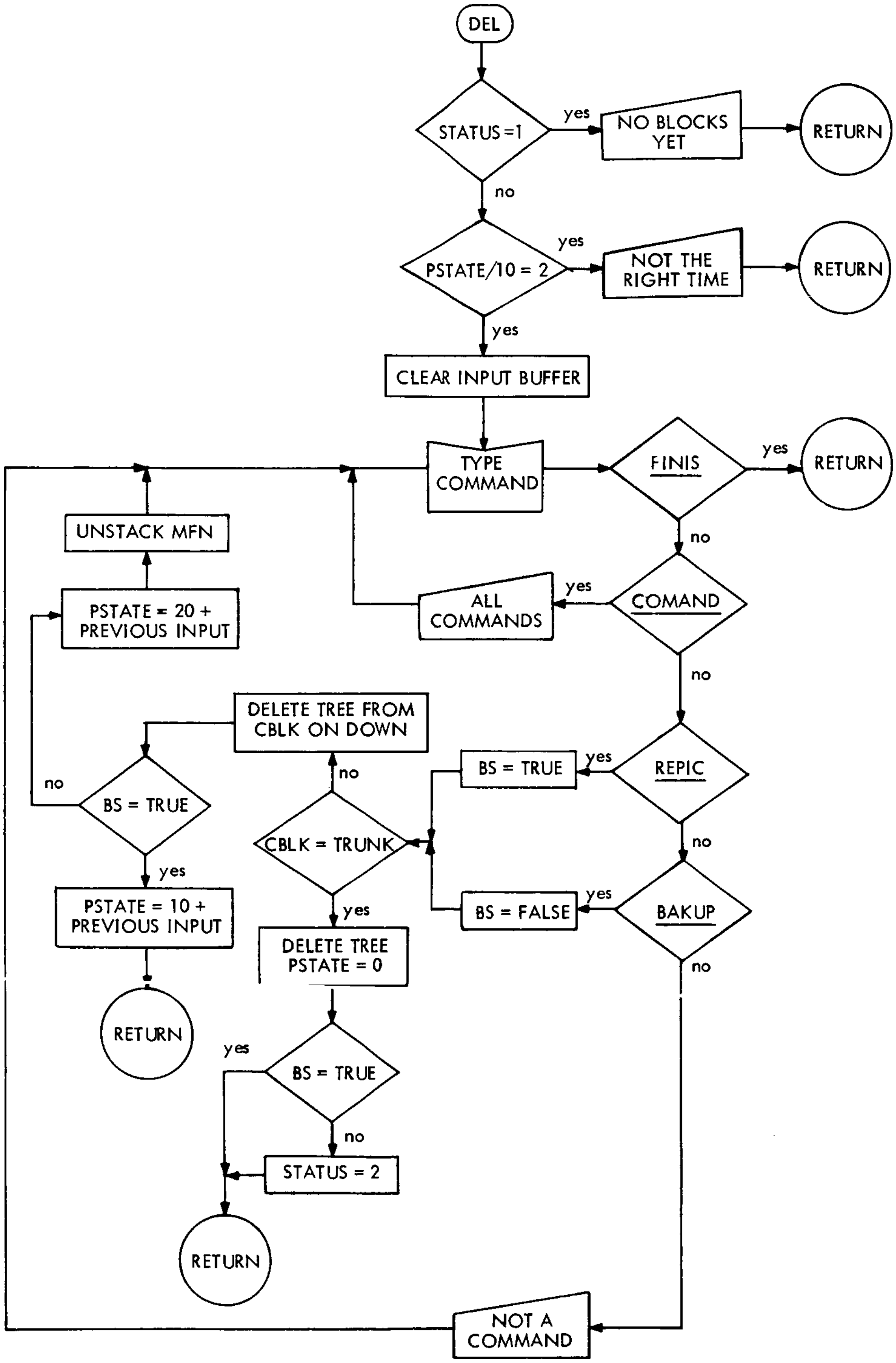

5
SFDCD (FITLIST, DONE)
DONE = TRUE
yes
RETURN
no
11
6
INPUT FILLED
yes
PRESENT INPUT FILLED
11
no
GET ROW AND VALUE
CALL TABFX TO INSERT VALUE AND FILL IN FURTHER RESTRICTIONS IN ROW
WARN IF ANY NEW ROWS FILLED
12
7
INPUT FILLED
no
INPUT NOT FILLED
11
yes
GO THROUGH LOOP FOR ALL VARIABLES
FUNCTION DEPENDS ON THIS VARIABLE
yes
ROTATE
no
FOLD
SET UP NEW FUNCTION SPECIFICATION AND STACK INTO MFN
PREPARE INPUT ENTRY FOR ATTACHMENT OF FURTHER TREE STRUCTURE
PSTATE = 10 + INPUT
TRAVELLING
RETURN
8
COUNT NUMBER OF ONES, ZEROS, AND DON'T CARES IN FUNCTION
STATE OF FUNCTION
COUNT NUMBER OF ONES, ZEROS, AND DON'T CARES OF SUBFUNCTION
STATE OF CONVERGENCE
STATE OF SUB-FUNCTION
11


9
GET TYPE AND AMOUNT OF ROTATION
ROTATE DECOMPOSITION TABLE
ROTATE GENERATOR TABLE
CHANGE FUNCTION SPECIFICATION
12
10
GET NAME OF BLOCK
SEARCH BLOCK DIRECTORY
FOUND
no
NOT SPECIFIED
11
yes
GET LIBRARY CODE
SET UP FUNCTION SPECIFICATION
KARNAUGH MAP OF LIBRARY FUNCTION
11


IPFIT (FITLIST)
ANY RESTRICTIONS
no
EVERYTHING FITS NO FITLIST GENERATED
RETURN
yes
ZERO OR ONE FIT
yes
INSERT IN FITLIST
CONSTANT FITS ZERO OR ONE
no
GO THROUGH LOOP FOR EVERY VARIABLE
VARIABLE FITS
yes
INSERT IN FITLIST
VARIABLE FITS VARIABLE NAME
no
NEGATION FITS
yes
INSERT IN FITLIST
NEGATION OF VARIABLE FITS VARIABLE NAME
no
HOW FAR CAN SUBFUNCTION BE FOLDED
ANY BLOCK BIG ENOUGH
no
NO BLOCKS HAS ENOUGH INPUTS
RETURN
yes
GO THROUGH LOOP FOR EVERY BLOCK THAT IS BIG ENOUGH
GO THROUGH LOOP FOR ALL LIBRARY ENTRIES OF BLOCK
UNFOLD LIBRARY FUNCTION AS FAR AS NECESSARY
FUNCTION FITS
yes
INSERT IN FITLIST (WITH FN.)
FUNCTION FITS BLOCK NAME AND FUNCTION CODE
no
RETURN


SFDCD (FITLIST, DONE)
DONE = FALSE
FITLIST EMPTY
yes
NO FITLIST
RETURN
no
CONSTANT VARIABLE, OR BLOCK
NO SUCH FUNCTION
RETURN
no
IN FITLIST
BLOCK NAME AND FUNCTION CODE
no
VARB
no
CONST
yes
yes
FILL IN FUNCTION INTO DECOMPOSITION TABLE USING TABFX
WHICH VARIABLE
CONST INFITLIST
no
NO CONSTANT FOUND
RETURN
ATTACH INPUT VARIABLES TO BLOCK AND ATTACH BLOCK TO INPUT ENTRY IN TREE STRUCTURE
VARIABLE NO GOOD
no
IN FITLIST AND FUNCTION OF THIS VARIABLE
FILL IN CONSTANT IN REMAINDER OF COLUMN IN DECOMPOSITION TABLE USING TABFX FOR EVERY UNFILLED ROW
RETURN
yes
ATTACH VARIABLE (OR NEGATION) TO INPUT IN TREE STRUCTURE
FILL IN VARIABLE (OR ITS NEGATION) INTO DECOMPOSITION TABLE USING TABFX
ATTACH CONSTANT TO INPUT IN TREE STRUCTURE
WARN IF ANY MORE INPUTS ARE FILLED
ERASE FITLIST
ANY INPUTS STILL UNSPECIFIED
yes
NEW OBJECT
no
PSTATE = 20 + INPUT
CURRENT BLOCK COMPLETELY SPECIFIED
RETURN
UNSTACK MFN AND CBLK ATTENTION SHIFTED TO PREVIOUS LEVEL
ERASE DECOMPOSITION TABLE AND GENERATOR TABLE
no
CBLK = TRUNK
yes
PSTATE = 30
DONE = TRUE
RETURN


INF
CLEAR INPUT BUFFER
TYPE COMMAND
RETURN
yes
FINIS
no
NOT A COMMAND
ALL COMMANDS
yes
COMAND
no
TREPR
yes
CURRENT BLOCK DIAGRAM USING DIAGRM
STATUS OF DECOMPOSITION
STATS
WKTBL
ENTIRE DECOMPOSITION TABLE
NAMES OF INPUT VARIABLES
IPTVR
INPFN
WHICH INPUT
KARNAUGH MAP OF SUBFUNCTION
NUMBER OF INPUT VARIABLES
NPTVR
INPUTS TO ELEMENT
KARNAUGH MAP OF ELEMENT FUNCTION
KARNAUGH MAP OF CURRENT FUNCTION USING KPR
MAINF
FIND ELEMENT USING NSRCH
NUMBER AND NAMES OF BUILDING BLOCKS
BLINF
TRINF
NAME OF ELEMENT
NAME AND INPUTS TO CURRENT BLOCK
CBINF
GENERATOR LIST AND KARNAUGH MAP OF BLOCK
NAME OF BLOCK
BLDES
STATE OF DECOMPOSITION
STATE
ALL PERMUTATIONS AND NEGATIONS OF INPUTS
NAME OF BLOCK
BLIBR


DEL
STATUS = 1
yes
NO BLOCKS YET
RETURN
no
PSTATE/10 = 2
yes
NOT THE RIGHT TIME
RETURN
yes
CLEAR INPUT BUFFER
TYPE COMMAND
FINIS
yes
RETURN
no
COMAND
yes
ALL COMMANDS
no
REPIC
yes
BS = TRUE
no
BAKUP
yes
BS = FALSE
no
NOT A COMMAND
UNSTACK MFN
PSTATE = 20 + PREVIOUS INPUT
DELETE TREE FROM CBLK ON DOWN
no
BS = TRUE
yes
PSTATE = 10 + PREVIOUS INPUT
RETURN
no
CBLK = TRUNK
yes
DELETE TREE PSTATE = 0
BS = TRUE
yes
no
STATUS = 2
RETURN

APPENDIX C

SAMPLE RUN FOR CADD-1

The following sample run is the basis for Case IX in Table 1. Upper case characters indicate output, lower case input. In function specifications, "2" stands for DON'T CARE.

```
r synths
W
TYPE. input
INPUTS WILL NOW BE ACCEPTED
TYPE. outfn
SPECIFICATION OF OUTPUT FUNCTION
HOW MANY INPUT VARIABLES
TYPE. 3
WHAT ARE THE VARIABLE NAMES
NO MORE THAN SIX CHARACTERS PER NAME PLEASE
TYPE. alpha beta gamma
PROCEED WITH FUNCTION SPECIFICATION
THE FUNCTION IS SPECIFIED BY INDICATING THE VALUE
( 0 = ZERO, 1 = ONE, 2 = DON'T CARE )
FOR EACH COMBINATION OF INPUT VARIABLE VALUES.  A GIVEN SET
OF INPUT VARIABLE VALUES IS TRANSFORMED INTO A 'TERM' BY
MULTIPLYING
GAMMA  BY    1
BETA   BY    2
ALPHA  BY    4
AND ADDING
TYPE 'TERM' FOR SPECIAL INPUT MODE, OR 'FULL' OTHERWISE
TYPE. term
TYPE INITIAL VALUE OF ALL ELEMENTS
TYPE. 0
SPECIFY VALUES OF INDIVIDUAL TERMS AS FOLLOWS
V1 T1 T2 T3 . . . * V2 T1 T2 . . . * *
TYPE. 1 0 2 5 * *
OUTPUT FUNCTION ENTERED
TYPE. bildb
WHAT IS THE NAME OF THE BUILDING  BLOCK
NO MORE THAN THREE CHARACTERS PLEASE
TYPE. nor
HOW MANY INPUTS
TYPE. 2
2IPNOR BEING ENTERED
SPECIFY FUNCTION.  ONLY ZEROS AND ONES ALLOWED
THE FUNCTION IS SPECIFIED BY INDICATING THE VALUE
FOR EACH COMBINATION OF INPUT VARIABLE VALUES.  A GIVEN SET
OF INPUT VARIABLE VALUES IS TRANSFORMED INTO A 'TERM' BY
MULTIPLYING
INPUT1 BY    1
INPUT2 BY    2
AND ADDING
TYPE 'TERM' FOR SPECIAL INPUT MODE, OR 'FULL' OTHERWISE
TYPE. term
TYPE INITIAL VALUE OF ALL ELEMENTS
TYPE. 0
SPECIFY VALUES OF INDIVIDUAL TERMS AS FOLLOWS
V1 T1 T2 T3 . . . * V2 T1 T2 . . . * *
TYPE. 1 0 * *
BUILDING BLOCK SPECIFIED AND ENTERED IN LIBRARY
TYPE. finis infor
```

```
INPUTS WILL NO LONGER BE ACCEPTED.  RETURN TO TOP LEVEL
INFORMATION NOW AVAILABLE
TYPE. mainf
THE MAIN FUNCTION IS
        BETA   GAMMA
                 00  01  11  10
               00 1   0   0   1
        ALPHA
               01 0   1   0   0
TYPE. finis mfpic
INFORMATION NO LONGER AVAILABLE. RETURN TO TOP LEVEL
READY TO PICK MAIN FUNCTION
TYPE. decid
WHAT IS THE TYPE OF BLOCK TO BE USED
TYPE. 2ipnor
MAIN FUNCTION BLOCK HAS BEEN CHOSEN. RETURN TO TOP LEVEL
TYPE. infor
INFORMATION NOW AVAILABLE
TYPE. wktbl
THE DECOMPOSITION TABLE FOR THE PRESENT BLOCK IS
ALPHA   BETA    GAMMA
     TERM  VALUE  IP1  IP2  CHOICES
      000      1    0    0   1
      001      0             1 - 2
      010      1    0    0   1
      011      0             1 - 2
      100      0             1 - 2
      101      1    0    0   1
      110      0             1 - 2
      111      0             1 - 2
TYPE. finis sfpic
INFORMATION NO LONGER AVAILABLE. RETURN TO TOP LEVEL
READY TO PICK SUBFUNCTION
TYPE. infit
CONSTANT ZERO
FITTING IS LIMITED TO BUILDING BLOCKS WITH 0 OR MORE INPUTS
2IPNOR CODE   1  INPUT2 = NOT GAMMA   INPUT1 = NOT BETA
TYPE. inant
2IPNOR CODE   1  ANTICORRELATION FACTOR =     2
TYPE. sfdcd
IS THE SUBFUNCTION A CONST, VARB, OR BLOCK
TYPE. block
WHICH BLOCK
TYPE. 2ipnor
CODE NUMBER
TYPE. 1
OBJECT IS NOW INPUT  2 OF BLOCK NOR000
TYPE. infit
FITTING IS LIMITED TO BUILDING BLOCKS WITH 3 OR MORE INPUTS
NO BLOCK HAS SUFFICIENT INPUTS
TYPE. finis infor
SUBFUNCTION CHOICE DISABLED. RETURN TO TOP LEVEL
INFORMATION NOW AVAILABLE
TYPE. wktbl
```

```
THE DECOMPOSITION TABLE FOR THE PRESENT BLOCK IS
ALPHA   BETA    GAMMA
     TERM  VALUE  IP1  IP2  CHOICES
      000     1     0    0   1
      001     0     0    1   1
      010     1     0    0   1
      011     0     1        1 - 2
      100     0     0    1   1
      101     1     0    0   1
      110     0     0    1   1
      111     0     1        1 - 2
TYPE. finis sfpic
INFORMATION NO LONGER AVAILABLE. RETURN TO TOP LEVEL
READY TO PICK SUBFUNCTION
TYPE. itpic
WHICH ROW
TYPE. 3
WHAT VALUE
TYPE. 2
TYPE. itpic
WHICH ROW
TYPE. 7
WHAT VALUE
TYPE. 2
INPUT  2 FILLED
TYPE. travl mfpic
TRAVELING. RETURN TO TOP LEVEL
READY TO PICK MAIN FUNCTION
TYPE. decid
WHAT IS THE TYPE OF BLOCK TO BE USED
TYPE. 2ipor
2IPOR  HAS NOT BEEN SPECIFIED
TYPE. decid
WHAT IS THE TYPE OF BLOCK TO BE USED
TYPE. 2ipnor
MAIN FUNCTION BLOCK HAS BEEN CHOSEN. RETURN TO TOP LEVEL
TYPE. infor
INFORMATION NOW AVAILABLE
TYPE. wktbl
THE DECOMPOSITION TABLE FOR THE PRESENT BLOCK IS
GAMMA   ALPHA
     TERM  VALUE  IP1  IP2  CHOICES
       00     0              1 - 2
       01     1     0    0   1
       10     1     0    0   1
       11     0              1 - 2
TYPE. finis sfpic
INFORMATION NO LONGER AVAILABLE. RETURN TO TOP LEVEL
READY TO PICK SUBFUNCTION
TYPE. infit
CONSTANT ZERO
FITTING IS LIMITED TO BUILDING BLOCKS WITH 0 OR MORE INPUTS
2IPNOR CODE   1  INPUT2 = NOT ALPHA   INPUT1 = NOT GAMMA
2IPNOR CODE   4  INPUT2 = ALPHA   INPUT1 = GAMMA
TYPE. sfdcd
```

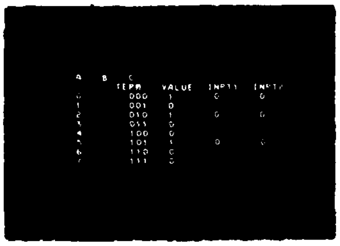
```
IS THE SUBFUNCTION A CONST, VARB, OR BLOCK
TYPE. block
WHICH BLOCK
TYPE. 2ipnor
CODE NUMBER
TYPE. 1
OBJECT IS NOW INPUT  2 OF BLOCK NOR002
TYPE. infit
FITTING IS LIMITED TO BUILDING BLOCKS WITH 2 OR MORE INPUTS
2IPNOR CODE   4   INPUT2 = ALPHA   INPUT1 = GAMMA
TYPE. sfdcd
IS THE SUBFUNCTION A CONST, VARB, OR BLOCK
TYPE. block
WHICH BLOCK
TYPE. 2ipnor
CODE NUMBER
TYPE. 4
NOR002 HAS NOW BEEN COMPLETELY SPECIFIED
NOR000 HAS NOW BEEN COMPLETELY SPECIFIED
DECOMPOSITION DONE. RETURN TO TOP LEVEL
TYPE. infor
INFORMATION NOW AVAILABLE
TYPE. trepr

NOR000
   IP1----NOR001
             IP1----NOT BETA
             IP2----NOT GAMMA

   IP2----NOR002
             IP1----NOR003
                       IP1----NOT GAMMA
                       IP2----NOT ALPHA

             IP2----NOR004
                       IP1----GAMMA
                       IP2----ALPHA


TYPE. finis termn
INFORMATION NO LONGER AVAILABLE. RETURN TO TOP LEVEL
R
```

APPENDIX D

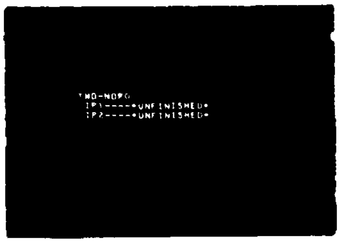
SAMPLE RUN FOR CADD-2

The following sample run covers the same problem as in Appendix C. All displays, with the exception of a few that were redundant are given at the point in the text where they occurred.

```
r synth2
W
EXECUTION.
MASTER COMMAND     input
INPUT COMMAND     outfn
FUNCTION PROGRAM   *
NUMBER AND NAMES OF VARIABLES 3 a b c
MAJOR VALUE AND MINORITY ELEMENTS   0 0 2 5 *
INPUT COMMAND     bildb
BLOCK NAME  two-nor
BLOCK PROGRAM     norip2
 NEED NORIP2
 GIVE LOADING COMMANDS
 TYPE.  USE norip2
INPUT COMMAND      finis mfpic sfpic
INPUT PICK  COMMAND
```

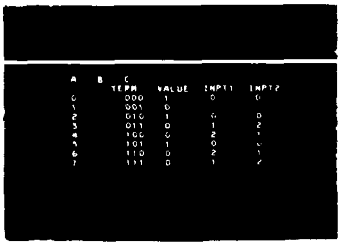


```
A    B    C
          TERM    VALUE   INPT1   INPT2
0          000     1        0       0
1          001     0
2          010     1        0       0
3          011     0
4          100     0
5          101     1        0       0
6          110     0
7          111     0
```



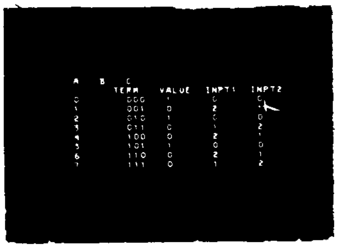
```
TWO-NORG
  IP1-----*UNFINISHED*
  IP2-----*UNFINISHED*
```

```
                                    itpic 1 1 3 7 * 2 4 6 * * *
INPUT PICK  COMMAND
```



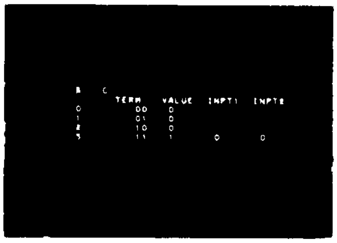
```
A    B    C
          TERM    VALUE   INPT1   INPT2
0          000     1        0       0
1          001     0
2          010     1        0       0
3          011     0        1       2
4          100     0        2       1
5          101     1        0       0
6          110     0        2       1
7          111     0        1       2
```

```
                                    indep
SYMMETRIC ABOUT VARIABLES  1
VARIABLE NUMBER   1 *
INPUT  1 FILLED
INPUT  2 FILLED
INPUT PICK  COMMAND
```



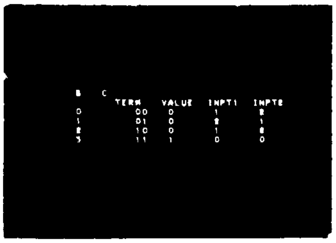
```
A    B    C
        TERM   VALUE  INPT1  INPT2
0        000     1      0      0
1        001     0      2      1
2        010     1      0      0
3        011     0      1      2
4        100     0      2      1
5        101     1      0      0
6        110     0      2      1
7        111     0      1      2
```

```
                                   travl mfpic sfpic
INPUT PICK  COMMAND
```



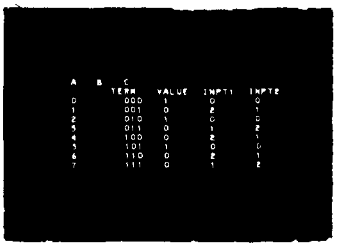
```
B    C
        TERM   VALUE  INPT1  INPT2
0        00      0
1        01      0
2        10      0
3        11      1      0      0
```

```
                                   infit
CONSTANT ZERO (1)
VARIABLES  NOT C ( 2)  NOT B ( 3)
INPUT PICK  COMMAND     inant
  ( 1) =   -3  ( 2) =    1  ( 3) =    1
INPUT PICK  COMMAND     sfdcd
FIT NUMBER  2
INPUT  2 FILLED
OBJECT IS NOW INPUT  2 OF BLOCK TWO-NOR1
INPUT PICK  COMMAND
```

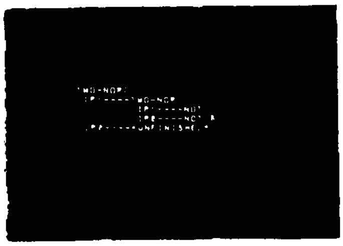


```
B    C
        TERM   VALUE  INPT1  INPT2
0        00      0      1      2
1        01      0      2      1
2        10      0      1      2
3        11      1      0      0
```

```
                                  infit sfdcd
VARIABLES  NOT B ( 1)
FIT NUMBER  1
TWO-NOR1 HAS NOW BEEN COMPLETELY SPECIFIED
OBJECT IS NOW INPUT  2 OF BLOCK TWO-NOR0
INPUT PICK  COMMAND
```



```
A      B      C
              TERM    VALUE   INPT1   INPT2
0             000     1       0       0
1             001     0       2       1
2             010     1       0       0
3             011     0       1       2
4             100     0       2       1
5             101     1       0       0
6             110     0       2       1
7             111     0       1       2
```

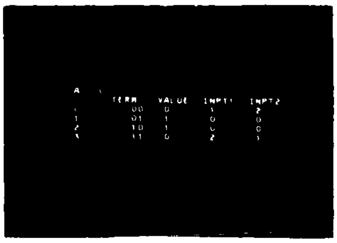


```
TWO-NOR0
 IP1------TWO-NOR
          IP1------NOT
          IP2------NOT B
 IP2------UNFINISHED *
```

```
                                  travl mfpic sfpic
INPUT PICK  COMMAND
```

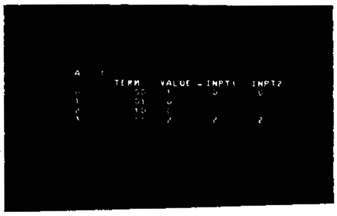
```
A      B
              TERM    VALUE   INPT1   INPT2
0             00      [illegible]
1             01      1
2             10      1
3             11      [illegible]
```

```
                                  infit
CONSTANT ZERO (1)
INPUT PICK  COMMAND               itpic 1 1 0 * 2 3 * * *
INPUT  1 FILLED
INPUT  2 FILLED
INPUT PICK  COMMAND
```



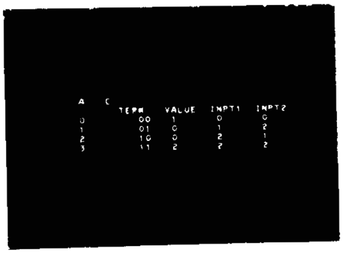
```
A      B
              TERM    VALUE   INPT1   INPT2
0             00      0       1       2
1             01      1       0       0
2             10      1       0       0
3             11      0       2       1
```

```
                                  travl mfpic sfpic
INPUT PICK  COMMAND
```



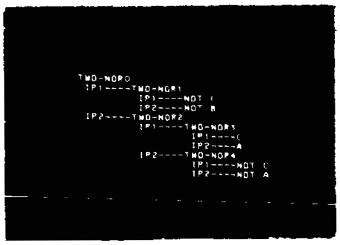
```
A    C
        TERM    VALUE - INPT1   INPT2
0         00      1       0       0
1         01      0
2         10      0
3         11      2       2       2
```

```
                                infit
CONSTANT ZERO (1)
VARIABLES   C ( 2)   A ( 3)
INPUT PICK  COMMAND      inant sfdcd
  ( 1) =    -2  ( 2) =   -0  ( 3) =    0
FIT NUMBER  2
INPUT  2 FILLED
OBJECT IS NOW INPUT  2 OF BLOCK TWO-NOR3
INPUT PICK  COMMAND
```



```
A    C
        TERM    VALUE   INPT1   INPT2
0         00      1       0       0
1         01      0       1       2
2         10      0       2       1
3         11      2       2       2
```

```
                                infit sfdcd 1 travl  mfpic sfpic
VARIABLES   A ( 1)
TWO-NOR3 HAS NOW BEEN COMPLETELY SPECIFIED
OBJECT IS NOW INPUT  2 OF BLOCK TWO-NOR2
INPUT PICK  COMMAND      infit sfdcd
CONSTANT ZERO (1)
VARIABLES   NOT C ( 2)  NOT A ( 3)
FIT NUMBER  2 infit sfdcd 1
INPUT  2 FILLED
OBJECT IS NOW INPUT  2 OF BLOCK TWO-NOR4
VARIABLES   NOT A ( 1)
TWO-NOR4 HAS NOW BEEN COMPLETELY SPECIFIED
TWO-NOR2 HAS NOW BEEN COMPLETELY SPECIFIED
TWO-NOR0 HAS NOW BEEN COMPLETELY SPECIFIED
DECOMPOSITION DONE
MASTER COMMAND    termn
R
```



```
TWO-NOR0
 IP1----TWO-NOR1
        IP1----NOT C
        IP2----NOT B
 IP2----TWO-NOR2
        IP1----TWO-NOR3
               IP1----C
               IP2----A
        IP2----TWO-NOR4
               IP1----NOT C
               IP2----NOT A
```

BIBLIOGRAPHY

1. Class Notes for MIT Course 6.252 Digital Systems Engineering.

2. a) Quine, W.V., "A Way to Simplify Truth Functions," The American Mathematical Monthly, Vol. 62, November 1955, pp. 627-631.

   b) McCluskey, E.J., Jr., "Minimization of Boolean Functions," The Bell System Technical Journal, November 1956, p. 1417.

3. Ross, D.T., AED-0 Programming Manual, Preliminary Releases 1 through 4, AED Flashes 1 through 15, and Internal Memorandum, 1964-65.

4. Fortran II Assembly Program (FAP), IBM Form C28-6235-2, 1963.