DIRECTED SEARCH MINIMIZATION OF MULTIPLE-OUTPUT NETWORKS

by

MICAELA SERRA
B.Sc., University of Manitoba, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computer Science

[redacted]

We accept this thesis as conforming
to the required standard

[redacted]
Dr. J. Muzio

[redacted]
Dr. F. Roberts

[redacted]
Dr. A. Antoniou

[redacted]
Dr. L. Bruton

© MICAELA SERRA, 1984
UNIVERSITY OF VICTORIA
April 1984

All rights reserved. This thesis may not be reproduced in whole or in part, by mimeograph or other means, without the permission of the author.

ABSTRACT.

Given a switching function, a two-level sum-of-products expression is often a useful representation for it. A definition for a minimal cost of realization for a circuit can be agreed upon and techniques exist to produce a minimal or quasi-minimal expression.

Often several switching functions of the same input variables have to be realized in a network. Most of the time the simultaneous synthesis of all the outputs yields more economical results than their separate designs, since the sharing of gates can reduce costs considerably. While there are numerous techniques to minimize single-output functions, few are feasible for extension to multiple-output problems of any size.

A new method for single functions was introduced by Rhyne, Noe, McKinney and Pooch in 1976, called DSA, for Directed Search Algorithm. It has the considerable advantage of not needing to generate all possible prime implicants of a function before being able to choose the essential ones for the coverage.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS.

I want to express my gratitude to Dr. Jon Muzio for his invaluable guidance and support during my time as a graduate student. I also would like to thank Dr. M. Miller who encouraged me to start on the project and NSERC who provided me with the means to continue my studies.

Chapter I

INTRODUCTION.

The application of the rules of Boolean Algebra to Switching Circuits Design has been a most important development for the whole field of Digital Logic Design. All the simplification theorems were made available to the designer who, from the starting specifications, could then implement a circuit using less hardware and simpler logic.

Given a set of formal requirements for a system, the most common type of algebraic expression used to describe the logic needed is the sum-of-products expression. This is the exact equivalent to a disjunctive normal form in Logic Theory, where the truth of an expression is determined by a series of conjunctive terms united by disjunction. For example, the expression AB + CD is in disjunctive normal form where each of A, B, C and D can have a truth value. Then the expression is true when either the term AB is true or when the term CD is true or when they are both true (a disjunction). This in turn means that AB is true when both A and B are true; and CD is true when both C and D are true (two conjunctions).

The switching circuit equivalent implementation presents a level of AND gates (the conjunction) which feed into an OR gate (the disjunction). It is thus called a two-level sum-of-products expression.

It is the interest of the designer to obtain such an expression describing a circuit where the terms are in minimal form. A definition of minimal cost is a necessary assumption and following that techniques can be used to transform a given expression into a logically equivalent one of lowest cost. Computerized algorithms have been developed to achieve such a result, and they usually employ a subset of the simplification rules of Boolean Algebra applied to the binary representation of the given expression.

A similar and more complex problem is when more than one such expression describing circuits must be handled concurrently to find a minimal form. One approach is obviously to minimize them separately using whichever technique is best suited and implement the result, disregarding the fact that they may have been generated by the same primitive values and constants. However it is often the case that trying to achieve a minimal form which looks for possible shared terms leads to a better final result.

Relatively few techniques are available to achieve minimal coverage for networks of functions, for any chosen definition of minimality. It seems important that this lack of tools should be filled, preferably by some technique suited for computer implementation, since the problems can become rather overwhelming for manual minimization.

For the minimization of single functions a new method was introduced by Rhyne, Noe, McKinney and Pooch in 1976 [RHY1]. It was called DSA, for Directed Search Algorithm, and a summary and discussion of it can be found in Chapter III.

We introduce here an algorithm which extends DSA to multiple synthesis retaining its main characteristics, while redefining some structures as necessary. We believe that this extension will prove to be useful and fill the gap currently existing. The algorithm is not overly complicated, does not necessitate new operations, and is suited for both manual and computer implementation.

In Chapter II general statements defining terms and background are presented. Here terminology, PLA's, classical minimization techniques for both single and multiple-outputs can be found. In Chapter III the Directed Search Algorithm is explained as presented originally by its author, with appended to it some modifications and enhancements which improve its efficiency.

Chapter IV treats the multiple-output extension to DSA, and the new method is fully explained, on a theoretical level and with guidelines for a manual algorithm. The implementation point of view is introduced later. Examples follow in Chapter V to illustrate the method more fully.

A computer implementation has been worked out in APL, and full details of it can be found in Chapter VI, while the actual code is in the Appendix. The objective of this work is to inquire into the possibility of extending DSA which so far had given encouraging results for the single-output case. The new algorithm has proved itself very sound in a variety of examples and it demonstrated itself as a powerful tool for multiple-output simplification which has direct applications in the design of all two-level circuits and PLA's. However the implementation itself should be considered as a feasibility study and not as a production program. It is believed that better efficiencies could be introduced and stricter management of space, possibly including a different choice of language, should be investigated if a production program is required.

In Chapter VII a discussion and analysis of the new algorithm can be found demonstrating its usefulness and feasibility.

Chapter II

BASIC CONCEPTS AND DEFINITIONS.

2.1 TERMINOLOGY.

The following terms are defined to assure a common basis of understanding of the digital logic field.

Definition : A *Boolean switching function*, f, of input variables, $x_1, x_2, \ldots, x_n$, each from the set {0,1} is a relation which maps every n-tuple $(x_1, x_2, \ldots, x_n)$ into an output z_i from the set {0,1}.

If there is only one output, the switching function is called a *single-output switching function*, and if there is more than one output, it is called a *multiple-output switching function*.

The symbols used as switching operators are the ones commonly found in Digital Logic. Thus OR is written as "+" to imply disjunction. The AND is sometimes explicitly seen as a dot between variables, but most frequently is indicated by concatenation, and it means conjunction. The logical NOT is written as "'", that is , NOT A will be A'.

Definition: A *literal* is a variable or its complement.

Definition: A *minterm* of n variables is a product of n literals in which each variable appears exactly once in either true or complemented form, but not both. The minterm which corresponds to row i of the truth table is designated m_i, for $i=0,1,\ldots,2^{n-1}$. In general, minterms are designated in decimal notation, so that the minterm corresponding to the input combination $(X_1,X_2,X_3)=(011)$ will appear in row 3 and be identified as m_3

When a Boolean function f is written as a sum of minterms, it is called a *minterm expansion* or a *standard sum of products*. Other names are "canonical sum of products", or "disjunctive normal form", or "canonical disjunctive form".

In some cases there may be combinations of input values which do not actually occur in the logic of the function. The corresponding output can remain unspecified as in the design of the circuit it means that one "does not care" what happens at that point since it should not occur. The function f, for which some of the minterms are left unassigned, is called *incompletely specified*, and the corresponding minterms are referred as "don't care" terms (usually denoted as "X" or "-"). This means that any function that agrees with the original one for all inputs which are specified, can be

used as a cover, where the don't care terms can be assigned as convenient.

Definition: Given a function f of n variables, a product term P is an *implicant* of f if and only if for every combination of values of the n variables for which P=1, f is also equal to 1.

Definition: A *prime implicant* of a function f is an implicant which is no longer an implicant if any literal is deleted from it.

Definition: A *minimal sum-of-products* expression for a function consists of a sum of some (but not necessarily all) of the prime implicants of that function, which completely covers the function. The qualification of minimal is relative to some pre-agreed cost criteria which will be discussed later on. That means that a sum-of-products expression containing a term which is not a prime implicant cannot be a minimal expression.

A sum-of-products expression consists of a series of terms which are products of literals summed together with the OR operator. It corresponds to a circuit with AND gates on the first level and an OR gate on the second level, and for this reason it is called a two-level sum-of-products expression.

Definition: If a minterm is covered by only one prime implicant, then that prime implicant is called an *essential* prime implicant and must be included in any minimal sum of products expression.

Some switching functions do not possess essential prime implicants that cover all the true minterms. The remaining ones can be covered by two or more prime implicants, all of which appear to present equivalent resulting circuits relative to some cost criteria. A cyclic situation is said to exist as the equivalent choices have to be made according to some rule, maybe random, external to the covering procedure itself.

For the rest of the discussion some terms and abbreviations will be used. Each minterm of a switching function (also called "vertex"[1]) can appear in one of three forms:

1. A minterm whose value is the boolean "1" is called a "true form" and denoted as TF. It must be covered by the resulting minimal sum of products.
2. A minterm whose boolean value is "0" is called a "false form" and denoted as FF. It cannot be used in any prime implicant for the cover.

---

[1] From the representation of a boolean function as an hypercube, where each minterm is a vertex [MUR1].

3. A minterm which is redundant is denoted as XF for a "don't care". It need not be covered, unless that proves useful to the minimality of the sum of products.

Definition: The *distance* between two minterms is given by the number of bits which differ in their binary input value representation. For example, m_4 and m_8 are at distance 2 since (0100) and (1000) differ in two bit positions.

Definition: A minterm is *adjacent* to another minterm if they are at distance one from each other. In other words, only one bit differs in their input values $(x_1, \ldots, x_n)$, and the two minterms lie next to each other in the Karnaugh map representation. Moreover, the difference in their decimal representation is a power of two.

When dealing with multiple-outputs, flags are often needed to characterize minterms or prime implicants which come from different parts of the network. Binary vectors are usually introduced as flags and used with some boolean operations.

Definition: Given two vectors $V=(v_i)$ and $W=(w_i)$ from $\{0,1\}$, $V>=W$ if $v_i>=w_i$ for all $i=1,\ldots,n$. Also, if for all i, $v_i>=w_i$ and if $v_j>w_j$ for some j, then $V>W$. If none of

V>W, V<W, V=W is true, then the two vectors are said to be incomparable (V><W).
For example, (1011)>(1001), but (1010)><(0111). In other terms, V is said to be a subset of W if V<=W. W is called a superset of V.

Definition: The *intersection operator* (#), used between two vectors V and W as V#W, performs a bit-by-bit AND operation over the two binary vectors.

2.1.1 Search Trees.

Throughout the Directed Search Algorithm, [DSA], the data structure which is fundamental, at least in the manual method, is the "search tree".[NIL1]

Figure 2.1 shows an example of a search tree. The circled numbers identify *nodes*, the lines between them are the branches which link the nodes together. In DSA each branch will have a weight, i.e. a value associated with it to precisely define the process which brought one node connected to another. Node 1, n_1, is the *base* of the tree, which means it is the starting point for the search. Each node directly below another node is called a *child* of that node, and the first node its *parent*. Thus n_2,n_6,n_8 are the children of n_1. Nodes at the same distance from n_1 are said to

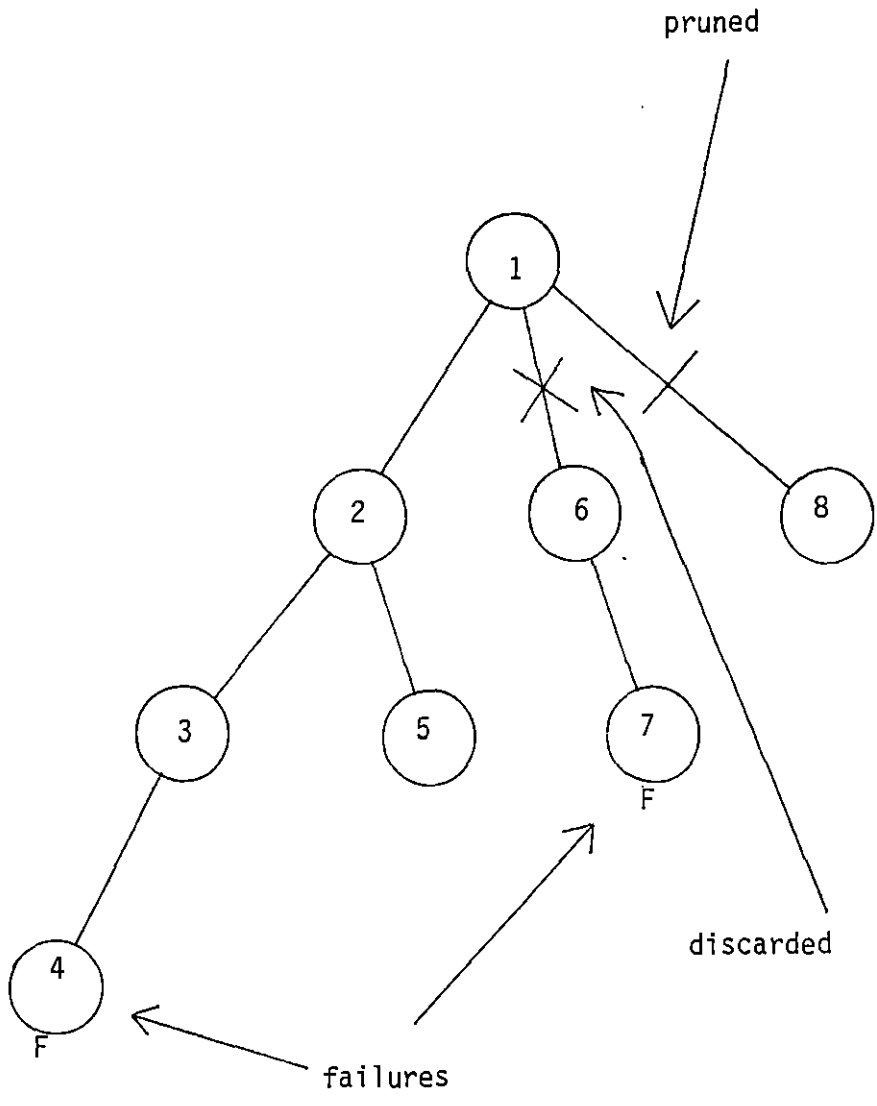


Figure 2.1

be on the same *level*, and they are *siblings* as well if they have the same parent.

All searching is done left-to-right, depth first. By following the numerical order of the nodes of Figure 2.1 we describe the actual search order. Moreover, not every node in the search tree is necessarily reached or even constructed. We call a *failure node* one that has been constructed by the search, tested (by whatever "testing" is implied), and found to be not successful: in this case an "F" is added to it. A branch is *pruned* if it is not traversed at all, i.e. nodes are not even generated on it. In the Figure 2.1 it is indicated by a line perpendicular to the branch. A *discarded* branch instead is one that is partially traversed and then later abandoned, for whatever reason: it is shown by an "X" across it. The failure nodes are reached by the search but deleted from the tree; the pruned nodes are never reached; the discarded nodes are reached and left on the tree until further "testing" deletes them; at that point only the "successful" node(s) are left on the tree as a result of the search.

As opposed to traversing a tree, *backtracking* means ascending the tree from children nodes towards parent nodes. Backtracking is usually done one level at a time, with fur-

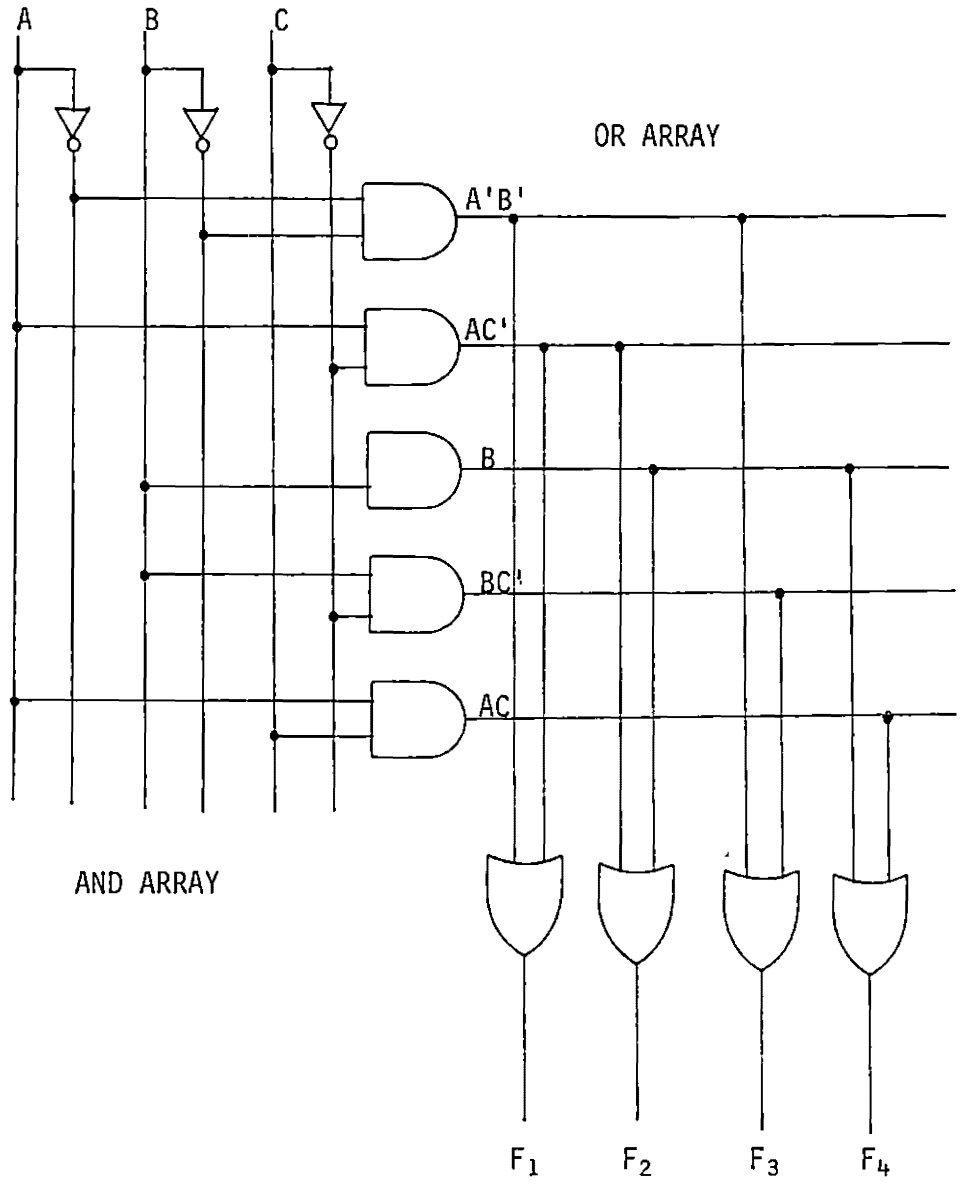
ther expansion downward, and then more backtracking for another level. For example, in Figure 2.1, the tree was traversed from n_1 to n_4, then n_4 found to be a failure and deleted. At this point backtracking occurred from n_3 to n_2 in order to expand n_5. Then further backtracking to n_1 to continue the other parallel expansions.

The idea of the search tree as a guideline for the DSA will prove to be very powerful as an aid to the algorithm. For the implementation itself only the logic behind DSA was retained while the actual data structures used are at times different.

2.2 PLA'S.

A programmable logic array is an LSI circuit which consists of two arrays of semiconductor devices (diodes, bipolar transistors or field-effect transistors) which are interconnected. A PLA with n inputs and m outputs can realize m functions of n variables. (See Figure 2.2).

The first array is called the AND array and it realizes selected product terms of the input variables. The emphasis is on "selected". Unlike a decoder which has to implement *all* 2^n minterms and then choose the useful ones from among them by use of extra control lines, a PLA need only implement the ones needed for that particular set of functions.



F_1= A'B' + AC'

F_2= AC' + B

F_3= A'B' + BC'

F_4= B + AC

Figure 2.2

The array itself has switching elements at every intersection of a row and a column. Both 0 and 1 states for the inputs are generated by inverters . The outputs of the first array (the product terms) are the inputs of the second array, called the OR array. This second set of connections forms the logical sum (OR) of the product terms of literals, thus resulting in a sum-of-products for each of the final output lines which are in turn the functions we want to realize.

The number of outputs from the AND array which is equal to the number of inputs into the OR array is exactly the same as the number of product terms which need to be used. Only the lines corresponding to a product of literals are used by the PLA and fuses blown accordingly at the intersection with the proper literal (or its negation). This latter aspect has no bearing on the actual size of the PLA, only on its internal "crowding" factor. By crowding we mean the actual number of fuses blown on the internal lines, which may become of significance only in relation to testing and fault detection.

Both mask-programmable and field-programmable PLA's are available. The mask-programmable type is programmed by the manufacturer, while the field-programmable one has "fusible

links" which can be blown for a particular pattern. From this we may notice that while the number of output lines of the PLA is fixed by the number of functions we want to implement, the number of input lines varies. Reducing the size of a PLA implies only reducing the number of the input lines, since the internal structure does not change. This in turn means a reduction of the number of product terms.

A product term which corresponds to a larger implicant of a function will have fewer literals and hence fewer connection at the row/column intersections. However this does not reduce the geometrical layout of the PLA, which is the only consideration when wanting to reduce the actual amount of space taken on a chip or on a board.

In summary then, when minimization of boolean function towards a sum-of-products expression is done with a PLA implementation in mind, then the only required criterion for minimal cost (see next section for further discussion of this) is the number of product terms, disregarding their size (the number of literals) or the presence of inverters.

2.3 MINIMIZATION TECHNIQUES.

Minimization means reducing the size of the expression which describes the given switching function . In order to be able to talk about minimal expressions we must have some criteria that define the "cost" of an expression. A number of possible cost criteria can be introduced including:

-the number of product terms, or

-the total number of literals in the expression.

An expression is then said to be minimal when according to the chosen cost criteria, all other representations have a higher cost.

Considering the structure of PLA's, however, one notices that the number of output lines is determined, and fixed, by the number of functions that the particular PLA is called upon to implement. The number of lines between the AND and OR arrays, on the other hand, corresponds to the number of product terms in the canonical disjunctive form.

Thus minimizing the number of lines of a PLA corresponds to minimizing the number of product terms, while the number of literals has no effect since they only indicate the number of connections inside the structure.

2.3.1 Classical minimization.

We will first consider the handling of single-output functions and their reduction to two-level minimal sum-of-products expressions. An overview of the most frequently used techniques is given, together with the appropriate references where more detailed explanation of each method can be found.

All these algorithms are based on the same set of procedures:

a) Generate *all* prime implicants of the given function.

b) Obtain a minimal covering using all essential prime implicants and choosing appropriately among the non-essential ones.

Since the number of prime implicants can be quite large as n, the number of inputs, increases, the first part of the process can be very time consuming. As a result the covering problem which has to be later solved is very difficult.

To give an idea of the sizes we are talking about, the following Table 1 shows results compiled by Fridshal [FRI1], presenting some minimum upper bounds for the number of prime implicants as a function of the number of inputs.

It can be seen that to both generate such a large number of PI's and then solve a covering table can be a very big job,

TABLE 1.

Number of variables	Number of PI's
2	2
3	6
4	13
5	32
6	92
7	218
8	576
9	1,698
10	4,300
11	11,000

particularly when a minimal cover may only include a very small percentage of the prime implicants.

In the case of incompletely specified functions, one other drawback is present. When identifying the prime implicants of a function, the don't care terms are included in the list of TF's, since they might be useful towards obtaining a minimal cover. This of course can add considerably to the generation process, and the amount of work done could at times prove to be totally wasteful, since quite a few prime implicants might be generated containing only don't care terms.

At the stage when the covering table is used to choose the final cover, only prime implicants containing at least one TF are used, and all minterms with XF status are not even listed for needing a cover. The following methods are most widely used:

1. *Algebraic Factorization*: [*ROC1*] This technique is always valid, but at times quite difficult to apply systematically, and most of all to determine whether a minimal solution has been achieved or not .
2. *Karnaugh Maps*: [*KAR1,ROC1*] An excellent topological method, which may be used to graphically solve functions of up to 6 variables .
3. *Quine-McCluskey*: [*MCKL1,ROC1*] This well-designed deterministic method starts from a minterm list and generates all prime implicants through an application of the Simplification Theorem AB+AB'=A to its continually updated list. A table then reduces the final set to the minimal cover. It is well-suited for computer implementation, but not for manual use. Moreover it tends to involve an unreasonable amount of work when the number of inputs is large, since the list of prime implicants generated can be very long.
4. *Consensus*: [*QUI1,TIS1*] In this iterative algorithm, implicants are successively "joined" together through the application of the Consensus Theorem (AB+BC+A'C=AB+A'C), and used to form prime implicants. It is more general than Quine-McCluskey as it can start from any sum of product representation.

5. *Topological method*: [*DIE1*] It uses the concept of "cubes" to represent a Boolean function, and the application of the "sharp" operator to the cubes generates the prime implicants.
6. *Others*: [*PET1,BOW1,MCC2*] Other work has been done more specifically in the covering problem area, dealing with the reduction of large tables of prime implicants to a set of essential ones. They are mainly tabular methods, using direct scanning techniques as opposed to more heuristic searches.

2.3.2 Other Minimization Methods.

Other methods have been presented and are less well-tested: Sureshchander [SUR1] published an algorithm based on the r^{th} degree consensus term of a "well-chosen true form". He did not follow it up with further research or even results on applications, possibly because of the complexity of the operations involved.

Reusch [REU1] presented a more classical approach of generating prime implicants and then finding a cover, with optimizations introduced in terms of simplification trees, Shannon decompositions, and backtracking. He did not claim it to be programmable. The computation of irredundant sums

as a better covering techniques is based on Petrick's method [PET1].

The methods described so far have been based upon the search for a minimal expression. However in practice the realization of a switching function has to conform to other specifications besides minimality, and often an expression which is not the absolute minimal form is perfectly acceptable. This type of expression is usually called a "quasi-minimal" form. More heuristics approaches have also been introduced in the literature, based on the assumption that a quasi-minimal cover is sufficient in most cases. The best example is MINI, based on iterative improvement, and presented in 1974 by Hong et al [HON1]. From MINI, further minimization techniques have been developed specifically for PLA's, using more heuristics to also deal with input and output redundancies, especially for very large functions.[KAN1]

In the following chapter the DSA algorithm as explained by McKinney will be discussed as a basis for further work. [MCK1]

2.3.3 Multiple-output minimization.

The approaches discussed so far are applicable to single-output switching functions. In the cases of multiple-output the choice of methods becomes even more restricted. The main addition to any of them is of a flag for every minterm or implicant to specify for which of the functions in the network this particular term is being considered. While at first this appears to be just a simple extension, it actually makes the number of choices much larger.

The approach to multiple-output synthesis has to take into consideration that some gates could be shared among functions, thus reducing the total number. However one has to be able to decide which gates exist, in common and not, and which ones should be selected for sharing.

In general, the simultaneous synthesis with some common gates will improve the result, and does no worse than the separate minimization of the functions provided that some reasonable rules of choosing shared gates are followed.

Generating all prime implicants for one function requires a large amount of work: generating all prime implicants for a set of functions, *including* all possible common prime implicants can result in a very large covering table. Moreover the basic definition of "essential" prime implicant,

which is fundamental in reducing the set of resulting terms into a minimal cover needs to be further qualified at all times during the process. A prime implicant, in fact, can be essential for one of the functions of the network and not even appear in any of the others. Or it might appear as part of a larger term, which superficially might be thought as desirable. Conversely the best covering choice for a minimal realization might include some terms which are not essential for all the functions, substituting for an essential one, yet achieving a smaller cover.

These concepts will become clearer in the next example, where a multiple-output covering table is processed in detail and the determination of essentiality for one or more functions is pragmatically solved.

To show exactly how the process works and can become complex, an example will be illustrated in detail, by applying the Quine-McCluskey extended algorithm to a multiple-output problem.

Example 1. Consider the following multiple-output switching function, f. Its truth table can be seen in Table 2, while the Karnaugh maps are shown in Figure 2.3 for further clarity.

TABLE 2.

f:

x_1	x_2	x_3	x_4		Z_1	Z_2	
0	0	0	0	->	0	0	m_0
0	0	0	1	->	1	1	m_1
0	0	1	0	->	0	0	m_2
0	0	1	1	->	1	1	m_3
0	1	0	0	->	0	0	m_4
0	1	0	1	->	1	1	m_5
0	1	1	0	->	0	0	m_6
0	1	1	1	->	1	X	m_7
1	0	0	0	->	1	1	m_8
1	0	0	1	->	X	X	m_9
1	0	1	0	->	1	1	m_{10}
1	0	1	1	->	X	X	m_{11}
1	1	0	0	->	0	0	m_{12}
1	1	0	1	->	0	0	m_{13}
1	1	1	0	->	1	0	m_{14}
1	1	1	1	->	1	X	m_{15}

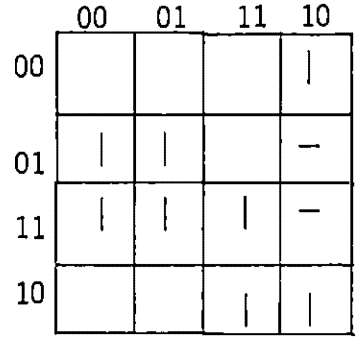
Let the inputs be x_1 ,x_2 ,x_3 ,x_4 ,and the outputs be z_1 ,z_2. Then the function f can be specified as a multiple-output switching function as a composition of two single switching functions of the same inputs.

$Z_1\ (x_1\ x_2\ x_3\ x_4) =$ m(1,3,5,7,8,10,14,15)+ d(9,11)

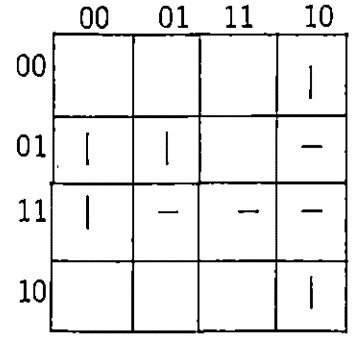
$Z_2\ (x_1\ x_2\ x_3\ x_4) =$ m(1,3,5,8,10)+ d(7,9,11,15)

These three functions could be implemented separately, but the sharing of gates will show that we can achieve a smaller circuit.

The first step is to classify the starting TF minterms as to their relative distance from minterm 0, i.e. the number of 1 bits in their inputs. Table 3 shows the initial setting.



Z_1



Z_2

Figure 2.3

TABLE 3.

1	Z_1 Z_2	0001
8	Z_1 Z_2	1000
3	Z_1 Z_2	0011
5	Z_1 Z_2	0101
9	Z_1 Z_2	1001
10	Z_1 Z_2	1010
7	Z_1 Z_2	0111
11	Z_1 Z_2	1011
14	Z_1	1110
15	Z_1 Z_2	1111

Each term is designated by its decimal representation (m_i), its flag, as to which function it covers, and its binary representation. As usual we combine terms by comparing members of one group with every member of the adjacent group. Then 1-[Z_1,Z_2] and 3-[Z_1,Z_2] are combined into an implicant, 1,3-[Z_1,Z_2].(Table 4 shows all subsequent steps.) A check mark (*) can be placed on both 1 and 3, because they are fully contained in 1,3-[Z_1,Z_2].

Then one can obtain 1,5-[Z_1,Z_2] by combining 1-[Z_1,Z_2] and 5-[Z_1,Z_2], and so on for the others. Note however, that when we come to combine 10-[Z_1,Z_2] and 14-[Z_1], only 14 can be checked off, since the flag for 10 is not completely covered by the new term 10,14-[Z_1]. Minterm 10 can be checked off from the list when it is coupled with 11 to form 10,11-[Z_1,Z_2].

It must be noted that if the case had been that 10-[Z_1,Z_2] was combined separately with two other terms each of which included only a portion of the flag, for example 10,14-[Z_1] and maybe 10,2-[Z_2], then 10-[Z_1,Z_2] should not be checked off since the whole flag had not been used. This would add to the final number of PI's by forcing the inclusion of 10-[Z_1,Z_2] by itself.

After the first set of iterations, fourteen terms are present, as listed in the second part of Table 4. This process continues until no new term can be generated, exactly as it is performed in the single-output case. The extra caution has to be always applied when checking off terms by inspecting the relative flags.

The last part of Table 4 shows that the combination process yields five prime implicants:

1,3,5,7-[Z_1,Z_2]

1,3,9,11-[Z_1,Z_2]

8,9,10,11-[Z_1,Z_2]

3,7,11,15-[Z_1,Z_2]

10,11,14,15-[Z_1]

In order to determine the minimal cover from Table 5, we notice that in the Z_1 subtable, the column for minterm 5 possesses only one check mark: thus the PI indicated from

TABLE 4.

1	Z_1 Z_2	0001 *
8	Z_1 Z_2	1000 *
3	Z_1 Z_2	0011 *
5	Z_1 Z_2	0101 *
9	Z_1 Z_2	1001 *
10	Z_1 Z_2	1010 *
7	Z_1 Z_2	0111 *
11	Z_1 Z_2	1011 *
14	Z_1	1110 *
15	Z_1 Z_2	1111 *
1,3	Z_1 Z_2	00-1 *
1,5	Z_1 Z_2	0-01 *
1,9	Z_1 Z_2	-001 *
8,9	Z_1 Z_2	100- *
8,10	Z_1 Z_2	10-0 *
3,7	Z_1 Z_2	0-11 *
3,11	Z_1 Z_2	-011 *
5,7	Z_1 Z_2	01-1 *
9,11	Z_1 Z_2	10-1 *
10,11	Z_1 Z_2	101- *
10,14	Z_1	1-10 *
7,15	Z_1 Z_2	-111 *
11,15	Z_1 Z_2	1-11 *
14,15	Z_1	111- *
1,3,5,7	Z_1 Z_2	0--1
1,3,9,11	Z_1 Z_2	-0-1
8,9,10,11	Z_1 Z_2	10--
3,7,11,15	Z_1 Z_2	--11
10,11,14,15	Z_1	1-1-

that bit is essential, since it is the only one to cover

that minterm. By choosing PI=(1,3,5,7) we associate to it the largest flag possible, in this case $[Z_1,Z_2]$, and by doing so we have also covered part of the subtable for the other output function.

Similarly, PI=(8,9,10,11,) is essential for minterm 8 in Z_1 and it must be chosen with flag $[Z_1,Z_2]$. The only ones left now are m_{14} and m_{15} in Z_1 PI=(10,11,14,15) is essential for them. The expressions are thus reduced to the use of only three prime implicants, as in:

$$Z_1 = x_1'x_4 + x_1x_2' + x_1x_3$$

$$Z_2 = x_1'x_4 + x_1x_2'$$

Notice here that a total of five product terms appear, three in Z_1, two in Z_2; however only three of them are unique, and the circuit diagram in Figure 2.4 shows the sharing of gates.

The immediate conclusion that can be drawn from this rather simple example is that the Quine-McCluskey method is quite elaborate for multiple-output applications, and the covering table part can assume horrendous proportions. This is because each table for each function in the network can be rather large, and all the tables have to be concatenated together to enable a simultaneous synthesis. It is defi-

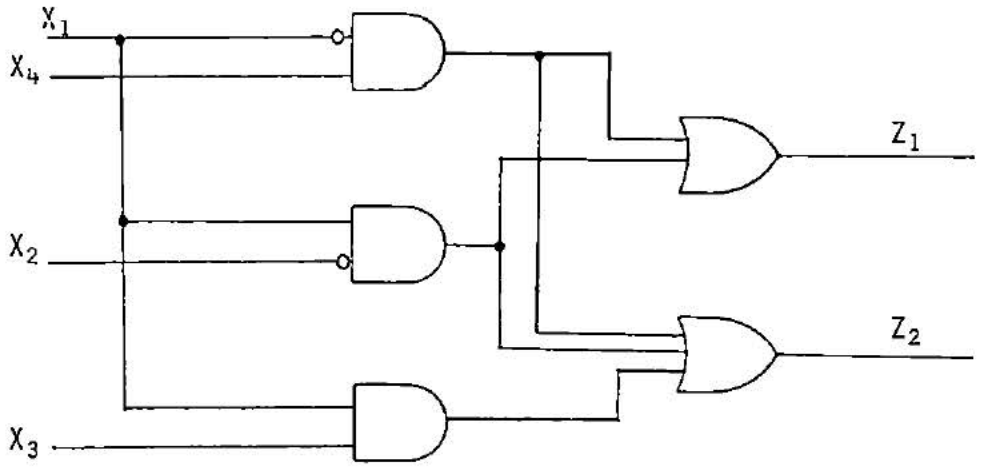


Figure 2.4

TABLE 5.

	Z_1								Z_2				
	1	3	5	7	8	10	14	15	1	3	5	8	10
1,3,5,7	1	1	1	1	0	0	0	0	1	1	1	0	0
1,3,9,11	1	1	0	0	0	0	0	0	1	1	0	0	0
8,9,10,11	0	0	0	0	1	1	0	0	0	0	0	1	1
3,7,11,15	0	1	0	1	0	0	0	1	0	1	0	0	0
10,11,14,15	0	0	0	0	0	1	1	1	0	0	0	0	1

nitely impossible, even on computer, to handle large functions.

Chapter III

THE DSA ALGORITHM.

The Directed Search method was presented first by McKinney in his Ph.D. Thesis [MCK1] and later a full manual and computer algorithm was presented with Rhyne et al. [RHY1]

Almost all other techniques handle the minimization as the sum of two separate processes. In the first part *all* prime implicants of the given function are generated. Then the set of PI's which provide the best cover is chosen.

In DSA the two phases are not distinct, and the method operates in what its author calls "a synergistic manner". Not all prime implicants are generated, only the ones which will be likely to provide a possible minimal cover. Moreover any PI is constructed only once, as opposed to the Quine-McCluskey method where the avoidance of duplicates has to be taken into consideration. In section 3 of this chapter an example will be worked out both with DSA and the Quine-McCluskey method, and the actual number of PI's generated, duplicated and eventually selected will be shown explicitly.

Essential prime implicants are determined through a search process applied during the generating phase. Cycles[2] are detected but not directly resolved by the algorithm. Thus instead of employing an iterative search for all possible prime implicants, DSA uses a directed tree-search over only a portion of the possible cells covering minterms.

A few definitions have to be stated to explain the algorithm. We previously defined that two minterms are adjacent when they are at distance one, and hence they lie next to each other in the Karnaugh map. In order to quickly determine whether two minterms are adjacent, two conditions have to be satisfied:

a) $ABS(m_i-m_j)$ is a power of two

b) $AND(m_i,m_j)=MIN(m_i,m_j)$

where ABS(a) indicates the function returning the absolute value, AND(a,b) indicates the bit-by-bit boolean conjunction operation, and MIN(a,b) returns the minimum of a and b.

Every adjacency for a minterm has associated with it a magnitude, (the power of two which is the decimal difference between the two adjacent minterms), and a direction denoted by a positive or negative sign. Hence an adjacency can be

---

[2] Cycles are instances when two or more distinct sets of PI's provide minimal cover, and the choice cannot be made in a deterministic way.

seen as a vector. For example, 0 is +1 adjacent with 1 and +4 adjacent with 4; while 7 is -2 adjacent with 5. Given that the magnitude of the adjacency is a power of two, we must also know whether the direction of adjacency is positive or negative. For this we can AND the binary representation of the minterm with the binary representation of the absolute value of magnitude of the adjacency under consideration. If the result is equal to zero, then the direction is positive, else it is negative. For instance: consider minterm 1 and adjacency 4. Since AND(001,100)=000, then 1+4-->5, and minterm 5 is adjacent to 1. If we consider minterm 7 and magnitude 2, then AND(111,010)=010 and 7-2-->5.

The reason behind the use of a positive or negative sign is that we want to be able to associate a minterm and a possible adjacency, with the resulting new related minterm both being the correct adjacent one and remaining in the prescribed range of values, $\{0,1,\ldots,2^{n-1}\}$.

We can now define a *Required Adjacency Direction*, called RAD, as an adjacency which relates a true minterm to either a true or a don't care minterm. For example, consider the four-variable function with TF=(0,1,3,6,11,14,15), and XF=(2,4,5,13), and the rest of the minterms FF. The list of RAD's would be:

TF	RAD's	Total TF RAD	Total XF RAD
0	+1,+2,+4	1	2
1	-1,+2,+4	2	1
3	-1,-2,+8	2	1
6	-2,-4,+8	1	2
11	+4,-8	2	0
14	+1,-8	2	0
15	-1,-2,-4	2	1

The whole DSA is based on using RAD's in expanding a TF towards a larger implicant. Since the RAD's are vectors, the expansion process can be represented as a sort of tree structure, where each node is either a prime implicant or an implicant. Choosing an essential prime implicant means pruning the tree appropriately.

As an example the TF=0 of the previous function is expanded to show its RAD tree in Figure 3.1.

The above tree shows that TF=0 is covered by three prime implicants: (0,1, 2, 3), (0,1, 4,5) and (0, 2,4, 6). The underlining refers to don't care terms. The full tree need not be fully expanded all the time as it is in this case, we show it here as such to illustrate the function of RAD's.

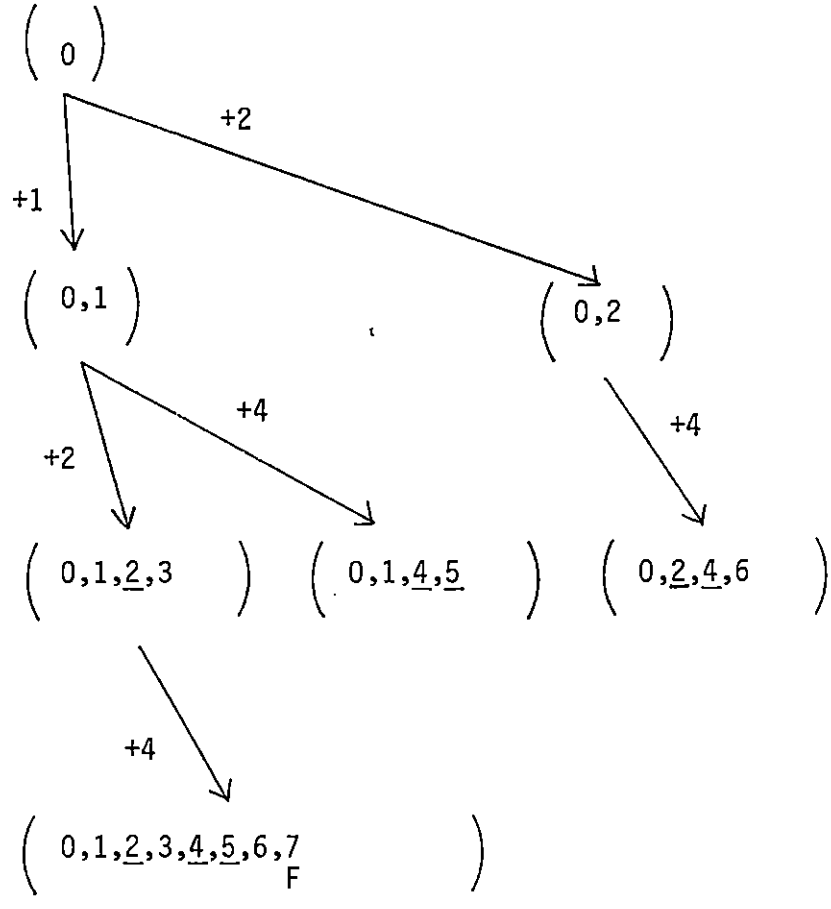


Figure 3.1

A RAD is a direction of adjacency that, when associated with either a minterm or an implicant, may possibly relate that term to a previously unidentified prime implicant. Reducing the number of RAD's will reduce the amount of search involved in the tree expansion, and hence in the covering of a TF.

Given a TF, there are n possible directions of adjacency in an n-variable function. However, when the new term related from a base TF through an adjacency is either a FF, a XF, or an already expanded TF (=XF), then that direction is not required to be used by the tree-search, since it will not produce any *new* PI's.

A few observations can be made at this early stage, which will help for the more precise definition of the algorithm.

1. The order of expansion does not matter. Here, in the example, an ascending order of RAD's has been applied, left to right. However a permutation of the RAD's would simply cause a reordering of the branches, while the respective contents of the nodes would remain the same.
2. A term (a node) formed by RAD expansion is an implicant if and only if it does not contain a false minterm. Thus (0,1, 2, 3, 4,5, 6,7) is not an implicant since it contains FF=7.

3. An implicant becomes prime when it cannot be expanded further with another RAD.
4. Given a TF, every prime implicant covering it can be found by RAD expansion.
5. The number of RAD's defines the order of the largest prime implicant that may possibly cover a given TF.

As was stated earlier, in order to find the covering for a given TF, the whole tree need not be examined. A RAD tree can be pruned by eliminating branches which reflect paths already followed in another sequence. Then the leftmost branch depth-search is the most expanded, while the other branches are developed only if the leftmost one has not exhausted all possible RAD's, and only for those RAD combinations not yet explored.

Once a TF has been expanded into a RAD tree and a node of the tree proves to be both a prime implicant and it has used all possible RAD's, then that node provides an essential cover for the original TF. That essential prime implicant can be selected for the minimal cover, no further branches need to be expanded in the tree. Moreover, all the minterms listed in it are now covered as well and we need not consider them any longer as TF's. Hence their status can now be changed to XF. This implies that they will never be selected as a starting point for tree expansion.

These last simple observations bring up the strength of DSA: first of all, by proper pruning, not all prime implicants are generated. Secondly by the selection made in any tree causing a TF to become an XF for the remaining search, trees are not expanded to duplicate covers.

3.1 GENERAL STATEMENT.

DSA begins with the knowledge of the TF's, XF's and FF's of a given function. A TF is chosen as a starting point for the minimization process, called TF_0. XF's, while accepted in any cover, are never selected as starting points.

The initial RAD expansion identifies the members of $\{PI\}_0$, that is, the set of PI's that cover TF_0. Along with this general description, an example is worked out to clarify things. Let $Z(x_1,x_2,x_3,x_4) = m(0,1,5,8,9,13,15) + d(6,7,11,14)$, where "d" precedes the list of don't care terms. Let TF_0 be m_8. Then its RAD tree is very simple, since there are only two adjacencies (-8 adjacent to 0, +1 adjacent to 9), and $\{PI\}_0=\{(8,9,0,1)\}$.

If $\{PI\}_0$ contains only one member, i.e. it is a tree with one branch using all RAD's, then that PI is essential to the cover of F, since it is the only prime implicant covering TF_0. Then that PI is listed for the minimal cover, and all

the TF's contained in it are changed to XF's. This ensures that those TF's will not be chosen later as starting points, since a cover for them has already been found in the search process of a cover for this other TF_0. Then another TF is selected as TF_0 for the next expansion, from the possibly more restricted list, since some TF's will have changed to XF's.

In our example, PI=(8,9,0,1) is chosen as essential, and the status of all of m_8,m_9,m_0 and m_1 is changed to XF. The function is now: Z= m(5,13,15) + d(0,1,6,7,8,9,11,14). This is the simplest case.

On the other hand $\{PI\}_0$ may contain more than one member. The first step is to compare the members pairwise and delete any member containing only XF's (they do not add anything new to the cover). Then, if possible, select the PI containing the largest number of uncovered TF's, if it is uniquely so determined. This will be called *pseudo-essential a* PI. It is not "essential" by proper definition, but DSA makes it so by choosing it for dominance. It is the equivalent process of row-dominance in a covering table; however some unnecessary PI's have already been pruned by the RAD tree, and the comparisons are made only for the subset of PI's for a particular minterm, instead of the whole function.

Thus a *pseudo-essential* prime implicant is a PI which is part of a set of PI's all providing cover for a given TF_0, but one which provides cover for a larger number of TF's other than TF_0.

After this selection,other PI's in the set may be deleted as they will result containing only XF's from the pseudo-essential choice.

In the example, let the next TF_0=15. The RAD tree produces $\{PI\}_0$={(15, 14,7,6,), (15,13, 11,9), (15,13, 7, 5)}. The first member can be eliminated since it contains, besides for TF_0=15, only XF's. The third member, (15,13, 7, 5,), provides cover for three TF's in total, while the second one, (15,13, 11,9) only for two. Thus we can choose PI=(15,13, 7, 5) as pseudo-essential. Having done that, the remaining PI=(15,13,11,9) is now composed only of XF's and can be deleted.

As a last case, if dominance does not reduce $\{PI\}_0$ to a single member, then a new RAD-tree search is used to make a selection, where the origin TF_0 is one of the uncovered TF's present in one of the PI's listed in $\{PI\}_0$. This is the equivalent of generating subtrees expanded from within a node of the original tree.

The new RAD expansion will eventually make a selection. By returning towards the original tree (more than one subtree might have been used) some PI's in $\{PI\}_0$ will have had their minterms changed from TF's to XF's and can then be eliminated until $\{PI\}_0$ contains one or zero members. All the PI's chosen in this recursive process are pseudo-essential ones.

Thus if essentiality of cover cannot be directly constructed from the search of one minterm, the TF's that are listed in the prime implicants so far identified by the search become, themselves, the basis for additional tree-search. By using the recursive definition, any information collected in previous (recursive) searches can be used by any subsequent (recursive) searches, to reduce the expansion of the recursive trees (usually called "subtrees"). Moreover the information gathered in all searches iteratively is immediately inserted into the global temporary cover, i.e. TF's are changed to XF's, and this makes the total number of tree-searches considerably smaller than the breadth-first expansion of all minterms applied by other methods.

3.2 THE MANUAL ALGORITHM.

After the general explanation given so far, more precise rules can be outlined for a manual procedure.

STEP 1. (Initial RAD tree construction)

Choose an uncovered TF as TF_0 and construct its pruned RAD tree from its list of adjacencies. If no TF is left uncovered, the process is finished.

STEP 2.

a) If the RAD tree consists of only one branch exhausting all adjacencies, then the leaf node is an essential prime implicant. It should be added to the solution and all TF's included in it have their status changed to XF's.

b) If the RAD tree has more than one branch, examine the leaf nodes. Remove any of these which are dominated and any that contain only XF's. Among those remaining, choose as pseudo-essential, if possible, one PI containing the highest number of uncovered TF's other than TF_0.

c) If at this point all TF's in every PI on the RAD tree have been covered, then go to Step 1. Else, if some PI's are left, each containing the same number of uncovered TF's, go to Step 3.

STEP 3.

Choose an as yet unexpanded TF from those remaining on the RAD tree. This will become the root for another RAD tree, which will be a recursive subtree with origin TF_1. Expand this subtree, not using any RAD's previously used on the branch from which this TF originated. That is, do not use any adjacency which will relate TF_1 back to its previously expanded PI parent. Go to Step 2 to make choices.

If the possible choice for a TF_i in this recursive process happens to be the same as a previous TF_j, then a cycle has been encountered. This must be solved by some method: the simplest step to take is to add to the cover any of the PI's found in the cycle so far. By backtracking and changing status of TF's, more choices are made within the cycle.

An example will clarify these steps: Let $Z(x_1,x_2,x_3,x_4) = m(0,1,3,6,11,14,15) + d(2,4,5,13)$. Its Karnaugh map is shown in Figure 3.2. Leaving any other consideration aside, we can choose as TF_0 the first TF in the list, in this case being 0. Its RAD list is (+1,+2,+4) and its expanded RAD tree is shown in Figure 3.3. Notice how the node=(0,1,2,3,4,5,6,7) is not acceptable since it contains FF's. In this case $\{PI\}_0$ consists of {(0,1, 2, 3), (0,1,

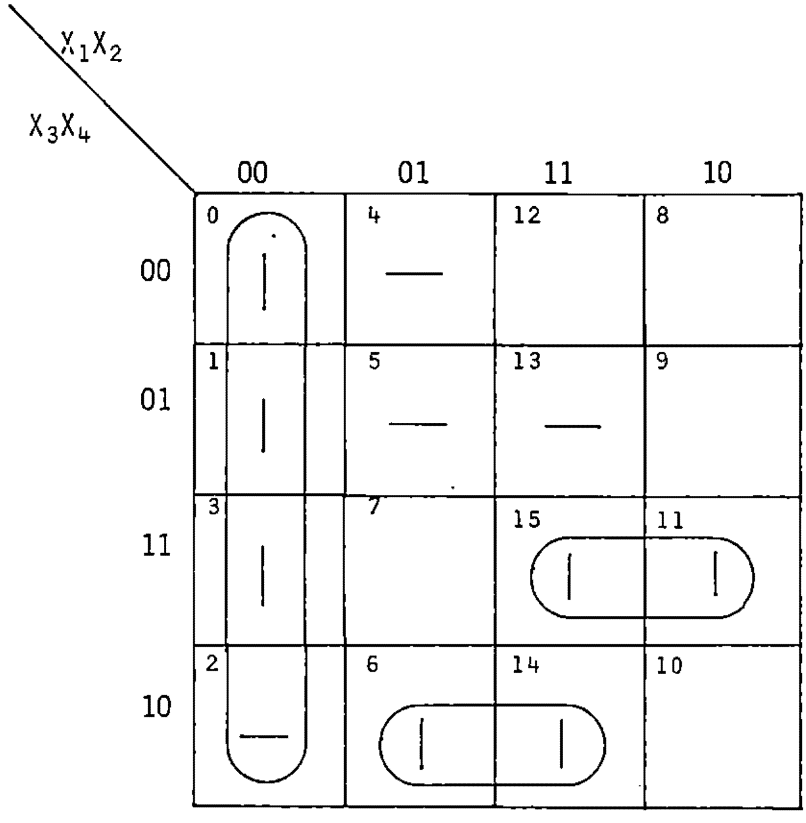


Figure 3.2

4,5), (0, 2,4, 6)}, and no member can be immediately eliminated. We can however make a first choice of pseudo-essential of PI=(0,1, 2, 3) since it is the one which contains 3 uncovered TF's.

After this choice, PI=(0,1,4,5) now consists all of XF's and can be eliminated. The only member left is (0, 2,4, 6) and we must decide, through the use of a subtree, whether it should be chosen or not. The subtree must start from TF_1=6 and it produces the set $\{PI\}_1$ ={(6, 4), (6,14)}. from this set we can delete (6, 4) since it contains no new TF's, and then we would like to choose (6,14) as pseudo-essential. In order to validate this choice a subtree can be started from TF_2=14. Only the +1 adjacency will be used, since the +8 RAD was used in the parent node to this subtree. $\{PI\}_2$={(14,15)}, and again TF_3=15 and $\{PI\}_3$={(15,11)}. When we finally use TF_4=11 we obtain $\{PI\}_4$={(11, 3)} which can be discarded. At this point backtracking can occur, and since no choice was made in the last subtree, the first pseudo-essential will be PI=(15,11). This makes PI=(14,15) in $\{PI\}_2$ redundant, and the next pseudo-essential is indeed PI=(6,14). The cover is now complete

The amount of work involved may seem at this point to be rather great. In fact in the next section more reasoning

and rules are discussed which will considerably reduce the number of trees involved in the search, even for this example.

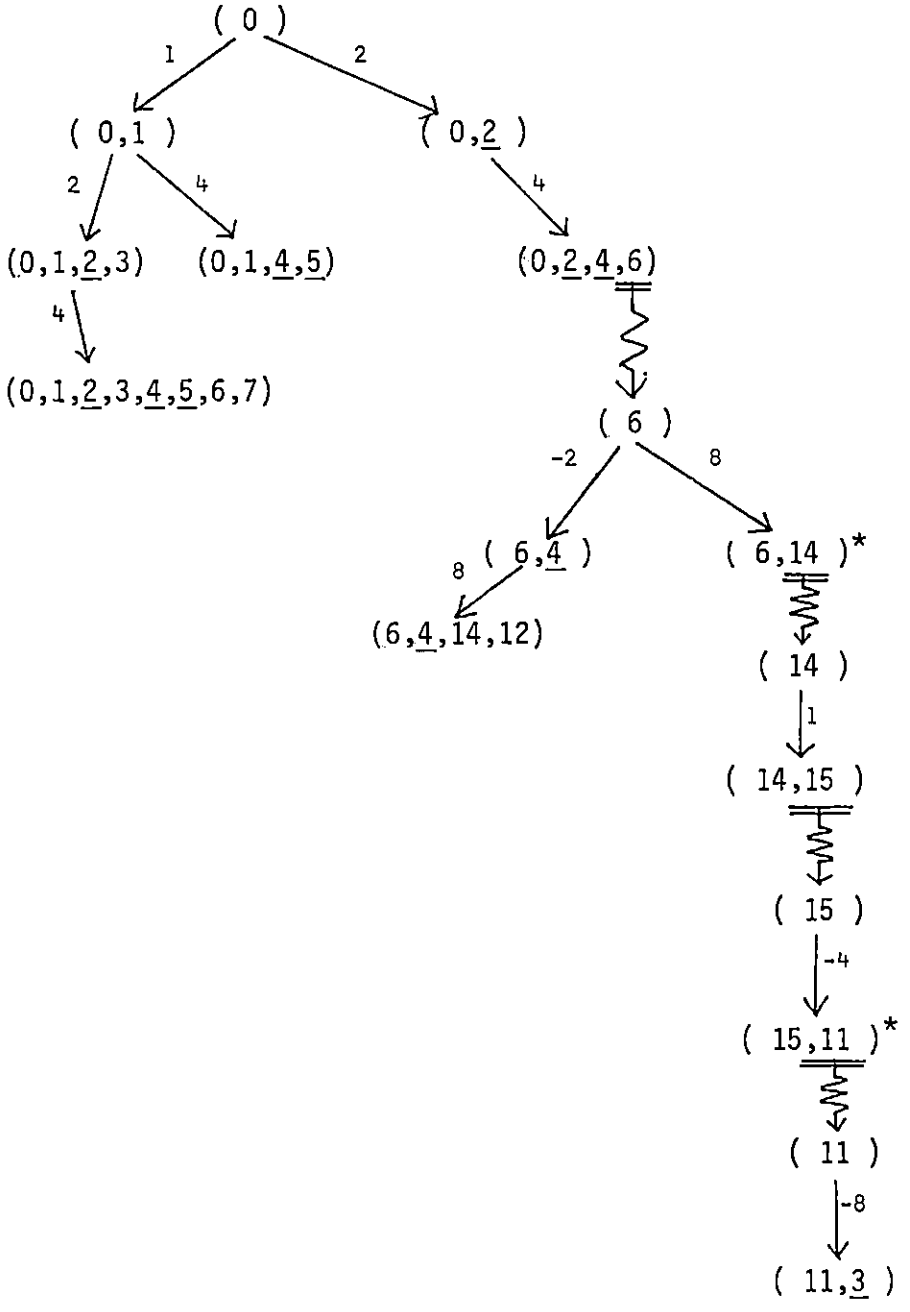

(0)
1
2
(0,1)
(0,2)
2
4
4
(0,1,2,3)
(0,1,4,5)
(0,2,4,6)
4
(0,1,2,3,4,5,6,7)
(6)
-2
8
8
(6,4)
(6,14)*
(6,4,14,12)
(14)
1
(14,15)
(15)
-4
(15,11)*
(11)
-8
(11,3)

Figure 3.3

3.3 SOME FURTHER OBSERVATIONS.

So far the algorithm has been applied based only on the concept of adjacency and RAD trees. While the previous example showed the development of the steps, it did seem to imply quite a lot of search.

In fact, the steps detailed above form only the basic outline, since some simple heuristic considerations make the whole process much more attractive. The author of DSA made some concluding remarks about the introduction of further rules into the algorithm in order to render it more efficient[MCK1, pp.107].

The main idea to be stressed, and which will prove to be vital in the multiple-output extension, is that of "ranking". This simply implies some rule to help determine which TF should be chosen from the available list as a starting TF_0 for the RAD tree. Obviously, if the minimal cover is being sought and it is unique, there should not be any difference in the end result whether TF's are selected in any particular order. And indeed the minimal sum-of-products expression remains the same, while its development can be greatly speeded up. When the cover is not unique, however, in most cases equivalent solutions are obtained even by

choosing the TF_i in arbitrary order, but the amount of work involved can be decreased by proper choices.

The ranking of the TF's is done on the basis of the number of RAD's they each possess. We can observe that if a TF has no RAD's, then it is its own unique cover, and the TF should be chosen as essential prime implicant. If a TF has only one RAD, then the prime implicant formed by itself and the related minterm is the only cover for that TF and must be chosen as essential. For example, if m_1,m_3,m_7 are TF's, then m_1 is +2 adjacent to m_3 and has no other RAD's. Instead m_3 has -2 and +4 as RAD's. If we consider developing m_1, we will immediately come to form PI=(1,3) which exhausts all adjacencies and is therefore essential. The tree is very short (1 node) and the choice can be made unequivocably. Moreover by choosing PI=(1,3), we change m_3 status to XF, and it is clear that the next TF_0 should be m_7. This in turn brings us to select PI=(7,3) as essential, and the cover is complete.

On the other hand if we had started from m_3, then two possible PI's could have been formed, namely (3,1) and (3,7) and both the tree and the choosing process would have been more elaborate. In fact, the use of subtrees would have been necessary, bringing us to the same results, but with

more work done. The two developments are shown in Figure 3.4.

By continuing the reasoning, one can observe that it is easier to develop a tree and make choices if a minterm has two RAD's as opposed to another minterm with three RAD's, and so on. Thus the choice process in Step 1 is augmented by the ranking in the number of adjacencies, and the TF's with fewer adjacencies should be developed first, as they are more likely to expand directly into an essential PI. Of course any choice of essentiality or pseudo-essentiality changes the status of even more TF's to XF's, hence shortening the search more and more since fewer terms are left uncovered, even if not all original TF's have been expanded.

On these premises, an example of a five-variable function will be shown with the proper ranking rule and without it. Let the TF's be 0,1,5,7,10,15,16,26,30. The ranking is as follows:

TF	0	1	5	7	10	15	16	26	30
RAD	+1	-1	-4	-2	+16	-8	-16	+4	-4
	+16	+4	+2	+8				-16	

TF's 10,15,16 and 30 have fewer RAD's, so 10 is chosen as starting one. Figure 3.5 shows all the following expansions. TF=10 expands into PI=(10,26) which is essential.

Mi	1	3	7
RAD's	+2	-2 +4	-4

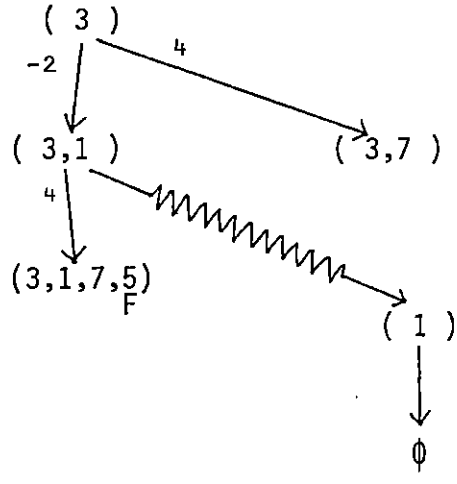
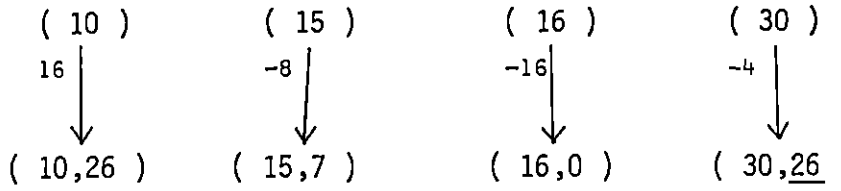


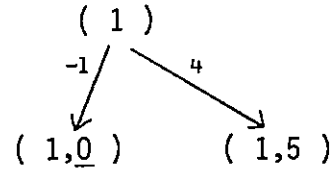
Figure 3.4

Similarly, 15,16 and 30 expand into three other essential PI's: (15,7), (16,0), (30, 26). At this point the list of uncovered TF's is 1,5. TF=1 expands into (1, 0) and (1,5), and the latter is chosen since it contains more TF's. The cover is now complete, after having constructed six prime implicants in order to choose five essential ones, through very short trees and without recursion.

On the other hand, if no ranking had been done, one would have started from TF=0. Figure 3.6 shows this alternate developments. From 0, two PI's are found: (0,1), (0,16), and a subtree from TF=1 is needed in order to make a choice of pseudo-essentiality. Then $\{PI\}_1=\{(1,5)\}$, and $TF_2=5$ and $\{PI\}_2=\{(5,7)\}$. Two more recursions are needed: $TF_3=7$ with $\{PI\}_3=\{(7,15)\}$, and $TF_4=15$. At this point we can stop since $\{PI\}_4$ is empty (15 has no RAD's left), and the first pseudo-essential choice is (7,15) which in turn makes $\{PI\}_2$ useless. Then we choose (1,5) which deletes (0,1) from $\{PI\}_0$. The only PI left (0,16) can be selected as well since no RAD's are left to be used in a subtree from 16.

The rest of the cover is straightforward with another two simple trees, and the total number of PI's constructed here was seven in order to choose five of them, as opposed to having developed only six with the ranking introduced.





Chosen in order :

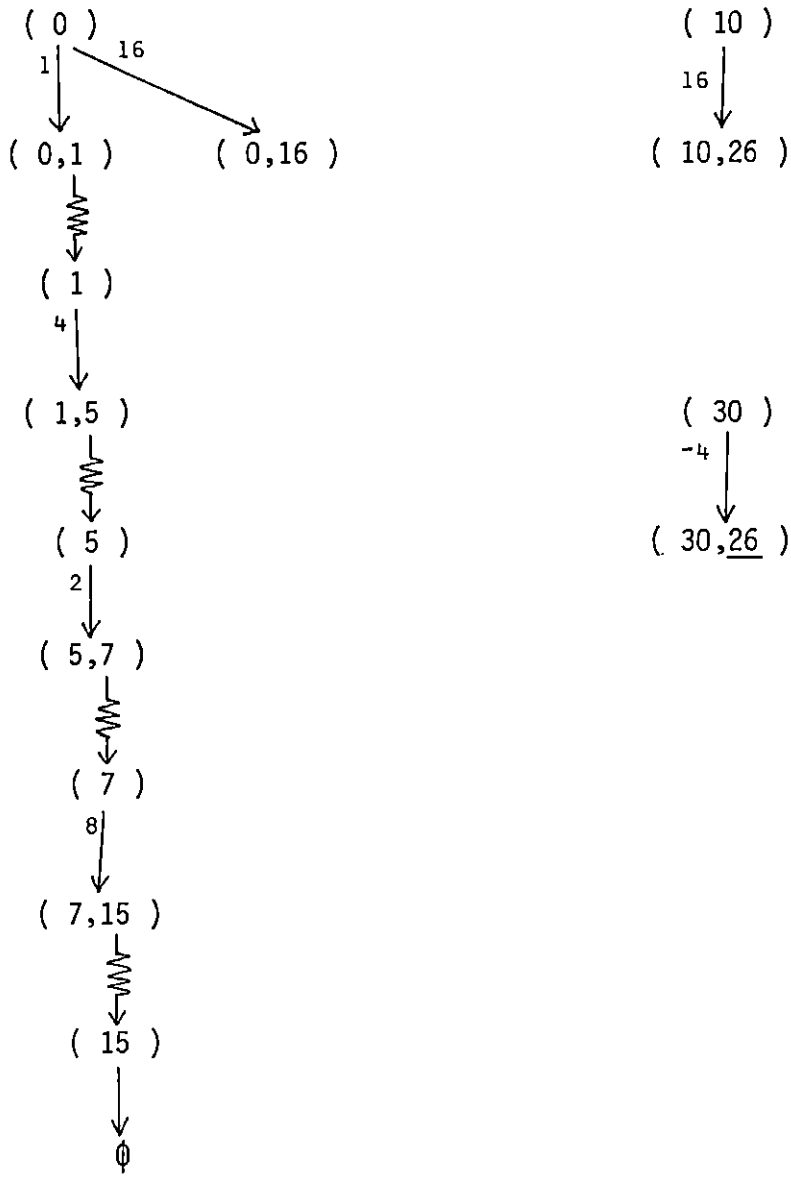
(10,26), (15,7), (16,0), (30,26), (1,5)

Figure 3.5

Since this is a small example the difference is less dramatic than one would expect: larger examples would show how the use of ranking of RAD's turns out to be even more important, while the amount of overhead needed to produce an ordering is not great. Moreover, the RAD's can be stored for later use if possible, and the use or not of subtrees can be introduced as an extra factor.

To conclude this chapter, we will present the Quine-McCluskey method as well as the DSA applied to an example in order to show a more direct comparison of the amount of work involved. Let $Z(x_1,x_2,x_3,x_4)$ = m(1,3,5,7,8,10,14,15) + d(9,11). In Figure 3.7 one can find the DSA applied to it, yielding the final resulting cover of PI's: (5,7,1,3), (8, 9, 10, 11), (14,15, 10,11). In order to achieve this, three PI's were reached through three very short trees.

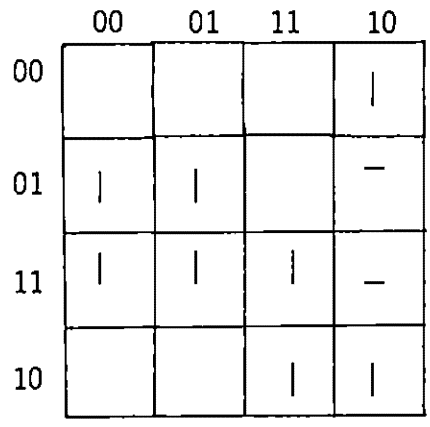
On the other hand Figure 3.8 shows the Quine-McCluskey method, where the PI's are identified and then selected. Three lists of ten, fourteen and five members respectively were used, with five prime implicants constructed, plus five duplicate ones had to be discarded. After this the covering table still has to be done to choose the essential cover of three PI's.



Chosen in order :

(7,15), (1,5), (0,16), (10,26), (30,26)

Figure 3.6



TF's	1	3	5	7	8	10	14	15
#RAD's	3	3	2	3	2	3	2	3

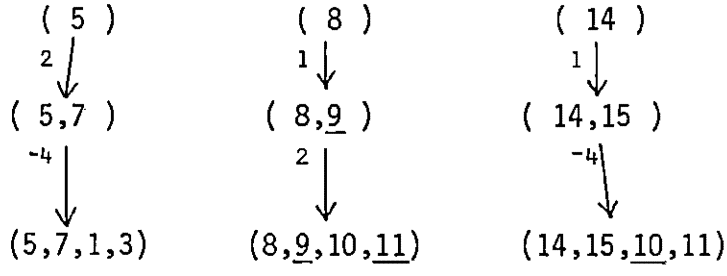


Figure 3.7

1	0001 ✓
8	1000 ✓
3	0011 ✓
5	0101 ✓
9	1001 ✓
10	1010 ✓
7	0111 ✓
11	1011 ✓
14	1110 ✓
15	1111 ✓

1,3	00-1 ✓
1,5	0-01 ✓
1,9	-001 ✓
8,9	100- ✓
8,10	10-0 ✓
3,7	0-11 ✓
3,11	-011 ✓
5,7	01-1 ✓
9,11	10-1 ✓
10,11	101- ✓
10,14	1-10 ✓
7,15	-111 ✓
11,15	1-11 ✓
14,15	111- ✓

1,3,5,7	0--1
1,3,9,11	-0-1
8,9,10,11	10--
3,7,11,15	--11
10,11,14,15	1-1-

	1	3	5	7	8	10	14	15
1,3,5,7	•	•	•	•				
1,3,9,11	•	•						
8,9,10,11					•	•		
3,7,11,15		•		•				•
10,11,14,15						•	•	•

Essential PI's : (1,3,5,7), (8,9,10,11), (10,11,14,15)

Figure 3.8

Chapter IV

DSA2 - THE EXTENSION.

4.1 GENERAL DEFINITION.

The original DSA was never explored by its author beyond a few test cases. The main new idea was to use search trees as opposed to lists and tables for finding the minimal cover of a function,but no other specific rules were developed on how to expand trees or subtrees in order to reduce the amount of overall work involved. The only indication given was that it would be better to initiate trees with TF's having fewer RAD's: specifically the one case which was identified was to choose immediately any TF with no adjacency at all.

Other heuristics have been introduced to make the system more powerful: strict ranking according to number of RAD's and choices among prime implicants on the same tree according to the number of new uncovered TF's that they may provide cover for. This can result in the avoidance of the expansion of some subtrees, since it introduces a further heuristic rule for pseudo-essentiality. In applications to

practical problem it is at times more important that the method might work in a reasonable amount of time given some constraints, even if this means making some heuristic choices along the way that in some example may not provide exactly the same answer, although usually an equivalent one, of a more deterministic approach.

The application to multiple output functions had not been considered at all, and at the beginning of this research some serious doubts existed as to whether DSA could incorporate the use of search-trees spanning more than one boolean function simultaneously.

Here the multiple output synthesis will be presented. The obvious change is of course to introduce flags for minterms and implicants. However, in order to be able to apply a search tree technique, further considerations of the structure of the network are necessary to arrive at a minimal or near minimal answer. A search tree is after all a particular combination of data structure and algorithm such that the total search space which has to be tackled in whatever application can be drastically reduced from a more simple, yet at times inevitable, sequential approach.

When more than one boolean function has to be dealt with simultaneously, the complexity of the problem is greatly in-

creased by the much larger number of choices available as gates. There are possibilities for gates considered for the single functions, while other prime implicants are shared by one or more functions, and the selection of one common gate precludes the selection of another common gate. We have to decide whether and when to select a prime implicant belonging to one or more function, keeping in mind that the objective is the overall minimization. It is not readily clear at first how search trees can incorporate both the reduction of search space and the necessity for choices which cannot be completely predefined. In fact so far DSA did not rely on the search tree for providing selections among prime implicants of apparently equivalent cost: in order to do that subtrees would be developed until a particular expansion brought us to either a point of no return or to a void tree.

The main advantage was in the restricted number of choices presented through the use of trees and the fact that duplicate covers would not be produced. Hence a careful balancing is necessary between what is appropriate or essential on a local basis as opposed to what is more suitable for a wider subset of the network under consideration.

In order to develop an algorithm for multiple-output synthesis a number of heuristic rules have been introduced.

The reason is twofold: an essential need to reduce the complexity of the problem, and a provision for dynamic selection according to the particular situation.

First of all, a renaming seems appropriate: from now on the original single-output method is called DSA1, while our multiple-output extension is called DSA2. Secondly, the cost criterion for the design of circuits is based only on the number of product terms in the two-level sum of products. Thus the determination of an acceptable decomposition for a multiple-output network is based solely on the total number of unique PI's needed, and not on their size (the number of literals involved), or on how many PI's are involved for each function. This, in particular, is the best cost criterion to be used towards a PLA-oriented implementation.

There are some immediate problems that we must address:
-each TF exists only for some F_i, not necessarily for all functions. How do we catalogue them?
-a ranking has to be defined. Which TF and which F_i to choose first?
-each TF has different local RAD's. Which RAD's should be used in the tree expansion?

-conflicts have to be resolved and choices made among PI's in a set where the PI's are on the same tree but belong to different partitions of the network.

The answers to these questions come into two parts: first of all a reevaluation of the ranking scheme to incorporate the new aspects; then the introduction of carefully selected heuristics in order to provide guidelines for expansion and choices in the dynamic part of the process when the trees are actually developed. While heuristic rules are a necessity in any algorithm dealing with a large search space, the effort is focused on keeping their number to a minimum, in order to insure as much as possible that whenever a deterministic path is evident, it is chosen, and not overriden.

The expansion in DSA1 of selected TF's gives a set $\{PI\}_0$ which covers the initial TF_0. If the set $\{PI\}_0$ contains only one member, then that PI is essential to the cover of the given function, since it is the only one covering TF_0. After choosing the particular PI for a minimal cover, care must be taken to change all TF's included in the selected PI to XF's, so that they are never subsequently used for some other expansion.

A definition similar to that of essential prime implicants can be given for multiple-output networks, but the la-

bel of "essential" for a PI must be always further qualified. A PI must also designate for which function(s) it has acquired the desired label of "essential".

Therefore one must always state that a PI_0, as cover for a TF_0, is *essential for* $[F_i...F_m]$, where $[F_i...F_m]$ is a subset of all the switching functions in the network. The expression $[F_i...F_m]$ is called *function qualifier or FQ*. Similarly to DSA1, when the set $\{PI\}_0$ has more than one member, comparisons are made among the members pairwise in order to find any dominance, and possibly lead to the choice of a single pseudo-essential PI, which is then added to the cover for F.

In the multiple-output case, the situation becomes somewhat more complex. The set $\{PI\}_0$ may contain more than one member, but in making further comparisons the function qualifier for each member, i.e. the qualification of the PI relative to the functions it covers, must also be compared and intersected. Care should be taken to compare members which have the same FQ's.

The set $\{PI\}_0$ can then be seen as partitioned into subsets, $\{PI\}_{00}, \{PI\}_{01}, ... \{PI\}_{0r}$, each of which contains PI's qualified for some subset $[F_i...F_m]$ of switching functions. The comparison among members can then only be made as com-

parisons within each $\{PI\}_{0i}$ subset. No member of any subset should be compared to members of another subset of the set $\{PI\}_0$, since the respective FQ's are not equal.

Furthermore we can actually discard from $\{PI\}_0$ all subsets $\{PI\}_{0i}$ where the FQ is itself a subset of any other function qualifier for any other $\{PI\}_{0j}$. The only times when an exception is made and the subset with a smaller FQ is not discarded is when $\{PI\}_0$ contains exactly two members, the second of which has a smaller FQ then the first one, and it has been developed by one further RAD expansion, or when *all* subsets have smaller FQ's than the original desired one.

This is the case which we will call "vertical comparison" between PI's, and the choice cannot be resolved immediately. It is called "vertical comparison" since graphically in the RAD tree the PI's are positioned one above the other, as opposed to being at the same level from a breadth expansion. Instead, new {PI} sets must be developed with subtrees from some TF in the parent node, the one with a bigger FQ following the subtree algorithm which is later explained in more depth.

In some cases, the parent node is the initial TF_0 itself and the development of a subtree has no meaning. Then a TF for a subtree must be chosen from within the child node, the

one with a smaller FQ. More examples of vertical comparisons will be illustrated later to make the concept clearer.

Again, as for DSA1, the case may arise when a $\{PI\}_0$, or a chosen subset $\{PI\}_{0i}$ of it, cannot be reduced to a single member in order to determine a pseudo-essential PI. Then another TF is chosen from within the $\{PI\}_0$ as the origin of a subexpansion, to generate a set $\{PI\}_1$, where $\{PI\}_0$ and $\{PI\}_1$ must have at least one common TF member with the same FQ if at all possible.

The $\{PI\}_1$ is treated in the same fashion as the $\{PI\}_0$, and, if it can be reduced to a single member, it will permit backtracking while changing TF's to XF's along the way. This means that some members in $\{PI\}_0$ will be discarded as they will contain only XF's and $\{PI\}_0$ will eventually reduce to a single entry giving a pseudo-essential cover.

If, however, $\{PI\}_1$ cannot be reduced, the procedure continues with $\{PI\}_2,...\{PI\}_i$ in the same way, until a resolution is obtained. The problem of cycles still exists when the development of some TF with some FQ eventually creates a PI identical to some PI produced earlier. In order for this to be a cyclic situation the FQ's must be the same as well, which brings us to the interesting observation that cycles are not quite as disrupting in DSA2 as they were in DSA1.

In fact, both the PI generated and its FQ must be repeated for the problem to arise in the first place. Moreover,even if local cycles in one function or in a subset of functions in the network exist, the simultaneous synthesis often makes a chosen common PI the resolution term for cycles, thus needing no further intervention.

When trying to reduce a $\{PI\}_i$ to a single member, we mentioned deleting some terms whose FQ was a subset of the other FQ's. Basically, since the solution which spans the most number of functions in the network is being searched for, one should consider as long as possible the PI's which are "widest" in their cover. This means looking first for the gates which can be used by the largest number of functions, i.e. the largest function qualifier set.

Once a TF_0 is selected with a large FQ, that FQ should be adhered to in the shrinking of the $\{PI\}_i$ set. That is why the FQ's which are smaller subsets should not stay there and be considered as valid choices. In this way the PI's in common are identified first; the $\{PI\}_{0i}$ subsets which were discarded will eventually be generated again if necessary, when the DSA2 algorithm will be considering smaller parts of the network or not be produced at all if, by other covers, all the TF's involved have by then changed their status to XF's.

4.2 THE EXTENDED ALGORITHM.

In DSA1 the adjacencies for a TF and their directions were the key factors in the expansion of the RAD tree. The same is true in DSA2 where, however, every TF has more RAD's attached to it. For every TF we must have:

1. the total number of adjacencies for all functions.
2. the number of adjacencies for each function.
3. the number of functions for which that minterm is a TF and which ones they are.

In DSA1 the rule which made the method work so speedily was based on the criterion of point (2), from which a ranking was extracted. That in turn directed the choice of the TF's to be expanded towards the ones with fewer RAD's first.

In DSA2 we must extract a similar ranking but based on all three points. The heuristic choice in DSA2 are defined by the following two criteria:

a) TF for highest number of functions (from point 3).

b) TF with lowest number of overall adjacencies (from point 1).

Point (a) should be used first and point (b), if needed, applied to the resulting set of TF's from point (a).

After these rules have been properly applied, a choice can be made of a TF, or alternatively a set of equivalent

choices are presented, and one of the TF's can be selected arbitrarily. Then one further heuristic selection should be made to aid the RAD tree expansion. Except towards the end of the whole procedure, a chosen TF will in general have associated to it a FQ set whose cardinality is greater than one. A decision should be made of which function from the set $FQ=[F_1,...,F_m]$ of that TF is to be the "directing window" for the tree.

In fact a TF_0 will have different RAD's for each of $F_1...F_m$, hopefully with some common ones which will lead to common factors. However it makes a definite difference which adjacencies for which function are chosen for the tree expansion. Obviously only one set of RAD's from a particular F_i can be used to avoid inconsistencies.

Another form of ranking is required and its rules are as follow:

c) expand over the F_i for which the chosen TF has fewer local adjacencies.

d) expand over the F_i in which fewer TF's are left uncovered.

e) or choose in cardinal order.

These rules are nested in their logical scope: point (c) should be applied first, and point (d) as further alterna-

tive applied to the result from point(c), while point (e) is last. They help in developing shorter trees (point(c)), and in covering completely the smaller functions first (point(d)), which usually have a lower probability of providing common factors.

A further extension comes into being during tree expansion as opposed to ranking time. While developing depth-first in a tree following the RAD's along a F_i for a TF_0 having FQ_0, at a particular node n_r the set of function qualifiers $[F_j...F_r]$, called FQ_r, might become a subset of the original FQ_0. The depth-first development should be halted here, even if the list of RAD's is not exhausted. This is one of the major departures from DSA1.

At this point, instead, a backtrack to node n_{r-1} is necessary and from there another branch development initiated. If from level n_{r-1} to level n_r another node is found with $FQ=FQ_0$, the search continues depth-first along that branch.

Otherwise further backtracking will be needed to level n_{r-2} from which all nodes at level r-1 with $FQ=FQ_0$ will be expanded. It is these nodes at level r-1 which will form the $\{PI\}_0$. The previously expanded node at level r can now be discarded. Even after the $\{PI\}_0$ of level r-1 has been reduced and resolved, the TF's remaining uncovered at that

level r node need not be considered as "left" on the tree. They should not be expanded into subtrees since they do not "belong" to this particular subset of function qualifier and will be dealt with later.

The reason for this is again that one wants to keep working for as long as possible with the largest FQ and stop expanding the PI's when their expansion depth first in the tree is detrimental to the FQ's.

The breadth-first expansion to a node at level n_{r-2} can happen only if the tree goes farther up than node n_{r-1}. That is, if the tree is of depth one, and n_r was both the first child and a leaf node, then no further backtracking is possible beyond the root node which is at level n_{r-1}. Here the case may arise of having a set of PI's at level two with a smaller FQ than the starting TF, and no other choices exist. Thus the only common gate seems to be the TF itself.

In order to decide whether that is the proper choice, the PI with the smaller FQ's are retained and subtrees developed from within them, until backtracking makes the selection.

The only other possibility arises when the backtracking at node n_{r-1} and/or n_{r-2} gives empty trees, i.e. when there are no more RAD's left. In this case we have a "vertical

comparison" as mentioned earlier. The decision has to be made between two PI's, with different number of TF's and XF's, with the second one having a FQ set whose cardinality is smaller than the cardinality of the original FQ_0.

For this situation subtree(s) from some TF in the second PI must be developed to aid in the choice. The subtree(s) are dealt with in the usual manner as they are for other "horizontal comparisons". Proper backtracking should resolve a pseudo-essential cover both for the original FQ_0 and for FQ_r.

4.2.1 Other cases.

A last heuristic criterion for the ranking and the selection of best expansion choices of TF's should also be introduced. This can be seen as some sort of presort based on some particular features which become apparent during the ranking of the TF's. We deal here with some cases in which certain TF's should be chosen for expansion first of all, disregarding their position relative to the overall network. These are simple trees that we like to expand first, since any result in DSA will bring its effect into the overall structure and probably reduce the search.

At the time of the first ranking, a preselection should be made for those TF's where the number of total adjacencies overall is less than or equal to the number of functions in which they appear as TF's. These TF's are of special interest as they will most likely define essential PI's for one or more functions, through very easy expansions, and hence cleaning up some other TF's, by changing them to XF's, for which tree choices would have been, by themselves, more of a problem.

The rationale behind this criterion is easily explained. If a TF appears in one function and has one overall adjacency for example, then its $\{PI\}_0$ will contain only one member which is essential for that function. That PI can be immediately added to the cover without further consideration.

If a TF appears in two functions and has two overall adjacencies, only two cases are possible: either the RAD is the same for both functions or it is not. When they are the same, a common PI has been found and can be selected. When they are not the same, then two different PI's, both easily found and both essential, can be added to the cover.

In general, when the total number of adjacencies is less than the number of function appearances, by choosing to expand along the F_i where the number of local RAD's is less

than in the other functions (one will have zero RAD's), essential PI's are readily determined. When the two numbers are the same, the distribution and the equality of the RAD's will determine how easily choices are made, but in most cases with the creation of simpler and shorter trees.

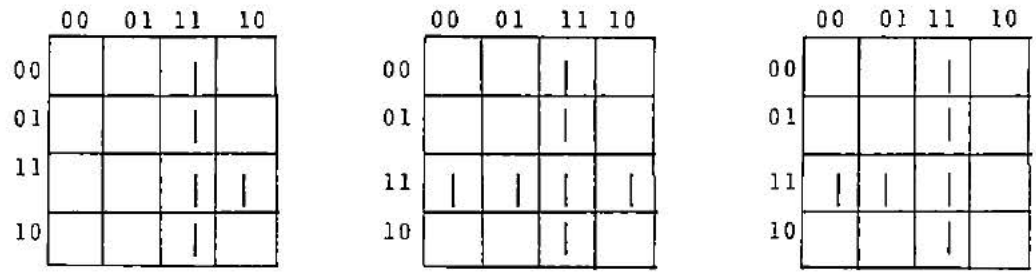
As soon as the number of RAD's becomes greater than the number of function appearances, the possible routes which could be taken in the minimization process increase considerably. That is why it is important that we exploit any chance to take advantage in the algorithm of some feature of the inner structure of the particular network. Certainly one of the aspects has proved to be the selection of those particular TF's which have such a high probability to give us efficient and straightforward results.

4.2.2 Examples.

A couple of examples to clarify matters are helpful. First an example where the ranking heuristics according to points (a), (b), (c), (d) are shown.

EXAMPLE 1. (See Figure 4.1)

This is an easy example where m_{12} is chosen first since it is the initial one in the ranked set of TF's which appear in three functions and have six overall adjacencies. The



$F_1=(11,12,13,14,15)$ $F_2=(3,7,11,12,13,14,15)$

$F_3=(3,7,12,13,14,15)$

Mi	3	7	11	12	13	14	15
#F	2	2	2	3	3	3	3
#RAD's	3	4	3	6	6	6	10
local RAD's	-	-	1	2	2	2	3
	2	2	2	2	2	2	4
	1	1	-	2	2	2	3

(12)[1,2,3]F_1 → (12,13)[1,2,3] → (12,13,14,15)[1,2,3]

(3)[2,3]F_3 → (3,7)[2,3]

(11)[1,2]F_1 → (11,15)[1,2]

PI's : (12,13,14,15) 1,2,3
(3,7) 2,3
(11,15) 1,2

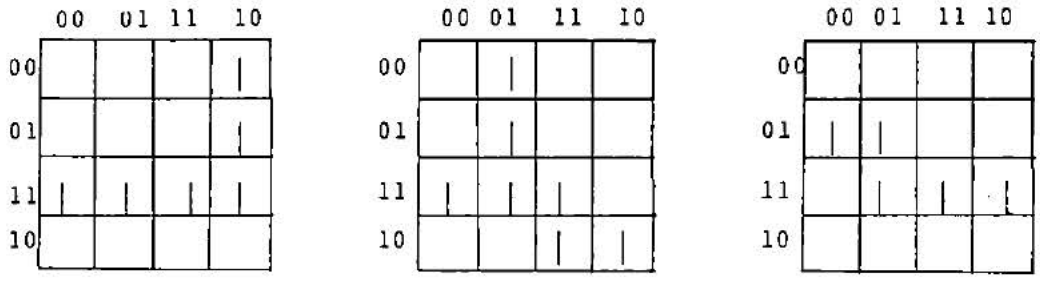
Figure 4.1

tree for TF_0=12 is developed along F_1 since the number of local adjacencies is the same (2) for all three functions, but F_1 has fewer TF's left uncovered.

The PI formed (12,13,14,15)[1,2,3] is found to be essential (all RAD's expanded, no loss in FQ). All TF's included are changed to XF's and the ranking reevaluated. Now the choice is between m_3 and m_{11}; the tree is done for m_3, over F_3 (fewer local RAD's), and an essential PI=(3,7)[2,3] is found. The last choice of m_{11} after expansion gives the complete cover by adding PI=(11,15)[1,2].

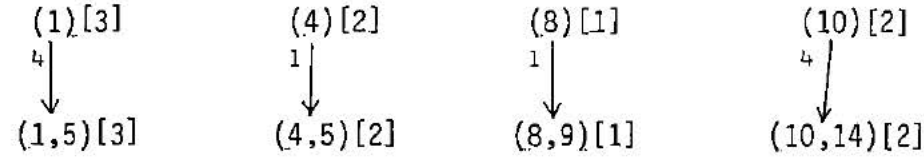
EXAMPLE 2. (See Figure 4.2)

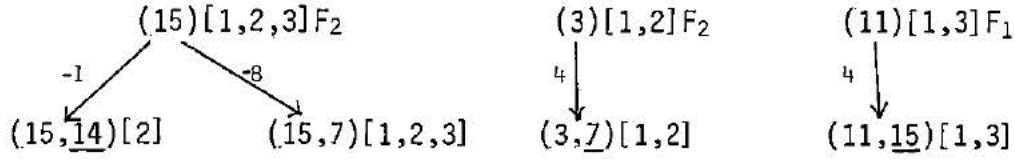
Here four special cases are noted from the ranking table, namely m_1, m_4, m_8, m_{10}, which all appear in only one function with one overall RAD. Four little trees are then developed, leading immediately to four essential PI's, (1,5)[3], (4,5)[2], (8,9)[1], (10,14)[2]. Notice how other TF's have been quickly turned to XF's (m_5, m_9, m_{14}). The next ranking has thus a smaller choice and m_{15} becomes the TF_0, because it has the highest number of function appearances, three, and the lowest number of RAD's, six. From this tree a set $\{PI\}_0$ is obtained containing {(14, 15)[2], (7,15)[1,2,3]}. Since the FQ=[1,2,3] for (7,15) is larger



$F_1=(3,7,8,9,11,15)$ $F_2=(3,4,5,7,10,14,15)$ $F_3=(1,5,7,11,15)$

Mi	1	3	4	5	7	8	9	10	11	14	15	
#F	1	2	1	1	3	1	1	1	2	1	3	
#RAD's	1	3	1	2	7	1	2	1	3	2	6	
local RAD's	-	2	-	-	2	1	2	-	2	-	2	F_1
	-	1	1	2	3	-	-	1	-	2	2	F_2
	1	-	-	-	2	-	-	-	1	-	2	F_3





PI's : (1,5)[3], (4,5)[2], (8,9)[1], (10,14)[2],(15,7)[1,2,3],
(3,7)[1,2], (11,15)[1,3]

Figure 4.2

then the other FQ and corresponds to the initial one for m_{15}, the set $\{PI\}_0$ can be reduced to only (7,15)[1,2,3], which becomes pseudo-essential. The last ranking gives development to m_3 and m_{11} for the last two PI's needed for the cover, (3,7)[1,2] and (11,15)[1,3].

4.3 SUMMARY OF METHOD AND RULES.

The manual method associated with DSA2 is based on the ideas proposed in DSA1. However, the multiple-output situation leads to a much more complex algorithm which, if it is to be useful for practical problems, has to make extensive use of carefully developed heuristics.

A step by step summary of the algorithm can be presented as follows:

1. Obtain the RAD's for all TF's left. If none are left the process is terminated. The ranking table must include:
   i) number of function appearances
   ii) number of RAD's and list of RAD's for each function
   iii) overall total number of RAD's.
2. Eliminate the special cases where (i) is greater or equal to (iii) by constructing the proper trees. Within this category start with lowest (ii) and pro-

ceed in ascending order. Change all TF's to XF's as they are covered.

3. Select TF_0 for the next RAD tree expansion from the ranking table. If none are left the process is terminated. The TF chosen must be one occurring first in the ranking table subset formed by:
   i) number of function appearances is the maximum in the ranking
   ii) number of RAD's is minimum overall within the previous selection.
4. Select along which of the functions from the function qualifier for that TF_0 the tree should expand by choosing:
   i) the F_i for which the TF_0 has fewer local RAD's
   ii) the F_i which has fewer TF's left uncovered.
5. Expand TF_0 forming $\{PI\}_0$
6. Reduce $\{PI\}_0$ to only one member and add it to the global cover with its FQ. Go to Step 3.
7. If $\{PI\}_0$ cannot be reduced, select a TF_1 from one member of $\{PI\}_0$ (possibly the one which appears only once and/or the one with fewer overall RAD's). Expand TF_1 along an F_j as per Step 4.
8. Continue as per Steps 6 and 7 until one set of $\{PI\}_j$ can be reduced to one member. Then select that one

as pseudo-essential, change the appropriate TF's into XF's, delete all other members of $\{PI\}_i$, i<j, which now contain only XF's, while retaining as pseudo-essential all members of $\{PI\}_i$, i<j, which contain uncovered TF's. The process backtracks for every $\{PI\}_k$ set, (j-1)>k>1. Go to Step 3.

9. If no $\{PI\}_k$ has been reduced to one member and a $\{PI\}_j$ has been obtained such that one of its members is equal to a member of some $\{PI\}_i$, i<j, then a cyclic chain has been discovered. The chain should now be resolved, then Step 8 executed.

The whole procedure may seem rather complex at first reading. But it should be remembered that multiple-output minimization is a rather difficult problem. In the next chapter a few examples will be presented, both to show concrete results of DSA2 and also to indicate that with a bit of practice the set of rules lend themselves quite nicely to a manual algorithm. Certainly it is actually possible to use DSA2 by hand without the great psychological burden that something like Quine-McCluskey would place on an individual.

Chapter V

EXAMPLES AND RESULTS.

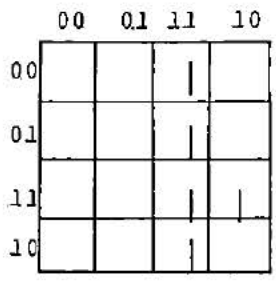
In this chapter a few examples are illustrated both by commentary and by figures to precisely show how DSA2 works, at least in the manual application.

The cases shown are relatively small as we believe that any multiple-output synthesis of any size would not be attempted by hand anyway. The notation used is quite straightforward: round parentheses enclose PI's whose TF's and XF's are listed by their decimal value; square parentheses enclose their respective flags. The functions represented in the flags are listed as decimals 1,2,...,n for F_1, F_2,...F_n.

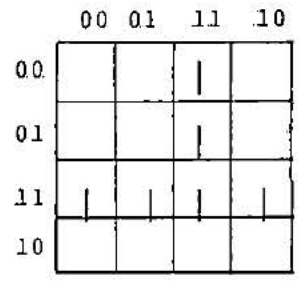
Karnaugh maps are used both as definitions of the functions involved and as a visual aid to the solution.

EXAMPLE 3. (See Figure 5.1)

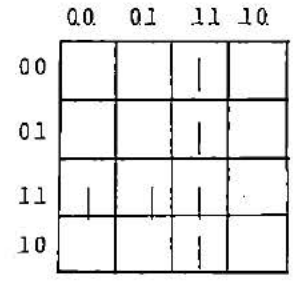
In this example the use of subtrees is shown. There are no special cases and the choice falls on m_{12} as first TF_0.



$F_1=(11,12,13,14,15)$

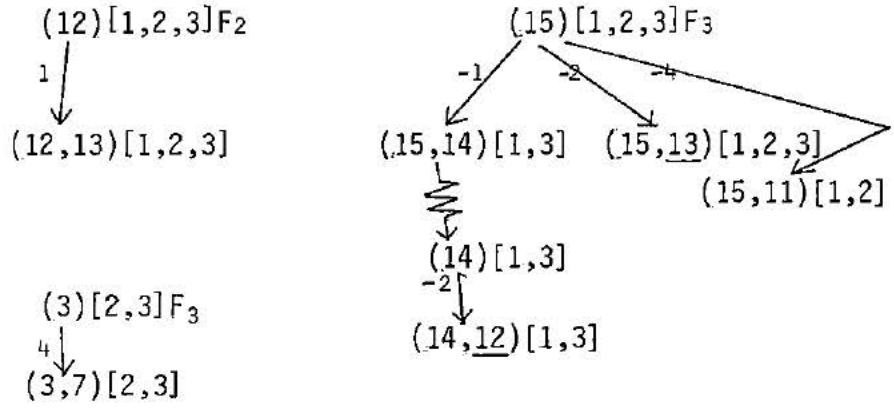


$F_2=(3,7,11,12,13,15)$



$F_3=(3,7,12,13,14,15)$

Mi	3	7	11	12	13	14	15	
#F	2	2	2	3	3	2	3	
#RAD's	3	4	3	5	6	4	9	
local RAD's	-	-	1	2	2	2	3	F_1
	2	2	2	1	2	-	3	F_2
	1	2	-	2	2	2	3	F_3



PI's : (12,13)[1,2,3]; (15,14)[1,3]; (15,11)[1,2];
(3,7)[2,3]

Figure 5.1

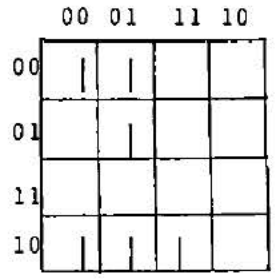
It has the fewest RAD's over the largest number of functions. We expand the tree along the adjacencies of F_2 since there the number of RAD's is smaller than in the other functions in the FQ. The result is an essential PI=(12,13)[1,2,3]. Next m_{15} is expanded along F_1 which is the function in its FQ with the fewest TF's left uncovered. The first PI produced is (14,15)[1,3] and the FQ is a subset of the original FQ_0=[1,2,3]. Hence backtracking occurs to form another PI=(15, 13) [1,2,3] which cannot go any further in depth. While this last PI has an acceptable FQ, notice that m_{13} is now an XF (from the PI choice of the previous tree) and therefore does not add anything to the cover. Continuing breadth-first, PI=(15,11)[1,2] is formed and the set $\{PI\}_0$ becomes {(15,14)[1,3], (15, 13) [1,2,3], (15,11)[1,2]}. A subtree should now start from either m_{14} or m_{11} and in this case we will randomly choose m_{14}. The expansion is along F_1 (fewer TF's uncovered) and the only result is (14, 12) [1,3] which does not add to the cover and in a subtree is discarded. This in turn induces the choice of (15,14)[1,3] as pseudo-essential, and makes (15,13) [1,2,3] completely redundant. Now $\{PI\}_0$ contains only (15,11)[1,2] which can be selected as well. Note here as an aside, that if the choice of this last PI appears dubious, the technique of subtrees can be reapplied starting from m_{11} and indeed the previous result will be confirmed.

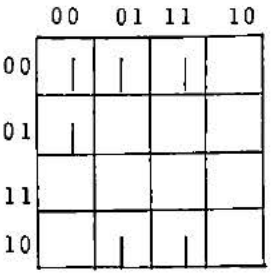
The new ranking table points to m_3 which gives the last PI for the network (3,7)[2,3]. The result given for this circuit by Roth [ROC1] differs in PI=(14,15)[1,3] which is instead chosen as (12,13, 14,15)[1,3]. While this discrepancy might make a difference with a classical gate model, a PLA realization makes the two solutions equivalent.

EXAMPLE 4. (See Figure 5.2)

This example is interesting because, as noted by Roth [ROC1], choosing the maximum number of common terms brings a worse solution. DSA2 however is able to recognize the fact and determine the better coverage. There are two special cases, m_1 and m_5, which provide two essential PI's (0,1)[2] and (4,5)[1]. Then after the ranking, m_{14} produces (6,14)[1,2] and again m_0 produces (0,2, 4,6) [1] and m_{12} produces (4, 6, 12, 14) [2]. Notice how when doing the last ranking, the TF's for m_0 and m_4 are listed as such for one function only: they have already changed their status to XF's for the other function by previous trees, and their ranking flag has to change dynamically.

If the solution with the maximum number of common gates had been chosen we would have had (0,4)[1,2], (6,14)[1,2], (4,5)[1], (2,6)[1], (0,1)[2], (4,12)[2], which gives a total of six terms.





F_1=(0,2,4,5,6,14) F_2=(0,1,4,6,12,14)

Mi	0	1	2	4	5	6	12	14	
#F	2	1	1	2	1	2	1	2	
#RAD's	4	1	2	6	1	5	2	3	
local	2	-	2	3	1	3	-	1	F_1
RAD's	2	1	-	3	-	2	2	2	F_2

Special : 1 5

(1)[2] —(-1)→ (1,0)[2]

(5)[1] —(-1)→ (5,4)[1]

#F=2 : 6 14
#RAD's 5 3

(14)[1,2]F_1 —(-8)→ (14,6)[1,2]

#F=1 : 0 2 4 12
#RAD's 2 2 3 2

(0)[1] —(2)→ (0,2)[1] —(4)→ (0,2,4,6)[1]

(12)[2] —(2)→ (12,14)[2] —(-8)→ (12,14,4,6)[2]

PI's : (0,1)[2]; (4,5)[1];(6,14)[1,2];
(0,2,4,6)[1]; (12,14,4,6)[2]

Figure 5.2

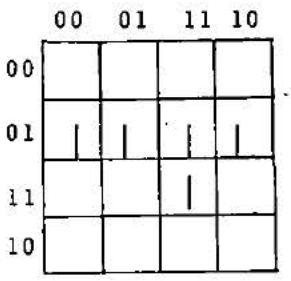
EXAMPLE 5. (See Figure 5.3)

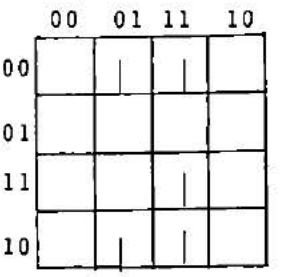
Here we show both a subtree and a vertical comparison. The first TF_0 is m_{15} which expands into only (15,13)[1], where the FQ is smaller than the original one [1,2]. No other possible choices are presented from the tree and we can only deal with either (15) alone or (15,13), which lies directly below it (vertically) with a smaller FQ.

A subtree is then initiated from within the node with the smaller FQ, i.e. m_{13}. The subtree determines pseudo-essential PI=(13,9,5,1)[1], and by backtracking only (15)[1,2] is left to be chosen as common gate. The selection of the last PI=(4,6,12,14)[2] is straightforward.

EXAMPLE 6. (See Figure 5.4)

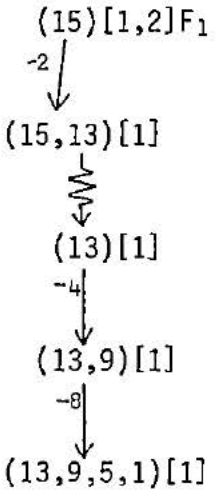
The first two trees for m_0 and m_5 proceed smoothly; the third tree for m_{10} gives a choice between PI=(10,14)[1,2] and PI=(10, 2) [1,2], and the former is chosen because it contains more TF's. The fourth tree however produces a set $\{PI\}_0$ = {(15, 14), [1,2], (15, 13) [1,2], (15, 14,13, 12)[1]}. The choices here are totally equivalent between the first two PI's in the set, or for that matter, considering the presence of XF's, with just (15)[1,2] alone. It is

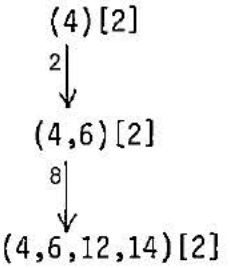




F_1=(1,5,9,13,15) F_2=(4,6,12,14,15)

Mi	1	4	5	6	9	12	13	14	15	
#F	1	1	1	1	1	1	1	1	2	
#RAD's	2	2	2	2	2	2	3	3	2	
local	2	-	2	-	2	-	3	-	1	F_1
RAD's	-	2	-	2	-	2	-	3	1	F_2





PI's : (13,9,5,1)[1]; (15)[1,2]; (4,6,12,14)[2]

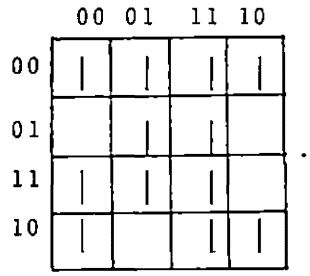
Figure 5.3

left unshown here, but the choice could be aided in another way: by not throwing away the last PI in the set and using a subtree on m_{12}. However, the end result produces, after a lot of work, an equivalent cover.

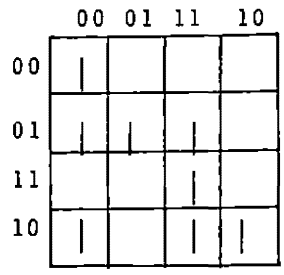
The next few trees show quite a set of different choice situations, some of which are again equivalent.

EXAMPLE 7. (See Figure 5.5)

At the beginning of this network there is one special case for m_{14} which obtains (14,12)[2]. The pseudo-essential choice for the tree from m_1 produces (1,9)[1,2]. When we come to m_4, however, we find a vertical comparison between (4)[1,2] and (4,5)[2]. The subtree generated from m_5 only partially solves the cover for m_4. In fact PI=(5,4)[2] added to the global cover still leaves m_4 as a TF for F_1. While from heuristics we may decide to discard certain PI's from {PI} sets when their FQ's are smaller, even if they do contain some unresolved TF, or when their FQ is valid but they contain only XF's, we note here that m_{12} has XF status only relative to F_2 and a subtree can be expanded with respect to F_1. The next PI from the subtree is (12,13)[1] and this covers m_{12} completely. This leaves m_4 unresolved for F_1 but the future rankings will take care of that. The rest of the development is straightforward.

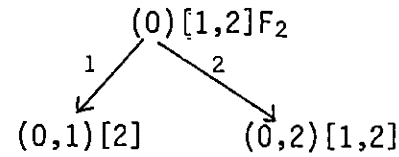


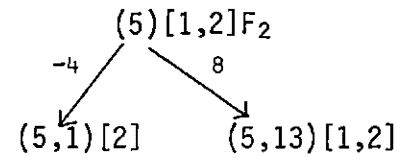
F =(0,2,3,4,5,7,8,
10,12,13,14,15)

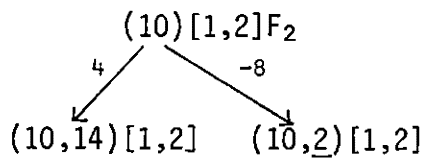


F =(0,1,2,5,10,13,14,15)

Mi	0	1	2	3	4	5	7	8	10	12	13	14	15
#F	2	1	2	1	1	2	1	1	2	1	2	2	2
#RAD's	5	2	5	2	3	5	3	3	5	4	5	5	5
local RAD's	3 2	-	3 2	2 -	3 -	3 2	3 -	3 -	3 2	4 -	3 2	3 2	3 2







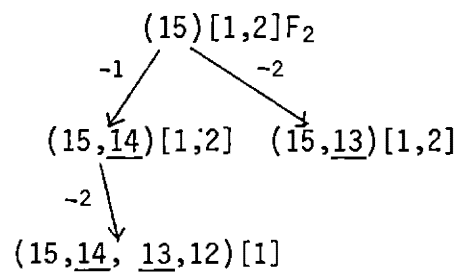
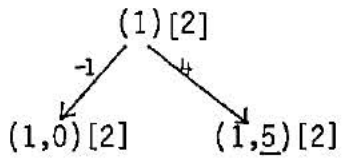
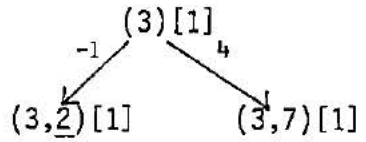
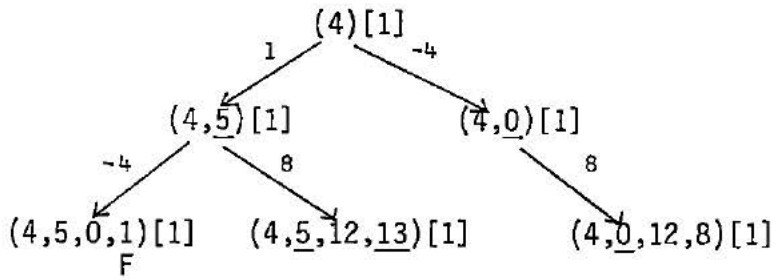


Figure 5.4 /continued

Figure 5.4 /continued







PI's : (0,2)[1,2]; (5,13)[1,2]; (10,14)[1,2]; (15,14)[1,2]

EXAMPLE 8. (See Figure 5.6)

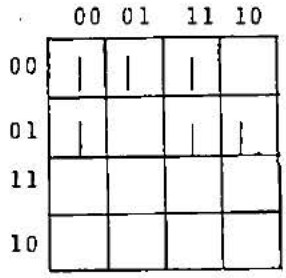
This example is the only network of some size examined here. We deal with seven functions of five inputs, and it turns out to be quite a straightforward application of DSA2. However, the resulting cover returns nine PI's which is better than the eleven PI's previously thought to be necessary when the synthesis was done by other methods, involving as well a considerable amount of extra work.

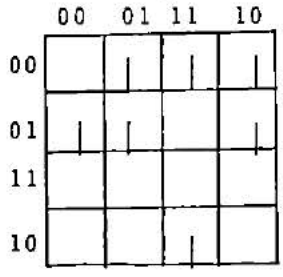
The major interest of the alternate and better solution given by DSA2 lies in the redundancies introduced for each single function realization. In a classical gate-model network, more lines and OR gates would be introduced possibly offsetting the advantage of having fewer PI's. However for a PLA this new set of terms is optimal.

EXAMPLE 9. (See Figure 5.7)

This example shows a direct comparison between DSA2 and Quine-McCluskey. Both procedures are applied and the resulting amount of work is graphically displayed in Figures 5.7 and 5.8.

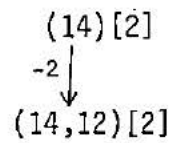
EXAMPLE 10. (See Figure 5.9)

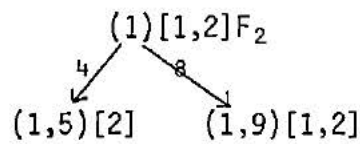


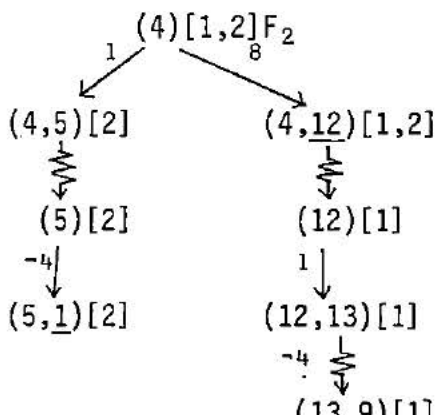


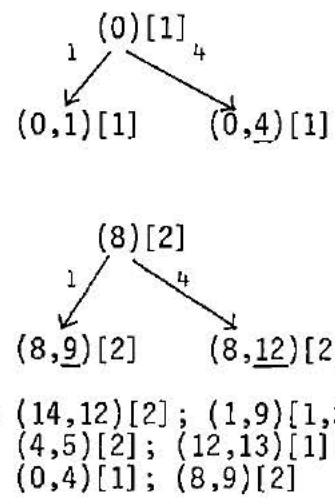
$F_1=(0,1,4,9,12,13)$ $F_2=(1,4,5,8,9,12,14)$

Mi	0	1	4	5	8	9	12	13	14	
#F	1	2	2	1	1	2	2	1	1	
#RAD's	2	4	4	2	2	4	5	2	1	
local RAD's	2	2	2	-	-	2	2	2	-	F_1
	-	2	2	2	2	2	3	-	1	F_2









PI's : (14,12)[2]; (1,9)[1,2];
(4,5)[2]; (12,13)[1];
(0,4)[1]; (8,9)[2]

Figure 5.5

This is the same example given earlier in Chapter 2 as an introduction to multiple-output synthesis. The first figure is a repetition of the Quine-McCluskey procedure, with the three lists which generate a total of five prime implicants, taking care to avoid the insertion of corresponding five duplicates. The covering table selects three of them as essentials.

The DSA2 applied in Figure 5.10 shows the ranking and the three, rather simple, trees necessary for the whole search.

In the Appendix more examples can be found as worked out by the implemented version of DSA2, details of which can be found in Chapter VI.

F_1=(1,3,6,7,9,11,14,15,17,19,22,23,25,27,30,31)

F_2=(2,6,10,14,18,19,22,23,24,26,28,30)

F_3=(3,5,6,7,11,13,14,15,21,22,24,27,28,29,30,31)

F_4=(1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31)

F_5=(2,6,10,14,18,22,26,30)

F_6=(0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30)

F_7=(1,5,9,13,17,21,25,29)

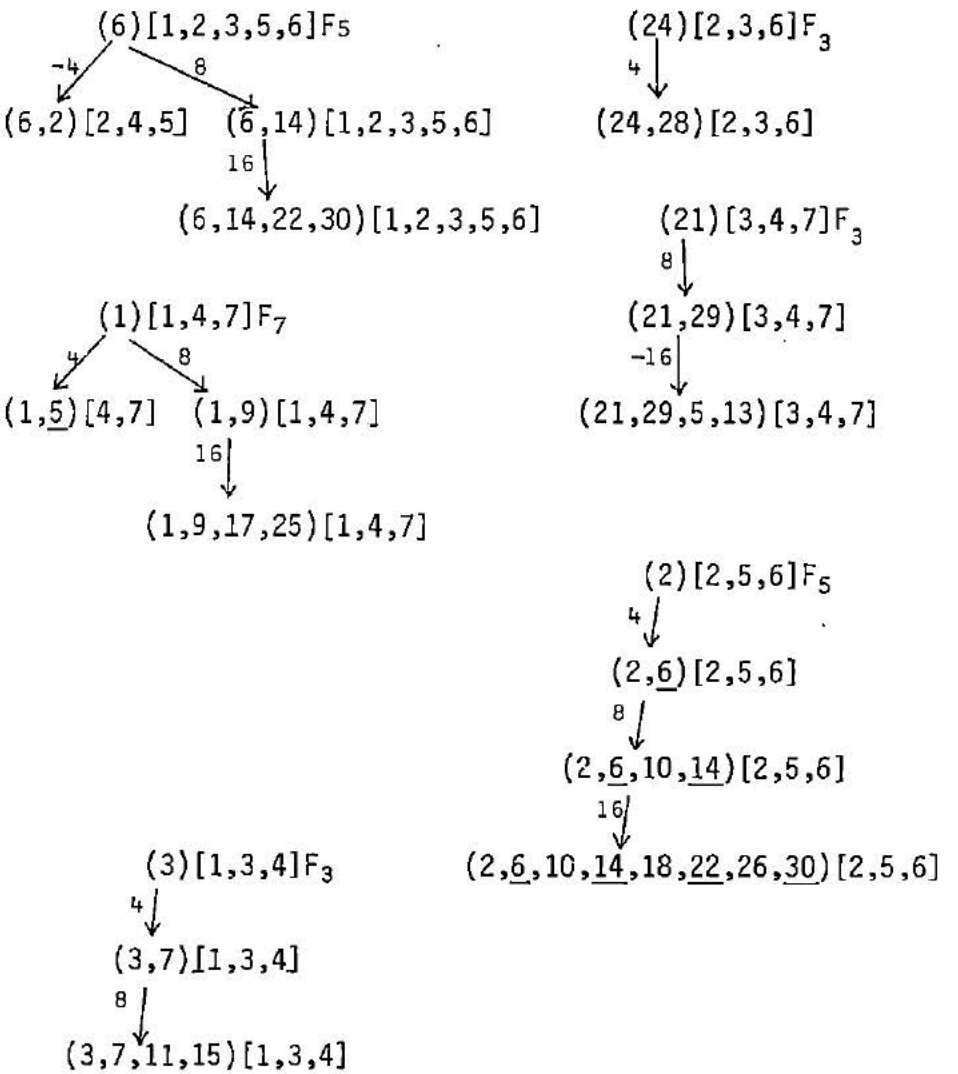
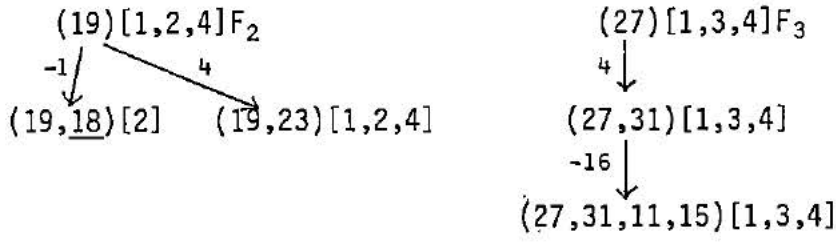
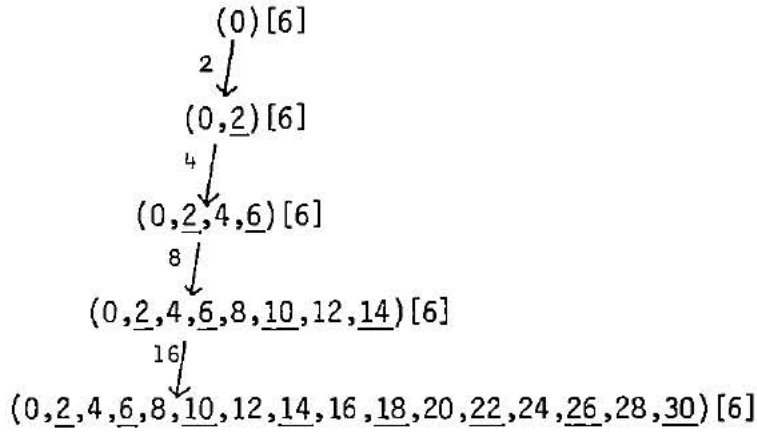
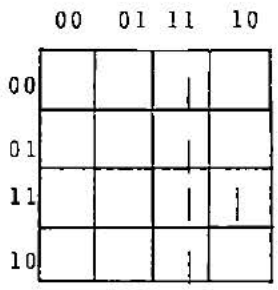


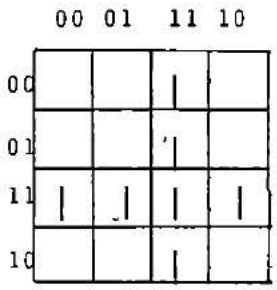
Figure 5.6 /continued

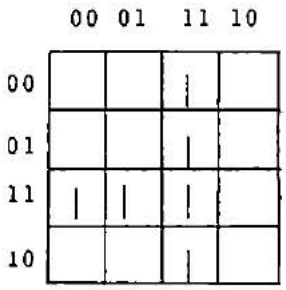




PI's : (6,14,22,30)[1,2,3,5,6]; (24,28)[2,3,6];
(5,13,21,29)[3,4,7]; (1,9,17,25)[1,4,7];
(2,6,10,14,18,22,26,30)[2,5,6];
(3,7,11,15)[1,3,4]; (19,23)[1,2,4];
(11,15,27,31)[1,3,4];
(0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30)[6]







F_1=(11,12,13,14,15) F_2=(3,7,11,12,13,14,15) F_3=(3,7,12,13,14,15)

Mi	3	7	11	12	13	14	15	
#F	2	2	2	3	3	3	3	
#RAD's	3	4	3	6	6	6	10	
local RAD's	-	-	1	2	2	2	3	F_1
	2	2	2	2	2	2	4	F_2
	1	2	-	2	2	2	3	F_3

(12)[1,2,3]F_1 —1→ (12,13)[1,2,3] —2→ (12,13,14,15)[1,2,3]

(3)[2,3]F_3 —4→ (3,7)[2,3]

(11)[1,2]F_1 —4→ (11,15)[1,2]

PI's : (12,13,14,15)[1,2,3]; (3,7)[2,3]; (11,15)[1,2]

Figure 5.7

3	0011	[2,3] ✓
12	1100	[1,2,3] ✓
7	0111	[2,3] ✓
11	1011	[1,2] ✓
13	1101	[1,2,3] ✓
14	1110	[1,2,3] ✓
15	1111	[1,2,3] ✓

3,7	0-11	[2,3]
3,11	-011	[2] ✓
12,13	110-	[1,2,3] ✓
12,14	11-0	[1,2,3] ✓
7,15	-111	[2,3]
11,15	1-11	[1,2]
13,15	11-1	[1,2,3] ✓
14,15	111-	[1,2,3] ✓

3,7,11,15	--11 [2]
12,13,14,15	11-- [1,2,3]

	11	12	13	14	15	3	7	11	12	13	14	15	3	7	12	13	14	15
3,7 [2,3]	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	0
7,15[2,3]	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1
11,15 [1,2]	1	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0
3,7,11,15 [2]	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0
12,13,14,15[1,2,3]	0	1	1	1	1	0	0	0	1	1	1	1	0	0	1	1	1	1
		F_1						F_2							F_3			

(11,15)[1,2]
(12,13,14,15)[1,2,3]
(3,7)[2,3]

Figure 5.8

1	0001 [1,2]
8	1000 [1,2]
3	0011 [1,2]
5	0101 [1,2]
9	1001 [1,2]
10	1010 [1,2]
7	0111 [1,2]
11	1011 [1,2]
14	1110 [1]
15	1111 [1,2]

1,3	00-1 [1,2]
1,5	0-01 [1,2]
1,9	-001 [1,2]
8,9	100- [1,2]
8,10	10-0 [1,2]
3,7	0-11 [1,2]
3,11	-011 [1,2]
5,7	01-1 [1,2]
9,11	10-1 [1,2]
10,11	101- [1,2]
10,14	1-10 [1]
7,15	-111 [1,2]
11,15	1-11 [1,2]
14,15	111- [1]

1,3,5,7	0--1 [1,2]
1,3,9,11	-0-1 [1,2]
8,9,10,11	10-- [1,2]
3,7,11,15	--11 [1,2]
10,11,14,15	1-1- [1]

	1	3	5	7	8	10	14	15
1,3,5,7	1	1	1	1	0	0	0	0
1,3,9,11	1	1	0	0	0	0	0	0
8,9,10,11	0	0	0	0	1	1	0	0
3,7,11,15	0	1	0	1	0	0	0	1
10,11,14,15	0	0	0	0	0	1	1	1

F_1

	1	3	5	8	10
1,3,5,7	1	1	1	0	0
1,3,9,11	1	1	0	0	0
8,9,10,11	0	0	0	1	1
3,7,11,15	0	1	0	0	0
10,11,14,15	0	0	0	0	1

F_2

(1,3,5,7)[1,2]; (8,9,10,11)[1,2]; (10,11,14,15)[1]

Figure 5.9

Mi	1	3	5	7	8	10	14	15	
#F	2	2	2	2	2	2	1	1	
#RAD's	6	6	4	3	4	5	2	3	
local RAD's	3 3	3 3	2 2	3 -	2 2	3 2	2 -	3 -	F_1 F_2

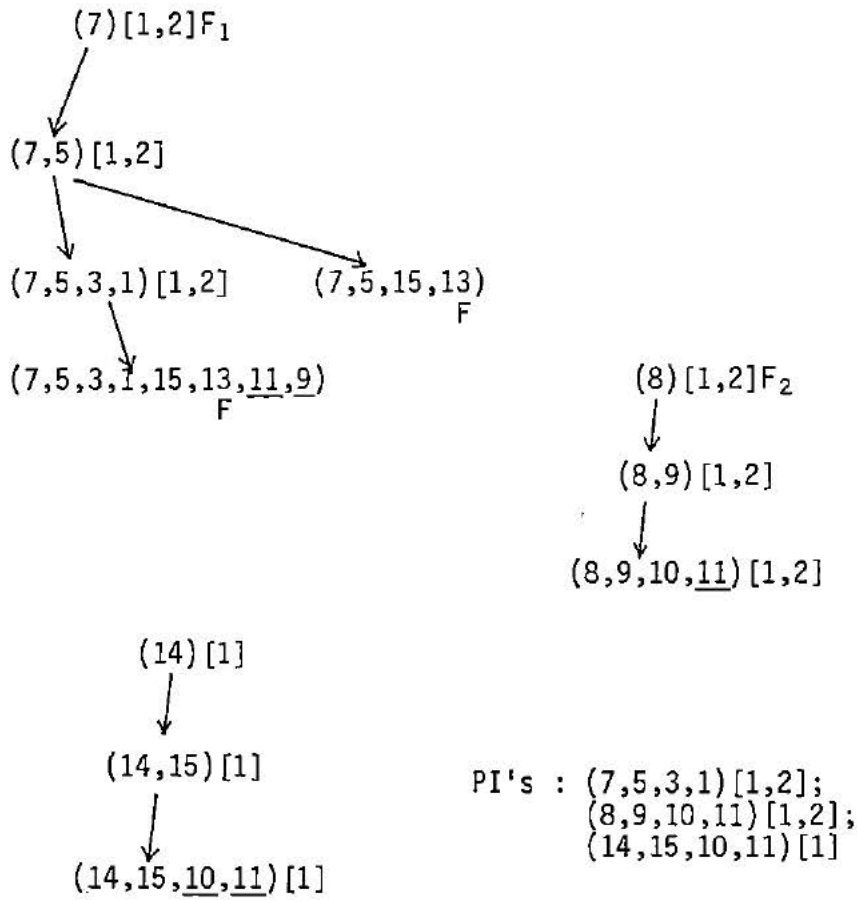


Figure 5.10

Chapter VI

THE IMPLEMENTATION OF DSA2.

6.1 BACKGROUND.

This chapter will detail the overall implementation of DSA2 as it was done in APL as a first feasibility study. Each routine is presented, together with its links to the rest of the program, and the global and local variables needed.

The particular choices of language and data structures are not at all binding for DSA2, but were a matter of personal preference in programming. The main question was whether it was possible to program efficiently the search including the new heuristics, without an unreasonable amount of complexity. This implementation does not particularly concentrate however, on all aspects of efficiency (storage or execution), as the feasibility of the algorithm itself remained the focus.

The algorithm developed for the manual method is changed only slightly in the implemented version; the basic logic remains the same, but each step has to be broken down into smaller substeps in order to make each action more explicit.

The major parts can be categorized as follows:

1. Receive inputs for the network.
2. Provide global ranking for TF's to be covered.
3. Expand tree to form a set of hypotheses.
4. Reduce the set of hypotheses to one or more pseudo-essential PI's using subtrees expansions when needed.
5. Update global ranking tables and check for completion.

There are three major routines involved in the system:

i) RAD

ii) EXPANDTREE

iii) MAKECHOICE

A short introduction to them seems necessary and later on a fully detailed description of every step is presented.

(A) RAD routine.

This routine covers the actions needed for parts (1) & (2), and controls the main loop to determine when the whole network has been satisfactorily covered by the chosen PI's, part (5).

(B) EXPANDTREE routine.

Covering part (3), it takes an initial TF, builds its list of adjacencies and expands along it to form a PI. When any difficulty is encountered (as an empty PI or a function

qualifier flag smaller than the original one), it stores the current expansion, backtracks and tries to form other PI hypotheses along other branches of the possible tree. Some branches may be pruned in the process. At the end, a minimal set of PI hypotheses with their flags has been produced in an array.

(C) MAKECHOICE routine.

Given a set of PI hypotheses, one must make choices to determine whether there exists an essential PI cover, or to select the appropriate PI's as pseudo-essential. This routine may need to expand subtrees to be able to choose appropriately and in that case it calls both EXPANDTREE and itself recursively.

There are smaller functions in the system which perform auxiliary tasks and a short description of them follows, in alphabetical order.

-CHECKEMPTY

It checks if any PI hypothesis (a row in array HYPO1) is made up only by XF's (don't cares). If so, it deletes that hypothesis.

-CHOSEN

Once a PI has been chosen for the network cover, it updates all global ranking tables and lists.

-FLAG

Given a TF, it returns its function qualifier flag.

-LISTRAD

Given a TF and a function number, it lists all adjacencies for that TF along that function.

-LOCADJ

Calculates the number of local adjacencies of a TF for a given function.

-RANK1

Performs the main ranking for TF's. Given a subset of functions of the network, it returns a list of TF's still uncovered in it, with the number of adjacencies for each of them and the first TF which should be expanded.

-RANK2

Given a TF it chooses along which function of its flag it should be expanded (i.e. which adjacencies to use for it).

-SPECIALCASE

It expands trees for all those TF's whose total adjacencies are less or equal to the total number of functions in their flag. They are considered special cases which should be dealt with at the beginning since they usually provide more direct paths to essential covers.

-TOTADJ

Calculates the number of total adjacencies for a given TF for the whole function qualifier.

-TOTFLAG

Given a PI as a list of TF's, it finds its function qualifier flag as intersection of the flags of each TF.

6.2 PART 1 : THE RAD ROUTINE.

The two major inputs to this routine are NUMF=number of functions in the network, and NUMV=number of variables in functions. Moreover, a prompt is given to enter the description of each function as characters, where every minterm is qualified as either 0, 1, or X . At this point a TABLE is constructed as a 3-dimensional array where each row defines a function, each column a minterm and the two planes are used for the proper representation of each entry as: TF=11; XF=01; FF=00; in bits. This particular representation minimizes storage requirements.

Next, two vectors are formed, COUNT and NONZ as:
COUNT= for each minterm it holds the total number of function appearances (=0 if FF or XF).
NONZ= each TF in row 1 has in row 2 its total number of adjacencies.

The main loop can now start. The subset of the network one considers corresponds to the maximum number of function appearances, so that common terms can be identified first. A list is made of those TF's for that subset and a starting TF is chosen by calling RANK1 and RANK2.

The function EXPANDTREE is now called for that TF and on return one expects that TF and usually others to have changed their status to XF's. Since the CHOSEN function has already done all necessary updating, one must now proceed to the next TF for this network subset, or, if none left, to the next smaller subset of the network, until no TF is left uncovered.

6.3 PART 2 : EXPANDTREE ROUTINE.

There are two inputs : the starting TF for the tree and the particular function of its flag along which the adjacencies should be used. The main aim is to fill up an array called HYPO1 with possible PI's as rows. A PI is a list of

integers (the size of the list always being a power of two) representing the minterms covered. Note that the first element of every PI is the original TF.

First a bit of housekeeping prepares a list of possible adjacencies and, if there are none, obviously the given TF is chosen immediately as its own cover. If this happens to be a recursive call, (i.e. this is a subtree development) then care must be taken to take away from the list of adjacencies the last one used in the calling tree, since it would create a cycle if expanded again.

The main thrust at this point is to proceed depth-first in the expansion, halting only when a problem occurs. Hence each adjacency is in turn added to the current node (called TREE) doubling its size. Along the node there is its current flag which is recalculated after every expansion. As long as this flag matches the original flag for the given TF, the depth-first search can continue. There are three possible cases and each is examined separately :

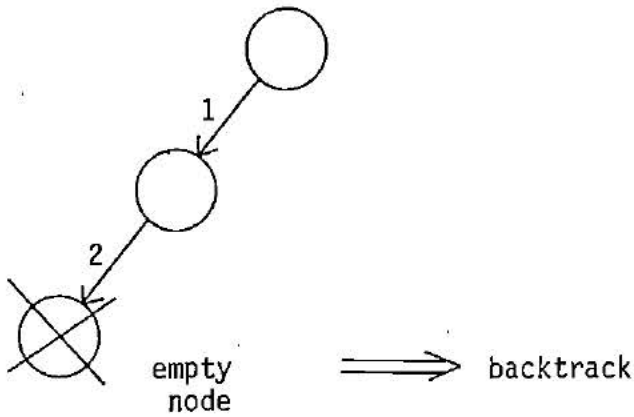
a) The flag is all right and no RADs are left to be expanded : if no other hypothesis has been previously formed, then this node can be selected as an "essential PI" with no further work. If however other nodes had been expanded in this tree (they would currently

be stored in HYPO1), then this final hypothesis must be stored as well and the choice among them left to the MAKECHOICE routine.

b) The flag is empty, i.e. there is no PI along this RAD. One must backtrack to the previous PI (half of this current one) which was all right, and choose another adjacency to expand along. The RAD just used (which formed the empty PI) must however be stored in a backtracking index vector, since it will have to to be used again at a higher level to provide an alternate path down the tree. (see figure 6.1) If no other adjacency is left, then backtracking to one further higher level is necessary, using now the adjacencies pointed to by the backtracking index, and the current node must be stored. (see figure 6.2)

c) The third case arises when a PI is valid but only for a subset of the original network considered. One does not want to choose this hypothesis, but it must be stored since it might have to be used for comparisons and/or subtrees. The situation will be explained more clearly in the MAKECHOICE routine, but the underlying idea is basically that one does not want to make any choices of cover while still expanding the trees (besides the trivial ones). Hence almost all expanded

RAD's list : 1 2 3



backtrack index = 2

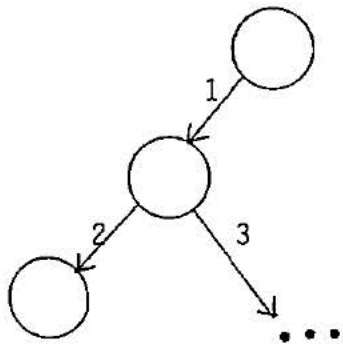


Figure 6.1

nodes should be stored as hypotheses and then compared among themselves.[3] At this point backtracking to the previous level is necessary and a new RAD picked out of the list. If none are left, further backtracking and use of the backtracking index is done as in case (b).

One part of this whole process which has been left unclear is the "storing" of the nodes-hypotheses. The problem had to be overcome (at least in APL) of how to store together nodes of different sizes, since there is no facility for ragged arrays. Hence some heuristics had to be developed not only to overcome this problem, but help speed up the "choice" process later on.

Three storage cases have been identified corresponding to the three stopping cases above:

a) store a node with a "good" flag.

b) store a node after an empty PI.

c) store a node with a "bad" flag.

---

[3] This may sound similar to other algorithms (Quine-McCluskey for example) where all possible PI's are formed and then a covering problem arises separately. However here one does not produce *all* possible PI's for the network : the essential ones are produced directly, while the pseudo-essential ones are selected from a limited choice of hypotheses, and hence the covering problem, if not totally eliminated, is greatly reduced.

RAD's list : 1 2 3

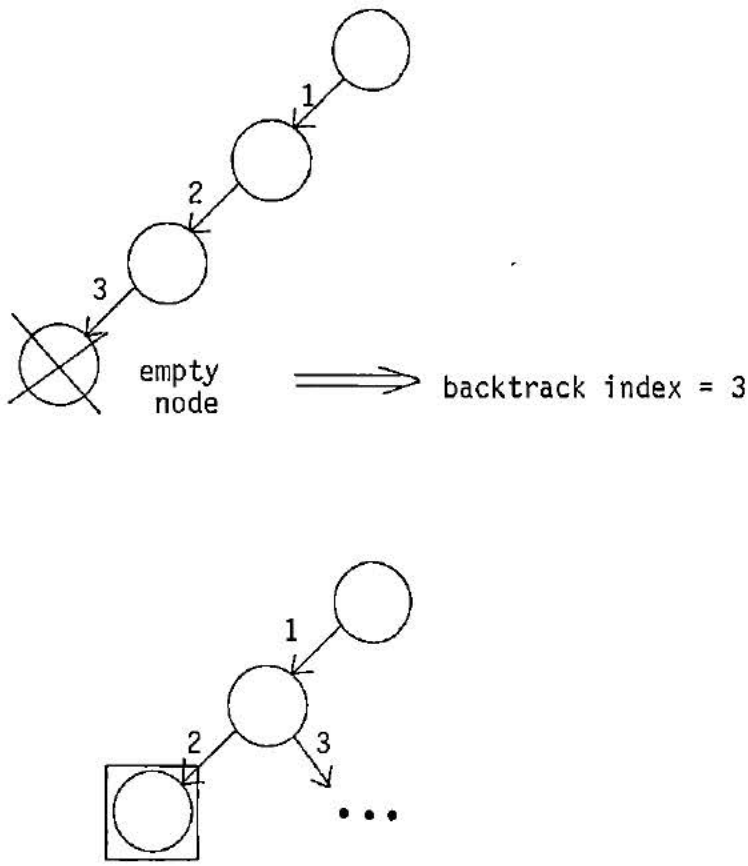


Figure 6.2

Each of the three cases has been broken down in three sub-cases according to whether :

i) node to be stored is of the same size as previous nodes-hypotheses.

ii) node to be stored is bigger than previous nodes-hypotheses.

iii) node to be stored is smaller than previous nodes-hypotheses.

Case (a) : store a node with a "good" flag.

If this is the first hypothesis, then there is no problem.
(i) If the current node is of the same size as previously hypothesized nodes, just add it to the array.
(ii) If the node is bigger, then it has gone further down into the tree and since its flag matches the original desirable flag for the tree, this is probably the best hypothesis so far. Hence this node only will be kept in the array for further comparisons. It would be nice if one could discard all the previous nodes : as it turns out, most of them will be subsets of this current one, but others might have their own covering value. This principle comes directly from the DSA2 algorithm, where, even after a pseudo-essential PI has been selected, the other nodes left on the tree, if they have a good flag, must be looked at and possibly chosen

themselves as pseudo-essential if they contain some TF and not just XF's by that time. Thus the previous hypotheses are stored in a secondary location, called HYPO2, and will be checked by MAKECHOICE after all other covers have been selected.

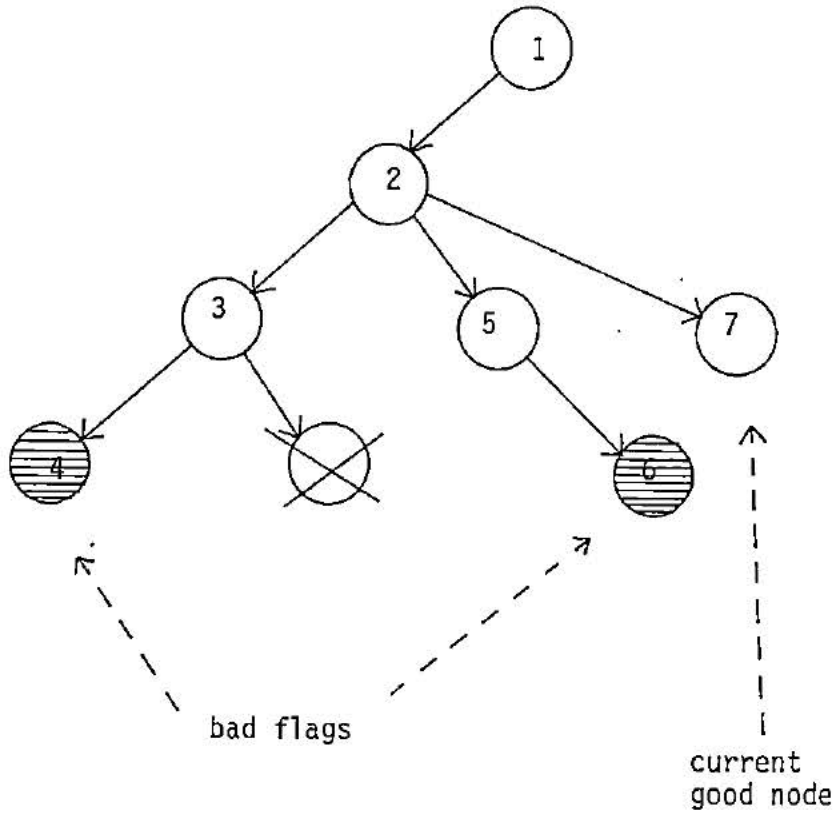
(iii) If the node is smaller ,then the situation has to be further examined. If all the nodes in HYPO1 have "bad" flags, then the current hypothesis is desirable. The previous hypotheses are themselves useless, but the preceding nodes which formed them must have had a good flag (or one would have stopped expanding) and they must be compared to the current one. Hence HYPO1 is halved and the new node added to it. (see figure 6.3)

On the other hand,

if any of the nodes stored so far have a good flag, they are valid hypotheses, possibly even more valid then the current one which may turn out to be a subset of them. Hence all the good nodes already there are retained, and the current one stored in HYPO2 where it will be revisited at the end.

Case (b) : store a node after an empty PI.

This case is very simple since, as explained earlier, one must halve the current node to reduce it to its predecessor which must have been a proper node with a good flag, and thus one falls back into Case (a).



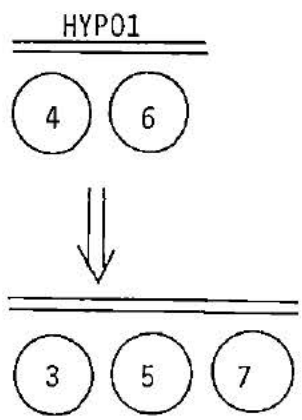


Figure 6.3

Case (c) : store a node with a "bad" flag.

If this is the first hypothesis, there is no problem.
(i) If the current node is of the same size as previously hypothesized nodes, just add it to the array.
(ii) if the node is bigger, one must check if any of the previous hypotheses have a good flag. If so, then the current node must be halved (i.e. reduced to a good node) and then stored using Case (a). If no hypotheses so far have a good flag, then surely the current one which goes deeper in the tree is altogether preferable and should be substituted for whatever is in HYPO1.
(iii) If the node is smaller, we can discard it immediately since all the previously found hypotheses will in any case be preferable.

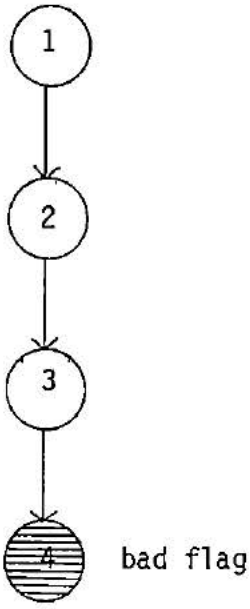
After all this, one exits this routine with basically four arrays :
HYPO1 - it contains all nodes-hypotheses.
FLAG1 - it contains for each row the flag for the node in the corresponding row of HYPO1.
HYPO2 & FLAG2 - either empty or containing secondary, and last to be checked, hypotheses.

These four arrays are passed globally when MAKECHOICE is called.

6.4 PART 3 : MAKECHOICE ROUTINE.

Finally at this stage the algorithm comes to a conclusion following the rules for choosing pseudo-essential PI's.

The first testing that must be done is whether there is a case of vertical comparison. To have more details of how and why this situation is selected by itself, one must refer to earlier explanations of the DSA2 algorithm, however the situation is easily illustrated. (see figure 6.4). In a vertical comparison HYPO1 will contain only one element and its flag will be different from the original starting flag. The choice of PI thus is between this "bad" node and its predecessor in the tree (i.e. its half list, whence the name of "vertical" comparison). In order to make this choice one must use subtrees, reexpanding one or more TF's in these lists to recursively point to the proper selection. In general one would choose a TF to expand upon from the "above" node with a proper flag, unless the tree is only one level deep (as it would imply reexpanding the original TF in an infinite loop).(see figure 6.5).


1
2
3
4
bad flag

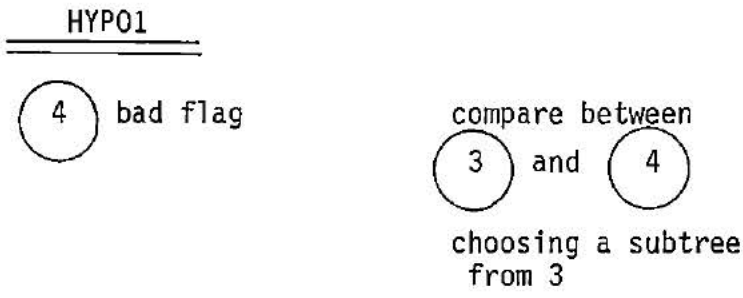
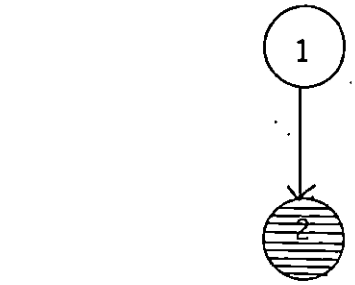

HYPO1
4
bad flag
compare between
3
and
4
choosing a subtree
from 3

Figure 6.4

Another easy choice comes about if only one element is in HYPO1 and its flag is equal to the starting flag : that PI can be chosen as "pseudo-essential", if this is not a recursive subtree call. In a subtree one can select this PI also but only if it contains at least one TF, and not just XF's. (see figure 6.6)

After these preliminaries, one is in the situation of having an array HYPO1 with more than one element. First of all some of these PI's may be comprised only of XF's and can be eliminated. Obviously if all of them contain only XF's, one has equivalent choices, hopefully among which at least one PI with a good flag. If not, the only cover for the original TF is found to be the minterm itself.

Otherwise, HYPO1 has more than one member and it is important to discern how many of these PI's have a good flag.
- If none of them has a good flag, subtrees are required. However, the subtrees are done on the parents of these nodes, if possible, in order to choose among hypotheses with a good flag.
- If exactly one PI has a good flag, then choose it as pseudo-essential if it contains any TF's, or delete it and use subtrees on the other PI's to make selections.



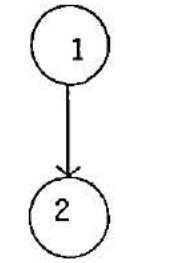
HYPO1

2 bad flag

compare between
1 and 2
choosing a subtree
from 2

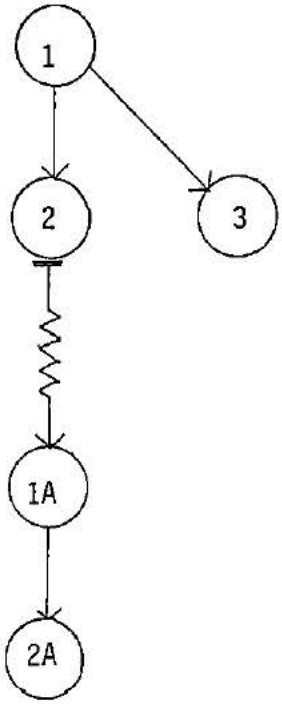
Figure 6.5

- If more than one PI has a good flag, then more heuristics are needed. The first criterion is to count the number of TF's in each node-hypothesis. One can select as first pseudo-essential the node which proposes a cover for the largest number of TF's. After that choice is made, most of the other PI's may become covered as well; if not the whole process is repeated on them. It can happen that all nodes-hypotheses can cover the same number of new TF's, in which case no immediate choice is possible and subtrees must be used to aid the selection.



HYPO1

choose it



HYPO1

choose it IF it
contains some TF's

Figure 6.6

6.4.1 Subtrees.

The use of subtrees has been often mentioned and certainly more explanation is needed. The idea is to choose a TF, other than the original one for the original tree, and expand a separate tree for it. By choosing a PI cover for this alternate TF one eliminates some of the original hypotheses among which direct choice was not possible. Both EXPANDTREE and MAKECHOICE are called recursively and they proceed as described, stopping the recursion in these two cases :

a) the subtree generates alternative PI's with only XF's: then the subtree is discarded and a choice can fall upon the PI in the original tree from which the TF to start this subtree was selected.

b) the subtree generates proper choices (essential PI's or pseudo-essential easily selected) and returns after having changed the global environment.

Hence, in (a), the return from recursion has left the environment intact, but a choice can now be made. In (b), choices have been made, the environment changed and all PI's in HYPO1 must be checked to see if they have become redundant (i.e. containing only XF's).

This may or may not conclude all choices, since it could be that more subtrees are needed. Moreover it must be noted that the subtrees themselves may often invoke the use of subtree expansions.

6.5 CONCLUSIONS.

While the first two parts of the algorithm, implemented by the RAD and EXPANDTREE routines, are relatively straightforward, however complex in subcases, the last part is less easily explainable, basically because of its recursive nature.

The only other point which remains uncovered in this description of the implementation of DSA2 is the storage requirements of the system. The global TABLE has dimensions proportional to the number of functions in the network (# of rows) and to the number of variables in the functions (# of columns = 2**number of variables), with two fixed planes. The whole thing is binary however and stored as bits.

The two other major structures which remain constant in the system are the vectors COUNT and NONZ. COUNT will have dimensions = 2**number of variables, that is the number of minterms, and since it contains integers it will take 4x2**(number of variables) bytes. NONZ has two rows and as

many columns as the number of true minterms overall the functions, so its dimensions depend entirely on the complexity of the functions. Moreover the size decreases as the search continues. Should problems arise for lack of storage when dealing with larger systems, the latter two vectors could be changed to a more compact representation.

Chapter VII

ANALYSIS AND DISCUSSION.

We have so far illustrated the Directed Search Algorithm for single-output functions and its new extension for multiple-output functions. The analysis of the method, of any method in general, cannot be adequately done as an isolated system, but needs to be based on its relative value both to the problem under consideration and to other applicable means of solving it.

A computational complexity argument is not entirely appropriate for DSA. The trees used are dynamic structures whose rate and width of expansion are directly dependent on the local adjacencies of the starting true minterms. While it is certainly true that the overall amount of computation involved is exponential in the number of input variables, as it is the case for other methods, some local factors which actually determine the feasibility of an implementation are not easily specified analytically.

We must remember that circuit synthesis is after all a very pragmatic problem and what makes a particular method

more acceptable than another one is not just the theoretical expectations, but its final performance after the housekeeping has been incorporated in the model. It has been acceptable in fact to compare algorithms as to the actual time scale involved in solving problems with specific numbers of inputs. If a large value of n, say in the range 19 to 22, can be used at all, then the method is considered adequate.

M. H. McKinney[MCK1] presents exactly such a milliseconds-based argument in advancing the claim that DSA is more effective than other earlier methods. We prefer to try adopting a more high-level approach: this may lose us some precision, but we believe that an analysis should not be entirely based on an implementation-dependent view.

Some advantages of DSA are common to both the single and the multiple-output cases. First of all the use of trees. Since tree-searches have been employed in large AI problems (from expert systems to the internals of PROLOG), their great value has already been highlighted.

Consider the development of a particular minterm. The depth-first expansion along its RAD's works in a straightforward manner towards its largest possible PI. It is clearly a goal-oriented process, where each middle node in the path is simply a stepping stone which does not even re-

quire extra storage or extra evaluation. Only when a contradiction of any kind arises is backtracking necessary and a second look at a previous result might be in order.

Consider now the same development with a more classical method like Quine-McCluskey. For the particular minterm, an iteration is necessary through almost all other minterms, true ones and don't care ones, which are present in the function. Notice that when the lists are initialized and the terms divided, the provision is made that every term gets compared to only those other terms which have a single extra one bit, i.e. in the next section below. This insures that a minterm with, say, two bits equal to 1 (for example 0011) is not unnecessarily compared to another minterm with four bits equal to 1 (say 1111), where obviously there can be no direct combining of the two as yet.

However minterms are still compared to others where the distance between them is greater than one, so that the concept of adjacency is not exploited. For example it is feasible to compare 0101 to 1110 from two sections, yet the distance definition would tell us that they are not adjacent and their comparison cannot lead to a starting implicant.

Clearly the point is made: RAD's are a restriction of the list-sections and they "direct" the search in the more fea-

sible likely area of solution. Moreover the tree is not just the basis for the search. When its development is finally halted, with or without backtracking, the leaf nodes contain the answers as prime implicant(s) of the minterm considered. In some cases the essential cover is thus immediately found, while in others the search may have to continue using subtrees. However this is not a flaw of the algorithm: it is a problem of the minterm involved and exists in all methods. If that minterm does not possess an essential cover in that function, more work is always required in order to choose the proper prime implicant for it.

However, if an essential PI exists, DSA will find it in a one-branch depth-first tree-search, while Quine-McCluskey, after the final lists have been iteratively constructed, does not yet know anything about a choice. Even when this particular process becomes lengthy, as happens inevitably in certain functions, at all times only a restricted number of PI's and minterms are under consideration in DSA. Specifically the choices are kept as local as possible to what we might consider as the relevant area of the corresponding map, without having to include continuously the impact of the distribution of minterms in the other parts of the function, which are at that moment irrelevant. Yet this is exactly what a covering table approach does, since all rows

and all columns for all terms are always presented and no local information is highlighted in any way.

We can continue the list of advantages of DSA by noticing how the don't care terms are handled. Don't cares can be very helpful in the final solution, however for most circuit problems, whether in testing or in design, they can add complexity to the development of a method.

In Quine-McCluskey they are assumed to be true forms during the lists iteration so that some possible good connections with the true minterms are found. However many implicants and prime implicants using only don't cares, and thus useless, have to be worked through, stored, and used in comparisons.

DSA knows about don't cares and uses a RAD relative to such a term. But it never considers a don't care as a starting point and neither does it try to develop a possible cover for it. The idea is to truly have a don't care as an unspecified term, making no assumptions as to whether we should assign it to be a 1 or a 0 to start with, and to avoid any extra work on its behalf.

Another very important aspect in DSA is the constant exploiting of information. While it is true that each tree

expansion for a minterm uses the local RAD's, every choice of prime implicants at any point affects the knowledge for the development of the next tree-search. This is found in Quine-McCluskey only in the covering table part, when by choosing, according to some criteria, a particular row (a PI), some columns are also eliminated. The next choice is thus relative to the new updated table. However this time dependency does not apply any constraint or provide any information earlier in the method, when instead all minterms are developed in parallel and quite independently.

Each tree, or subtree, in DSA usually makes a choice. This prime implicant selected contains, most of the time, some true minterms in addition to the starting one. Since these other true forms now have a cover, they change their status to don't cares in a global knowledge table. This implies that the next tree-search starts with a different set of information from the previous one, specifically with the knowledge of some cover already existing in some area. Thus the starting minterm will not be from the covered area, and some potential trees are eliminated. Also the tree expansion itself may contain more pruning and less backtracking since some adjacencies may now be towards a newly set don't care term and provide a less probable pseudo-essential cover.

The only slight equivalent in Quine-McCluskey is found in the housekeeping required in not inserting an implicant in a new list if it had already been formed in some previous iteration. However the work of constructing the prime implicant remains, plus a bit of duplication checking labour. This point should be duly stressed since this global updating and usage of information becomes rather crucial in the multiple-output case, where otherwise the number of possibilities for prime implicants by developing them in parallel can be overwhelming.

The first part of the DSA algorithm is based on a concept called ranking. This is simply the idea of using knowledge about the minterms to extract a priority scale for the evaluation of their minimal cover. The idea itself is not new. In a Karnaugh map approach we do not circle indiscriminately all possibilities, but we methodically search for the essential ones first. Then the remaining ones are looked at. Applying even such a relatively simple topological algorithm without discrimination can lead to invalid solutions.

In DSA one must start with the minterms having a lower number of adjacencies, since they are the ones with more probability of leading to prime implicants whose essentiality is easily determined. Without this, the best one can

hope for is a lot more work and maybe an equivalent solution.

While the original presentation of the DSA for single functions mentioned this aspect, its author did not fully grasp its importance, since it is only given as a desirable heuristic rule and it is not even present in the implementation of the method. In fact: [MCK1, p.104]

> The minterms are placed in the new TF list in the order in which they are specified by the user; thus the first minterm expanded is the first minterm specified by the user. Example 11 shows the result of a situation where a non-essential TF is specified early in the specification of TF's. The resulting cyclic chain develops a large number of PCPI cells with a large number of non-essential recursive TF's..... The preceding discussion indicates that if the user knows something of the essentiality of some of the TF's, those essential TF's should be specified first.

This clearly shows how the original algorithm itself has been re-evaluated and improved and its results, both as final answers and development, have been rendered more consistent over different examples.

The concept of ranking has been widely expanded for the multiple-output synthesis where, without proper priority choices, a minimal solution cannot usually be achieved. It may appear at first that we have introduced quite an overhead in the calculations for the proper ranking of minterms. However it is necessary and it is only done m times, where m

is the number of functions realized by the network. Moreover only true minterms have their local and total RAD's counted, while, for example, in Quine-McCluskey even don't care terms have to be properly distributed in the list sections.

Hence we see the process of ranking as a positive feature of DSA since it is capable of extracting information of the topological layout of functions in a network and use the results as a basis for both deterministic and heuristic rules towards a minimal solution.

It has been said earlier that the complexity of minimization algorithms is exponential in the number of input terms. It is considered quite an achievement to be able to minimize functions of up to 16-20 variables in less than "geological" time. One serious problem which arises in most multiple-output synthesis methods is that the exponential factor is multiplied by the number of functions being considered. This comes from the obvious fact that, besides the lists being much longer, we must have a covering table for each function, concatenate them in parallel, and somehow reduce them simultaneously.

The same is not true in DSA. When the number of functions increases, one bit is added to the function qualifier

and the ranking process iterates one extra time. However the tree development remains the same, and the searches still depend on the topological layout of minterms without having to add or duplicate information. This means that large networks of many functions can be handled for a relatively large number of inputs.

For example, one might be able to use Quine-McCluskey up to 16-18 inputs for single functions, yet have to stop at 12-15 inputs for more than 3-4 functions in the network. In DSA we can still approach functions with 16-18 inputs, depending on the implementation, while the number of outputs could be larger.

As a last discussion point we would like to mention a few points of comparison with MINI[HON1]. MINI is the name given to a minimization algorithm developed by S.J.Hong, R.G.Cain and D.L.Ostapko, around 1974, and it is one of the better known and quoted new methods beyond the classical ones. It seems to be able to process functions with large numbers of inputs, to be easily extended to multiple-outputs and even to multi-valued logic.

The method used in MINI will not be described here in detail, only a few useful items mentioned. The most notable common element with DSA is that the cost function is the

same, that is an equal weight is assigned to every implicant and thus their shape considered unimportant. The basic idea of MINI is to start with an initial solution as a two-level sum-of-products expression and iteratively improve it. In order to do this many passes through the process are necessary, each of which uses three essential subprocesses of reducing, reshaping and enlarging the implicants. Every subprocess is based on the "sharp" operation and uses the AND, OR operators on cubes notation. The sharp operation first presented by Roth is a rather complex non-symmetric evaluation based on the topological meaning associated with a term. While it is indeed a powerful concept, its usage has not been widespread mainly due to its non-intuitive complexity which makes it hard to apply it easily in problems from various domains.

The internal representation of the cubes is decoded in such a way as to minimize the work of these operations. However they still stand in contrast to DSA where only addition is used on the elements of a node in expanding the trees, and, in the worst case, it is applied to a particular node only for as many RAD's as there are with full backtracking.

It is interesting to note that the number of iterations of MINI is proportional to the final number of prime implicants. In fact the authors state that: [HON1, p.444]

> the algorithm is designed for "shallow functions", those functions whose solution contains at most a few hundred implicants regardless of the number of variables.

The method in fact fails for an example exclusive-OR function of sixteen inputs where there exist 2^{15} irreducible prime implicants, since each iteration tries to do a lot of work with no success. DSA would recognize from the zero RAD count of each term that no tree is even necessary, and the only work involved would be to list as output each minterm.

One other point of interest raised by comparisons with MINI is their stated philosophy of starting from any given initial cover as opposed to a list of minterms. They claim that this approach poses fewer limitations on the number of variables that can be handled. While it might be correct for MINI which only strives to improve a solution, we do not believe that the input stream of minterm bits is a main disadvantage. Most functions come with high level specifications and a preprocess interface with a user must always exist to transform those specifications into a valid input form. This transformation can be easily seen as part of a user's interface and the work involved in translating from

sum-of-products expression to cube notation and to minterms is not necessarily the fundamental part of a minimization algorithm.

The last topic we would like to discuss briefly is the issue of minimality. MINI is designed for a near-minimal solution and achieves, at times, a minimal one. In the majority of cases, the determination of minimality can be based solely on previous known results. DSA has so far reached a minimal solution, and the only theoretical problem we see arises in the case of cycles. The method does recognize cycles, yet it has no inherent logic to solve them. Hence a minimal solution in these cases is left as a separate problem from DSA itself.

APL has been used to implement both MINI and DSA. At the moment MINI is being recoded in machine language in order to provide an efficient production program. We believe that DSA as well would profit from recoding into another language with higher efficiency, even if not necessarily machine language since the method is not computationally demanding. We would like to point out as a last advantage to DSA, that the algorithm is not very complex at all and the programming did not require an unreasonable amount of work. In contrast the algorithm description of MINI [HON1, pp.452-454] appears quite awesome.

7.1 SUMMARY.

We have presented here both the single and multiple-output problems for a minimized two-level design of functions. The Directed Search Algorithm has been used as the technique to approach these problems, as opposed to other classical methods. The main advantage is to reduce the processes of generating all prime implicants and selecting a suitable cover to only one process of search which immediately provides an answer.

This has been particularly important for the multiple-output case. The algorithm uses tree searches and adjacency relations as the main tools. Heuristics were introduced to permit the pruning of trees, which produce a considerable decrease of the total search space. The advantages of the new algorithm have been explored: most of all the total computation is quite small, thus permitting its application to large sets of functions.

The method is well-suited for computer realization and it has been implemented and tested. This has been mainly a development and feasibility study: the application of DSA to other domains remains to be researched, as do its optimization and its testing for very large functions.

BIBLIOGRAPHY

[BOW1] R. M. Bowman, E. S. McVey, "A method for the Fast Approximate Solution of Large Prime Implicant Charts," *IEEE Transactions on Electronic Computers*, vol. C-19, Feb. 1970.

[DIE1] D. L. Dietmeyer, *Logical Design of Digital Systems*, Allyn and Bacon, Boston, 1971.

[FRI1] R. Fridshal, "The Quine Algorithm," Summaries of Talks at the Summer Institute of Symbolic Logic, Cornell University, 1957.

[HON1] S. J. Hong, R. G. Cain, D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM Journal of Research and Development*, vol. 18, Sept. 1974.

[KAN1] S. Khan, "Synthesis and Optimization of Programmable Logic Arrays", Technical Report No. 216, Department of Electrical Engineering and Computer Science, Stanford University, Stanford, Ca, July 1981.

[KAR1] G. Karnaugh, "A Map Method for Synthesis of Combinational Logic Circuits," *AIEE Transactions Communications and Electronics*, pt.1, vol.72, Nov. 1953.

[MCC1] E. J. McCluskey, "Minimization of Boolean Functions," *Bell Systems Technical Journal*, vol. 35, Apr. 1957.

[MCC2] E. J. McCluskey, *Introduction to the Theory of Switching Circuits*, McGraw-Hill, New York, 1965.

[MCK1] Melvin H. McKinney, "A Directed Search Algorithm for the Canonical Minimization of Switching Functions, *PH. D. Thesis*, Texas A & M University, Computer Science, 1974.

[MUR1] S. Muroga, *Threshold Logic and its Applications*, Wiley-Interscience, a Division of John Wiley & Sons, Inc. Toronto, 1971.

[NIL1] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

[PET1] S. R. Petrick, "A Direct Determination of the Irredundant Forms of a Boolean Function from the Set of Prime Implicants," Technical Report No. 56-110, Bedford, Mass.: AF Cambridge Research Center, April 1956.

[QUI1] W. V. Quine, "A Way to Simplify Truth Functions," *American Mathematical Monthly*, Vol. 62, Nov. 1955.

[REU1] B. Reusch, "Generation of Prime Implicants from Subfunctions and a Unifying Approach to the Covering problem," *IEEE Transactions on Computers*, Vol. C-24, Sept. 1975.

[RHY1] Thomas Rhyne, Philip S. Noe, Melvin H. McKinney, Udo W. Pooch, "A New Technique for the Fast Minimization of Switching Functions, *IEEE Transactions on Computers*, Vol. C-26, No.8, August 1977.

[ROC1] C. H. Roth, *Fundamentals of Logic Design*, West Publishing Company, St. Paul, 1979.

[ROJ1] J. P. Roth, "Algebraic Topological Methods for the Synthesis of Switching Systems," *Transactions of the American Mathematical Society*, vol. 88, July 1958.

[SUR1] Sureshchander, "Minimization of Switching Functions-A Fast Technique," *IEEE Transactions on Computers*, Vol. C-24, July 1975.

[TIS1] P. Tison, "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions," *IEEE Transactions on Electronic Computers*, Vol. EC-16, Aug. 1967.

Appendix A

ghGG

GG▽▽□

```
      D5                          4 INPUTS; 4 OUTPUTS
0011100X011110XX

      D6
0100000X100010XX

      D7
1101100X000100XX

      D8
1100101X001101XX

      4 RAD 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X          WITH QUOTES
TF -> 1
A+L MINTERMS FOR EACH FUNCTIONS AT EACH PROMPT
□:
      D5
□:
      D6
□:
      D7
□:
      D8
COUNT
2 3 1 2 3 0 1 0 1 1 2 3 2 1 0 0
SPECIAL CASES : 1 8 9 13
PI IS (1) [0 1 1 1]


PI IS (8 12) [0 1 0 0]


PI IS (9 11) [1 0 0 0]


PI IS (13 15) [0 0 0 1]

TFLIST
 4
 4
PI IS (4 0) [0 0 1 1]
```

```
TFLIST
 3 10 11
 6  5  8
PI IS (3 7 11 15) [1 0 1 0]


PI IS (3 7 11 15) [1 0 1 0]

TFLIST
 2 4 6 12
 2 4 3  4
PI IS (2 3 10 11) [1 0 0 0]


PI IS (6 7 14 15) [0 0 0 1]


PI IS (4 12) [1 0 0 0]


TIME USED
935
```

4 *INPUTS*; 2 *OUTPUTS*

```
      EX10
1011101100110000

      EX11
1000100011111100

      2 RAD 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X         WITH QUOTES
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
      EX10
□:
      EX11
COUNT
2 0 1 1 2 0 1 1 1 1 2 2 1 1 0 0
TFLIST
 0 4 10 11
 4 4  4  4
PI IS (0 4) [1 1]


PI IS (3 2 7 6) [1 0]


PI IS (10 11) [1 1]

TFLIST
 8 9 12 13
 4 3  3  2
PI IS (13 12 9 8) [0 1]


TIME USED
531
```

4 *INPUTS*; 3 *OUTPUTS*

```
      EX20
0001000111010001

      EX21
0001110100100011

      EX22
0100010100010001

      3 RAD 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X         WITH QUOTES
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
      EX20
□:
      EX21
□:
      EX22
COUNT
0 1 0 2 1 2 0 3 1 1 1 2 0 0 1 3
SPECIAL CASES : 1 4 8 10
PI IS (1 5) [0 0 1]


PI IS (4 5) [0 1 0]


PI IS (8 9) [1 0 0]


PI IS (10 14) [0 1 0]

TFLIST
 7 15
 7  6
PI IS (15 7) [1 1 1]

TFLIST
 3 11
 3  4
PI IS (3 7) [1 1 0]
```

PI IS (11 15) [1 0 1]

TIME USED
571

4 *INPUTS*; 3 *OUTPUTS*

*EX*30
0000000000011111

*EX*31
0001000100011101

*EX*32
0001000100001111

3 *RAD* 4
ENTER +INTERM LISTS AS FOLLOWS
FF -> 0
XF -> *X* *WITH QUOTES*
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
*EX*30
□:
*EX*31
□:
*EX*32
COUNT
0 0 0 2 0 0 0 2 0 0 0 2 3 3 2 3
TFLIST
12 13 15
5 6 9
PI IS (12 13) [1 1 1]

PI IS (15 14) [1 0 1]

PI IS (15 11) [1 1 0+

TFLIST
3 7
3 4
PI IS (3 7) [0 1 1]

TIME USED
577

4 *INPUTS*; 3 *OUTPUTS*

```
      EX40
0000000000011111

      EX41
0001000100011111

      EX42
0001000100001111

     3 RAD 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X        WITH QUOTES
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
      EX40
□:
      EX41
□:
      EX42
COUNT
0 0 0 2 0 0 0 2 0 0 0 2 3 3 3 3
TFLIST
 12 13 14 15
  6  6  6 10
PI IS (12 13 14 15) [1 1 1]

TFLIST
 3 7 11
 3 4  3
PI IS (3 7) [0 1 1]


PI IS (11 15) [1 1 0]

TIME USED
320
```

4 *INPUTS*; 4 *OUTPUTS*

*D*1
1111X1110000X000

*D*2
1111X0100111X000

*D*3
0000X1111100X111

*D*4
0000X1000010X000

4 *RAD* 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> *X* *WITH QUOTES*
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
*D*1
□:
*D*2
□:
*D*3
□:
*D*4
COUNT
2 2 2 2 0 3 3 2 1 2 2 1 0 1 1 1
SPECIAL CASES : 10
PI IS (10) [0 1 0 1]

TFLIST
5 6
7 8
PI IS (5 4) [1 0 1 1]

PI IS (6 4) [1 1 1 0]

PI IS (2 3 0 1) [1 1 0 0]

TFLIST
7 9
6 4

```
PI IS (9 11) [0 1 0 0]


PI IS (7 6 5 4) [1 0 1 0]

TFLIST
 8 9 13 14 15
 2 4  4  3  3
PI IS (8 9 12 13) [0 0 1 0]


PI IS (14 15 12 13 6 7 4 5) [0 0 1 0]

TIME USED
988
```

```
                                        4 INPUTS; 4 OUTPUTS

      DD1
0X1111000X1111X0

      DD2
1X0011001X0011X0

      DD3
0X1011001X0001X0

      DD4
1X1010100X0101X1

      4 RAD 4
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X         WITH QUOTES
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
      DD1
□:
      DD2
□:
      DD3
□:
      DD4
COUNT
2 0 3 1 4 3 1 0 2 0 1 2 2 4 0 1
TFLIST
 4 13
 8 10
PI IS (5 1 13 19) [1 1 1 0]

TFLIST
 4
 8
PI IS (4) [1 1 1 1]

TFLIST
 2
 4
PI IS (2) [1 0 1 1]
```

```
TFLIST
 0 8 11 12
 6 4  5  7
PI IS (8 9) [0 1 1 0]


PI IS (11 9) [1 0 0 1]


PI IS (15 13) [0 0 0 1]


PI IS (0 1) [0 1 0 1]


PI IS (12 13 4 5) [1 1 0 0]

TFLIST
 3 6 10
 3 3  3
PI IS (3 2 11 10) [1 0 0 0]


PI IS (6 4 2 0) [0 0 0 1]

TIME USED
1302
```

```
                                        5 INPUTS; 7 OUTPUTS

      BIG0
01010011010100110101001101010011

      BIG1
00100010001000100011001110101010

      BIG2
00010111000101110000011010011111

      BIG3
01010101010101010101010101010101

      BIG4
00100010001000100010001000100010

      BIG5
10101010101010101010101010101010

      BIG6
01000100010001000100010001000100

      7 RAD 5
ENTER MINTERM LISTS AS FOLLOWS
FF -> 0
XF -> X         WITH QUOTES
TF -> 1
ALL MINTERMS FOR EACH FUNCTION AT EACH PROMPT
□:
      BIG0
□:
      BIG1
□:
      BIG2
□:
      BIG3
□:
      BIG4
□:
      BIG5
□:
      BIG6
COUNT
1 3 3 3 1 3 5 3 1 3 3 3 1 3 5 3 1 3 3 3 1 3 5 3 3 3
      3 3 3 3 5 3
```

```
PI IS (6 14 22 30) [1 1 1 0 1 1 0]

TFLIST
 1 2 3 5 7 9 10 11 13 15 17 18 19 21 23 24 25 26
     27 28 29 31
10 10 10 10 12 10 10 11 10 13 10 11 10 9 10 7 10
     11 10 9 11 12
PI IS (24 28) [0 1 1 0 0 1 0]


PI IS (21 29 5 13) [0 0 1 1 0 0 1]


PI IS (1 9 17 25) [1 0 0 1 0 0 1]


PI IS (2 6 10 14 18 22 26 30) [0 1 0 0 1 1 0]


PI IS (3 7 11 15) [1 0 1 1 0 0 0]


PI IS (19 23) [1 1 0 1 0 0 0]


PI IS (27 31 11 15) [1 0 1 1 0 0 0]

TFLIST
 0 4 8 12 16 30
 4 4 4  4  4  4
PI IS (0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30)
      [0 0 0 0 0 1 0]

TIME USED
1519
```

VITA

Surname: SERRA Given Names: MICAELA

Place of Birth: MILANO,ITALY Date of Birth: Nov. 29, 1953

Educational Institutions Attended, with Dates of Entering and Leaving:

UNIVERSITY OF MILANO, MILANO, ITALY	1972 to 1973
UNIVERSITY OF TORONTO, TORONTO	1974 to 1979
UNIVERSITY OF MANITOBA, WINNIPEG	1980 to 1983
UNIVERSITY OF VICTORIA, VICTORIA	1983 to 1984

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions:

B. Sc. (Major)	1983	University of Manitoba, Winnipeg

Honors and Awards:

1967 Science and Engineering NSERC Scholarship, 1983/1985

Publications:

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make *single copy only* for such users or in response to a request from the library of any other university, or similar institutions, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis

DIRECTED SEARCH MINIMIZATION OF MULTIPLE OUTPUT NETWORKS

Autho [redacted]

MICAELA SERRA
Name

April 1984
Date