Efficient Implementation Of An Exact Multiple-Output Boolean Function Minimization Algorithm.

by

César A. Dueñas

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Morton Nadler, Chairman

James R. Armstrong

Festus G. Gray

September, 1989.

Blacksburg, Virginia

Efficient Implementation Of An Exact Multiple-Output Boolean Function Minimization Algorithm.

by

César A. Dueñas

Morton Nadler, Chairman

Electrical Engineering

(ABSTRACT)

The performance of the Svoboda-Nadler-Vora algorithm for exact multiple-output boolean function minimization is studied and compared with a heuristic minimization method.

For this purpose, the algorithm has been implemented in optimized ANSI C code. This implementation introduces a new set of procedures to reduce the cost of prime implicant generation. The concept of weight as the number of 1 and don't care neighbors of a state is used to take advantage of the special cases when a state has only one neighbor or no neighbors at all. The cost of prime implicant generation is further reduced by using the fact that the input dependency of any given state is limited by which of its neighbors exist within an output that are 1's or don't cares. A detailed example illustrates how the heuristic method can fail to find the absolute minimum of a boolean function.

Acknowledgements

I am deeply indebted to Dr. Morton Nadler for his guidance. His continued encouragement and technical expertise provided throughout the development of this research are specially appreciated.

Special thanks are also due to Dr. J. R. Armstrong and Dr. F. G. Gray, not only for kindly accepting to serve in my advisory committee, but for their outstanding dedication as professors of my graduate courses.

I am also grateful to my fellow students and colleages in the computer engineering program, especially to Mr. Roshan Fernando and Mr. George Selibas whose friendship and support have been invaluable in the completion of my Master's program.

Thanks are also due to my friends in Blacksburg, particularly Miss Monica Lluch and Mr. Juan Sabate and for all their support, understanding, and a very special friendship.

Finally, I would like to offer my sincerest gratitude to my dear family, for their love and inconditional support that have made possible the completion of my studies.

Table of Contents

Chapter 1

Introduction

Boolean logic minimization has always been an important tool in digital systems design. In the early days of digital design, logic gates were expensive. This made the implementation of a given logic function with a small number of devices a prime design objective. A number of algorithms and graphical techniques were developed for the manual simplification of boolean functions. Karnaugh maps and Marquand charts [10] are two good examples of graphical tools to assist the boolean logic minimization process.

In recent years, with the introduction of VLSI and programmable logic devices (PLD), logic minimization has regained interest. For example, the area used by a programmable logic array (PLA) is directly affected by the number of product terms of the function being implemented. Also, the application of logic minimi-

zation has diversified to other fields of digital design such as the generation of test vectors for fault detection in sequential circuits [17].

The minimization of a logic function typically involves two major steps: The generation of the set of prime implicants, and the selection of a minimum subset of primes that realizes the function. The process of generating all the prime implicants has a worst case complexity of $O(3^n/n)$. The second step is known to be an NP-complete problem. Therefore, as the complexity of the designs grow, the traditional exact minimization techniques have become impractical. This problem motivated the development of approximate methods.

The first methods of this type still generated all prime implicants, but then used heuristics to produce a near-minimum cover. More recently, new heuristic approaches such as MINI [1], PRESTO [2] and MIN370 [3] have been very successful in reducing the time of the minimization process while obtaining good approximations of the minimum form. These programs use a method of expansion and elimination of prime implicants, avoiding the costly generation of the set of all prime implicants.

The most recent of this class of PLA heuristic minimizers is ESPRESSO-II [4]. This program consists of a set of efficient heuristic algorithms that produce very good solutions in a fraction of the time required by earlier methods. It gives a minimized cover of the on-set of an incompletely specified function from given covers of the on- and don't-care (dc) sets. Its goal is to minimize the number of

product terms, the number of literals in the input part of the cover, and the number of literals in the output part.

A different approach was developed by A. Svoboda [5-9], and later extended by M. Nadler [11]. Svoboda's weight algorithm is a graphical method for the minimization of single-output logic functions. In his work, he reintroduced the use of the Marquand chart (or logical matrix [11]) for the manual application of the algorithm. This method allows considerable reduction in the number of prime implicants generated. In 1987, Vora [12] proposed a new algorithm based on Svoboda's work. This algorithm guarantees the absolute minimum cover for the general problem of multiple-output logic function minimization with arbitrary cost function.

The purpose of this work is to give an efficient computer implementation of the Svoboda-Nadler-Vora algorithm, and compare its performance with that of heuristic minimization methods.

Chapter 2 presents a description of both ESPRESSO-II and the Svoboda-Nadler-Vora algorithm, as examples of two different approaches to boolean function minimization. The necessary definitions and theoretical foundation are given there as well.

Chapter 3 presents the new set of procedures for reducing the cost of prime implicant generation, and details of the ANSI C implementation.

Chapter 4 gives a detailed example of the minimization process. The example shows a case where ESPRESSO-II fails to find the absolute minimum of a boolean function.

Chapter 2

Two Different Approaches to Logic Function Minimization

2.1 Espresso-II

The popularization of programmable logic arrays and other regular structures in VLSI design brought the need for more efficient logic minimization schemes. In the middle 70s a heuristic method called MINI was developed at IBM [1]. MINI was the beginning of a series of heuristic minimization programs that culminated in the development of ESPRESSO-II [4].

ESPRESSO-II is a general purpose heuristic minimization program. It gives a minimized cover of the on-set of an incompletely specified function from given

covers of the on- and D-sets (don't-care). Its goal is to minimize the number of product terms, the number of literals in the input part of the cover, and the number of literals in the output part. The efficiency of its procedures comes mostly from a "divide and conquer" strategy, based on the unate recursive paradigm.

The following sections give an overview of ESPRESSO-II. The unate recursive paradigm and other important mathematical tools are presented. Also, the logic of each of the procedures is discussed in some detail. The main source for this material is the monograph written by the authors of ESPRESSO-II in 1984 [4].

2.1.1 Unate Recursive Paradigm

The basis for the unate recursive paradigm is the Shannon expansion of a logic function. The Shannon expansion uses the concept of cofactor of a logic function, whose definition is given below.

2.1.1.1 Cofactor of a Cube

Let f be a n-input, m-output boolean function, and let c be a cube[1] of a cover of f. The cofactor of c with respect to a cube p, is a cube c_p built as follows:

---

[1] In the following discussion, "cube" refers to a row of the matrix representation of the cover of a boolean function. That is, a cube is a product term of the algebraic representation of the function. For example, (11-0 1-1) is a cube of a 4-input, 3-output function. Note that inputs come first, and are separated from the outputs by a space. A dash "-" represents a don't care.

- If there is a conflict in an input variable for c and p (one has a 1 and the other a 0 in that position), the cofactor c_p is the empty cube.
- When both c and p have a 1 or 0 in an input, that input is raised to a don't care in the cofactor c_p.
- If p does not exist[2] in the output j, c_p does.
- Otherwise, c_p has the same inputs and outputs as c

The cofactor of the set of cubes $G = \{c^i\}$ with respect to a cube p, is the set of cubes $G_p = \{c_p^i\}$, where c_p^i is the cofactor of c^i with respect to p.

For example, if

$$G = \{ (110\text{-}\ \ 10),\ (01\text{-}0\ \ 11) \}$$
$$p = (11\text{--}\ \ 10)$$

Then,

$$G_p = \{ (\text{--}0\text{-}\ \ 11),\ \phi \}$$

Consider g_p^1, the cofactor of the first cube of G. The first two inputs are don't cares because both g^1 and p have a 1 in those positions. The other two inputs and the first output of g_p^1 are the same as for g^1, because those positions p do not fit in any of the special cases. Finally, the second output of g_p^1 is a 1 because p does

---

[2] Existence is defined within an output of a boolean function. A state "exists" in an output if it belongs to the on-set or to the don't care set of that output. A cube "exists" in an output if all the states that constitute that cube exist in that output.

not exist (is a 0) in that output. The cofactor of the second cube of G is the empty cube, because of the conflict in the first input.

The procedures in ESPRESSO-II use cofactorization with respect to a given variable (input or output). The cofactor of a cube c with respect to a variable x_i is a cube (c_{x_i}) that is independent of x_i. For example, let

$$c = (110\text{-}\ \ 10)$$
$$x_0 = (1\text{---}\ \ 11), \quad \bar{x}_0 = (0\text{---}\ \ 11)$$

Then,

$$c_{x_0} = (\text{-}10\text{-}\ \ 10),$$
$$c_{\bar{x}_0} = (\text{----}\ \ 00) = \phi$$

A useful result for recursive algorithms based on the Shannon expansion is the commutativity of cofactorization with intersection and complementation:

$$(FG)_{x_i} = (F_{x_i} G_{x_i})$$

$$(\bar{F})_{x_i} = \overline{(F_{x_i})}$$

2.1.1.2 *Shannon Expansion*

Let G be a cover of a logic function. Let G_{x_i} and $G_{\bar{x}_i}$ be the cofactors of G with respect to a variable x_i and its complement. Then, the Shannon expansion of G is

$$G = x_i\, G_{x_i} + \bar{x}_i\, G_{\bar{x}_i}$$

The effect of the Shannon expansion is easily visualized when using the algebraic representation of a boolean function. Consider the following 4-input, 2-output boolean function

$$F = \begin{bmatrix} F_0 \\ F_1 \end{bmatrix} = \begin{bmatrix} \bar{a}b\bar{d} + abcd + ab\bar{c} \\ \bar{a}b\bar{d} + ab\bar{c} \end{bmatrix}$$

The Shannon expansion of F with respect to input d is

$$\begin{aligned} F &= d\, F_d + \bar{d}\, F_{\bar{d}} \\ &= d \begin{bmatrix} abc + ab\bar{c} \\ ab\bar{c} \end{bmatrix} + \bar{d} \begin{bmatrix} \bar{a}b + ab\bar{c} \\ \bar{a}b + ab\bar{c} \end{bmatrix} \end{aligned}$$

2.1.1.3 *Unate Functions*

A logic function is unate (monotone) in a variable x_i if changing x_i from 0 to 1 causes *all* the outputs that change to do so also from 0 to 1 (or from 1 to 0). Using the last example, the function F is unate in variable b. A function is unate

if it is unate in all its variables. The importance of this class of functions is that they allow very efficient implementations for reduction, tautology checking and complementation.

2.1.1.4 Statement of the Paradigm

The unate recursive paradigm consists of two phases:

1. Split the function into two cofactors (with respect to a given variable), and apply the operation to each cofactor separately.

2. Merge the results by using the Shannon expansion.

This recursive process creates a binary tree and the result at each node will be the merging of the results of its two children. This procedure is interesting when the splitting variable is chosen so as to make each cofactor more "unate" than its parent function since, as mentioned before, several of the operations used by ESPRESSO-II can be implemented very efficiently for unate functions.

2.1.1.5 Merge with Containment

When working with a cover of a logic function, it is important to keep the merged results as compact as possible. Then, if each of the cofactors is a prime irredundant cover, the merged cover should be also prime and irredundant. In

order to achieve this, no cube in the new cover can be contained in any subset of the other cubes in the cover. Since this irredundant merge operation is expensive, the authors opted for an approximate procedure that gives a "good" answer.

This procedure uses the concept of single cube containment (checking that no one cube contains another cube) to ensure "minimality with respect to single cube containment". Before this operation, a fast check is made for identical cubes, in order to reduce the cardinality of the problem. Therefore, the merging step of the unate recursive paradigm is one of the points where ESPRESSO-II loses rigor, and approximates the solution.

2.1.2 Outline of the ESPRESSO-II algorithm.

Let F and D be the covers of the on-set and the don't care set of an incompletely specified boolean function.

Begin

Find the COMPLEMENT of $F \bigcup D$;

UNRAVEL the output part (eliminate any output sharing);

EXPAND: expand each implicant into a prime;

IRREDUNDANT_COVER: Find a minimal irredundant cover;

ESSENTIAL_PRIMES: Move the essential primes into the D;

```
Get (cost);
Do forever
    Begin
        Do While (cardinality of F decreased in iteration)
            Begin
                REDUCE, each implicant to a minimum cube that con-
                tains its essential vertices.
                EXPAND, each implicant into a prime;
                IF (cost improved) BREAK;
                IRREDUNDANT_COVER;
                IF (cost improved) BREAK;
            End
        LAST_GASP:  REDUCE,  EXPAND,  and
        IRREDUNDANT_COVER with a different strategy.
        IF (cost did NOT improve) BREAK;
    End
Move essential primes back to on-set;
MAKE_SPARSE: Make sparse PLA structure;

End;
```

2.1.3 COMPLEMENT

The complement of a function F is computed by recursive use of the Shannon expansion:

$$\overline{F} = x_i \overline{F}_{x_i} + \overline{x}_i \overline{F}_{\overline{x}_i}$$

The recursion is as explained above. At each node of the binary tree a set of special cases is checked. If none is satisfied, then the most "binate" variable (the one that has the most 0's AND 1's) is chosen as splitting variable and the recursion continues.

In the case of the complement it is not really important to obtain a minimal cover, since it is used only to check for implicants in other procedures. Therefore, in order to save time, only the check for identical cubes is performed in the merging process.

A simple example follows to illustrate the recursive complementation process[3] .

Given $F = [(000), (11\text{-})]$, a cover of a 3-input single-output function f. Let the input variables be labeled x_0, x_1, and x_2. $\overline{F}$ can be computed by using the unate recursive paradigm as follows:

---

[3] For completely-specified single-output functions, the output part of a cover will be all 1's. Therefore, it can be dropped without loosing any information.

- Selecting x_0 as the splitting variable, compute the cofactors of F with respect to x_0 and $\bar{x}_0$

$$F_{x_0} = [\text{-}1\text{-}], \quad F_{\bar{x}_0} = [\text{-}00]$$

- Selecting x_1 as the next splitting variable, we have

$$F_{x_0x_1} = [\text{---}] = \mathbf{1}, \quad F_{x_0\bar{x}_1} = \phi$$

and

$$F_{\bar{x}_0x_1} = \phi, \quad F_{\bar{x}_0\bar{x}_1} = [\text{--}0]$$

- Complementing the four leaves of the recursion tree,

$$\overline{F}_{x_0x_1} = \phi, \quad \overline{F}_{x_0\bar{x}_1} = \mathbf{1}$$

and

$$\overline{F}_{\bar{x}_0x_1} = \mathbf{1}, \quad \overline{F}_{\bar{x}_0\bar{x}_1} = [\text{--}1]$$

- Merging on x_1, we get

$$\begin{aligned} \overline{F}_{x_0} &= x_1 \overline{F}_{x_0x_1} + \bar{x}_1 \overline{F}_{x_0\bar{x}_1} \\ &= \phi + [\text{-}0\text{-}] = [\text{-}0\text{-}] \end{aligned}$$

and

$$\overline{F}_{\bar{x}_0} = x_1 \overline{F}_{\bar{x}_0 x_1} + \bar{x}_1 \overline{F}_{\bar{x}_0 \bar{x}_1}$$
$$= [\text{-1-}] + [\text{-0-}][\text{--1}] = [(\text{-1-}),\ (\text{-01})]$$

- Finally, merging on x_0, we have that

$$\overline{F} = x_0 \overline{F}_{x_0} + \bar{x}_0 \overline{F}_{\bar{x}_0}$$
$$= [\text{1--}][\text{-0-}] + [\text{0--}][(\text{-1-}),\ (\text{-01})]$$
$$= [(\text{10-}),\ (\text{01-}),\ (\text{001})]$$

  which is indeed a cover of $\bar{f}$.

2.1.4 TAUTOLOGY

One of the applications of the paradigm within ESPRESSO-II is tautology checking. This auxiliary operation is essential for the procedures IRREDUNDANT_COVER, REDUCE, ESSENTIAL_PRIMES and LAST_GASP.

It can be proven that a set of cubes G covers a cube c if and only if G_c, the cofactor of G with respect to c, is a tautology. Therefore, the process of checking if a cube is covered by a set of cubes is reduced to a tautology check which can be implemented very efficiently with the unate recursive paradigm.

If a Universe cube (all inputs are don't cares and exists in all outputs) is detected in a node, that is a tautology leaf. A parent cover is a tautology only if both

cofactors are tautologies. Therefore, if by checking some special cases a leaf is identified as NOT a tautology, the procedure immediately stops, returning a FALSE. Several heuristics are used to improve the speed of the procedure by reducing the cardinality of the cover.

2.1.5 EXPAND

EXPAND produces a prime cover of *F*, i.e., a cover in which all cubes are prime implicants. The quality of the cover depends on the order in which the cubes are expanded. Therefore, a sort is performed prior to the expansion. The ordering is the same as was used in MINI, which tries to identify the essential cubes and order them first. The expansion of a cube is done in such a way as to cover the maximum number of other cubes, and to make it as big as possible. Since the latest implementation of ESPRESSO-II supports multiple-valued logic, the output variables are treated as a single multiple-valued variable in this procedure.

Heuristics are used to ensure that the expanded cube is disjoint with the set of zeros of the boolean function. The direction of the expansions is selected dynamically in order to cover the most cubes, which yields better results than EXPAND's counterpart in MINI. As a bonus EXPAND identifies some non-essential primes. This will reduce the time spent by ESSENTIAL_PRIMES.

Figure 1 shows a 4-input, 2-output function and two possible expansions for cube (0-00 10) in (a)[4] .

The expansion shown in (c) is apparently better than the one in (b) because it covers more states. This may not be the case in a more complex function, since the larger cube can inhibit the expansion of another cube which could yield a cover of lower cardinality. This is why a good the initial ordering is so important.

2.1.6 ESSENTIAL_PRIMES

Essential primes,[5] by definition, must be present in all prime covers of a logic function. It is therefore desirable to eliminate them as soon as possible in order to reduce the cardinality of the problem.

ESPRESSO-II uses a boolean version of the procedure for essential prime identification for multiple-valued input functions proposed by Sasao [4]. This procedure is interesting as it allows finding the set of essential primes without having to find the set of all prime implicants. An outline of the procedure is given below.

Begin

For_each $c \in F$

---

[4] Throughout this work, the Marquand chart or binary matrix is used for graphical representation of boolean functions. Please refer to section 2.2 for its definition and detailed explanation.

[5] A prime implicant of a boolean function f is essential if it covers a minterm of f not covered by any other prime.

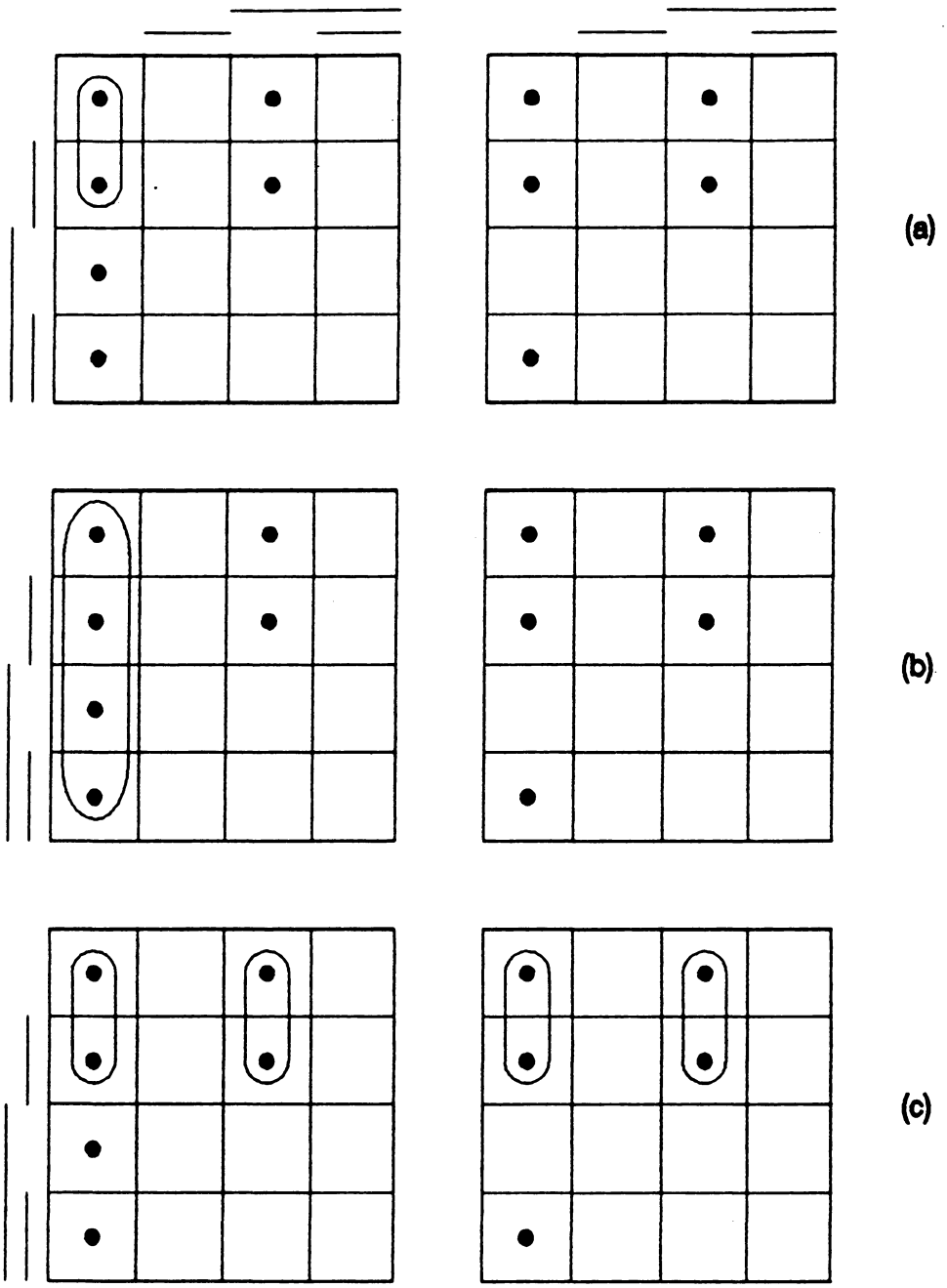


Figure 1. Expansion of a cube.

```
        /* Form consensus of c with all other cubes in F */
        For_each   p ∈ (F ⋃ D) − {c}
            H = H ⋃ consensus(p,c);
        Endfor
        /* c is essential only if it is not covered by H */
        If (c ⊈ H)
            E = E ⋃ {c};
        Endif
    Endfor
    /* Move essentials from F to D. */
    F = F − E;
    D = D ⋃ E;

End;
```

The consensus of two cubes p and q, can be thought of as a subset of the minterms covered by $p \bigcup q$ for which the distance between the cubes[6] is 1 or 0. Consider the 3-input 2-output function given in Figure 2. The consensus of cube (-00 11) with all the other cubes in the cover is shown in (b). A triple line represents the cube resulting of the consensus of a pair of cubes. For example, the consensus of cubes (-00 11) and (0-1 10) is the cube (00- 10). Clearly the

[6] The distance between two cubes is equal to the number of input conflicts plus 1 if the intersection of their output part is empty.

Attention Patron:

Page 20 is missing from
all copies

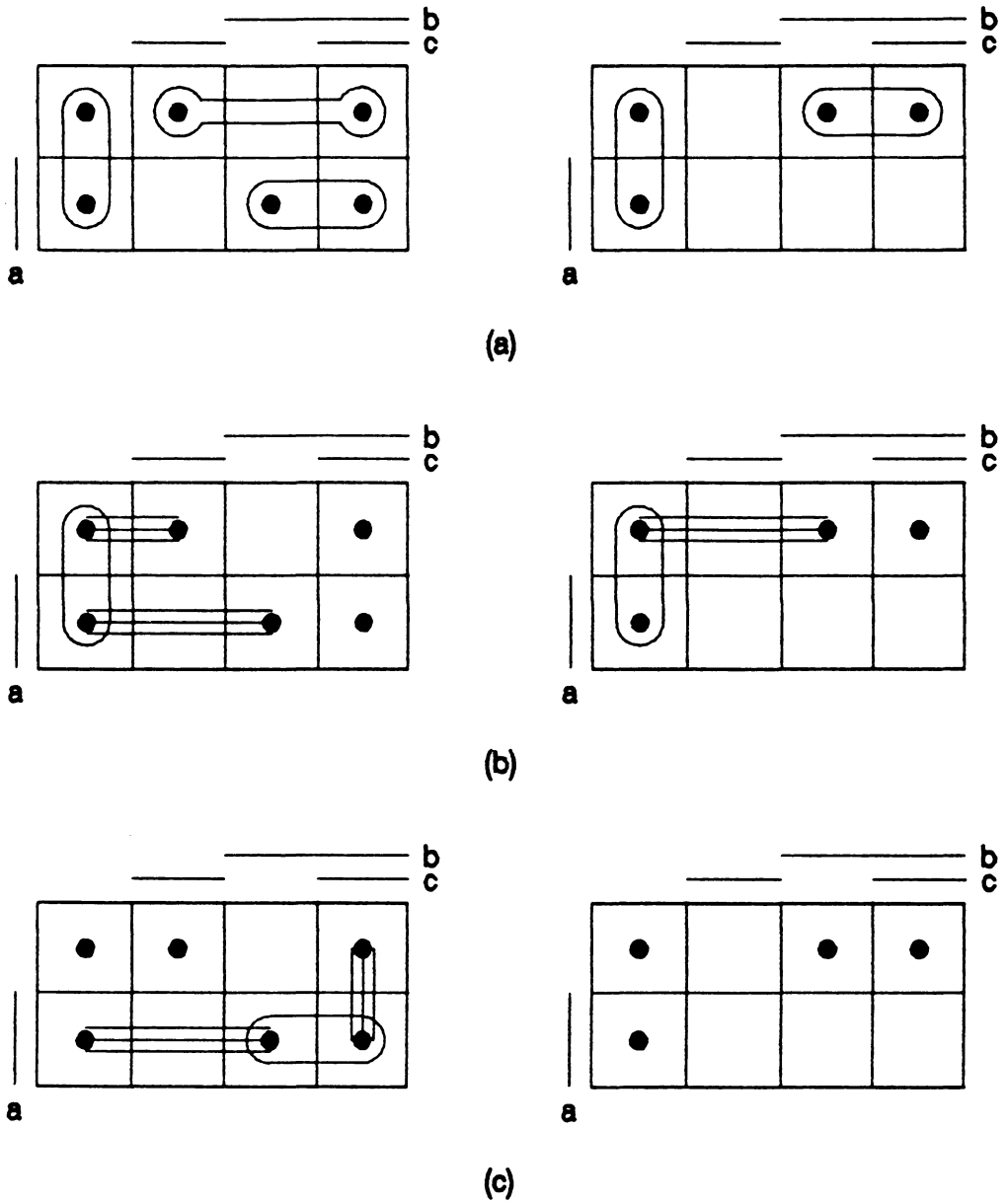


Figure 2. Sasao's method for essential prime identification.

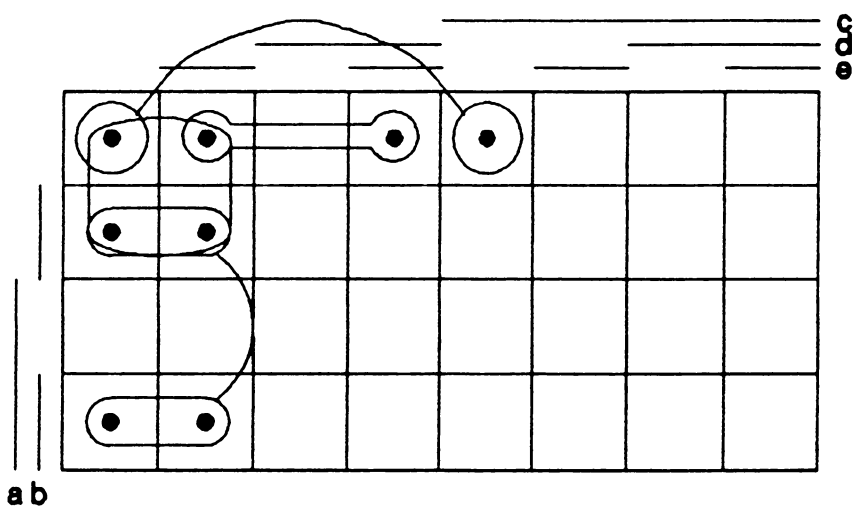


Figure 3. Redundancy in a cover.

EE: relatively essential cubes, $EE = \{p \in F, \ni F \nsubseteq F - \{p\}\}$

RR: redundant cubes, $RR = F - EE$

So, if a cube $c \in EE$ is removed from F, the resultant set of cubes is no longer a cover of the function f. Therefore, the cubes in EE must appear in any *subcover* of F. The set EE must not be confused with the set of essential primes discussed in the previous section.

Further, RR is partitioned into:

R_t : Totally redundant cubes, $R_t = \{p \in RR, \ni p \subseteq (EE \bigcup D)\}$

R_p : Partially redundant cubes, $R_p = RR - R_t$

IRREDUNDANT removes every cube in R_t, the totally redundant set. Then, it attempts to find R_c, a minimal subset of R_p such that $EE \bigcup R_c$ is still a cover for F and of minimum cardinality. Since finding a minimum subset R_c is a minimization problem and therefore an NP-complete one, heuristics are used once again to obtain an approximate solution. This is another point where ESPRESSO-II loses rigor and the absolute irredundancy of the cover cannot be guaranteed.

2.1.8 REDUCE

This procedure transforms each cube $p_i \in F$ into a smaller cube q_i such that the set of all q_i's is also a cover of F. REDUCE allows ESPRESSO-II to move away from a locally optimal solution to a better one. As the new cover will not be

prime any more, EXPAND has an opportunity to generate a different set of primes that may have fewer cubes than the original.

The order in which cubes are reduced has a strong influence on the quality of the final solution. In order to minimize the bias of the sorting method, ESPRESSO-II alternates two types of ordering. One is the order used by the analogous procedure in MINI. The second, is a "crude static reordering strategy": Using the largest cube as a seed, the remaining cubes are ordered inversely to the number of conflicts in the input and output parts. As the cubes are reduced, the degrees of freedom left for the remaining cubes are reduced. Therefore, the largest cube is reduced first (it can be reduced the most) and then the cubes nearest to it, increasing the probability that an expansion of the large cube will cover its neighbors.

The reduced cube q_i is computed as the smallest cube containing the complement of the cofactor of $F \bigcup D$[7] with respect to q_i (the cube being reduced). This operation is trivial when applied to a unate cover. Therefore, the unate recursive paradigm is used to quickly break the cover into unate ones.

Immediately after REDUCE, an expansion step is executed. If the cardinality of the cover decreased, a new iteration begins. If there was no improvement in the solution, the program exits from the main loop, and LAST_GASP is executed.

---

[7] F is a cover of the "ones" of the function. D is a cover of the "don't cares" of the function.

2.1.9 LAST_GASP

This procedure attempts to reduce the number of cubes by applying a modified version of REDUCE followed by a modified version of EXPAND. The rationale of this procedure is that by reducing the size of the cover, a reduced cube can either be covered by a neighboring cube after expansion, or the reduced cube can expand in a different direction to cover some neighboring cube.

The reduction step is independent of the ordering because each cube is reduced "maximally" without regard to the other cubes. The new cover H will have more freedom for expansion. As the expanded H may not be a cover of F, EXPAND deletes from H any expanded cubes that do not cover any other cube in H . Furthermore, *all* cubes in H are expanded (even those that get covered by other expanded cubes).

Finally the cover H is submitted to IRREDUNDANT_COVER, which adds the primes in H to F, and finds a new irredundant cover. If this step produces a smaller cover, then ESPRESSO-II loops back to the inner loop for a REDUCE/EXPAND/IRREDUNDANT sequence. Otherwise, it breaks the loop and the procedure MAKE_SPARSE is executed.

2.1.10 MAKE_SPARSE

After including the essential primes back into the on-set, this procedure attempts to make the PLA matrix as sparse as possible. This procedure removes redundancies in the output part and then executes an expansion step limited to the input part, in order to raise to don't cares as many inputs as possible.

2.2 Svoboda-Nadler-Vora Algorithm

Exact logic minimization techniques have been displaced by the more efficient heuristic logic minimizers available nowadays. The dominant cost of the exact minimization methods is the generation of all prime implicants, which has a worst case complexity of $O(3^n/n)$.

A different approach was developed by Svoboda [5-9]. Svoboda's weight algorithm is a graphical method for single-output logic function minimization. This method allows considerable reduction in the number of prime implicants generated.

For manual minimization, he reintroduced the use of the Marquand chart [10] (also called logical matrix [11]). In contrast to the better known Karnaugh map, the logical matrix is encoded in straight binary. This is illustrated in Figure 4,

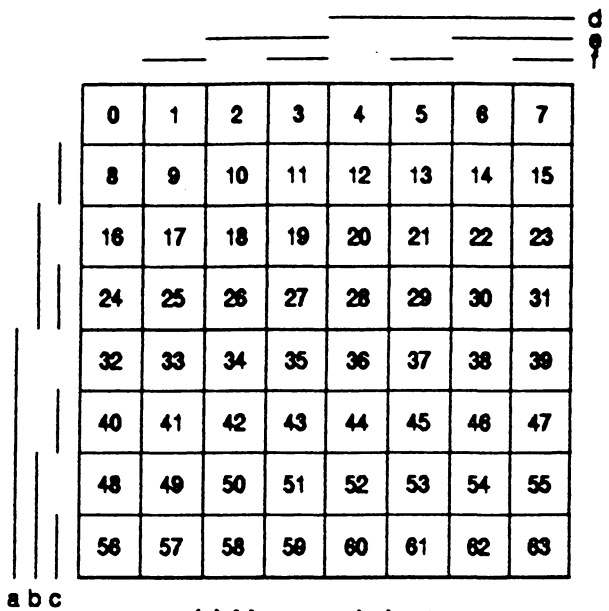
where decimal indices are used to represent the binary codes. The binary matrix is much more convenient for problems with more than 6 variables, since the neighborhood patterns extend naturally as the matrix grows (see definition of neighbor states in subsection 2.2.1 below). Using the combination of Svoboda's weight algorithm and the logical matrix, functions as large as 12 variables have been manually minimized in practical situations [11].

Svoboda's theorems were later extended by Nadler [11]. He condensed the method into two rules. The first one, for selection of prime implicants, which extends Svoboda's definitions to properly handle incompletely specified functions. And the second one defines a branching procedure for implicants that fail to satisfy the first rule.

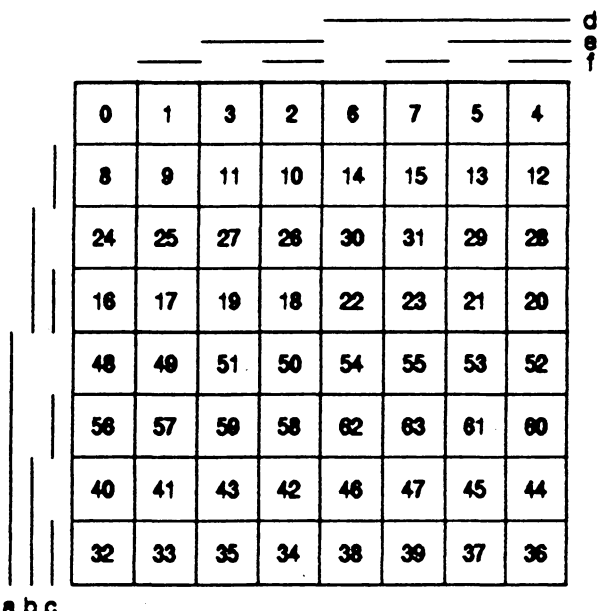
In 1987 Vora [12] proposed a new algorithm based on Svoboda's work. This algorithm guarantees the absolute minimum cover for the general problem of multiple-output logic function minimization with arbitrary cost function. Before describing the algorithm in detail, some definitions are in order.

2.2.1 States

A state is simply one of the 2^n minterms of an n -variable boolean function. A state is either a 1-state, a 0-state or a D-state (don't care). The binary index of a state (s-index) is the binary representation of a minterm.



(a) Marquand chart



(b) Karnaugh map

Figure 4. Graphical representations of a 6-variable boolean function.

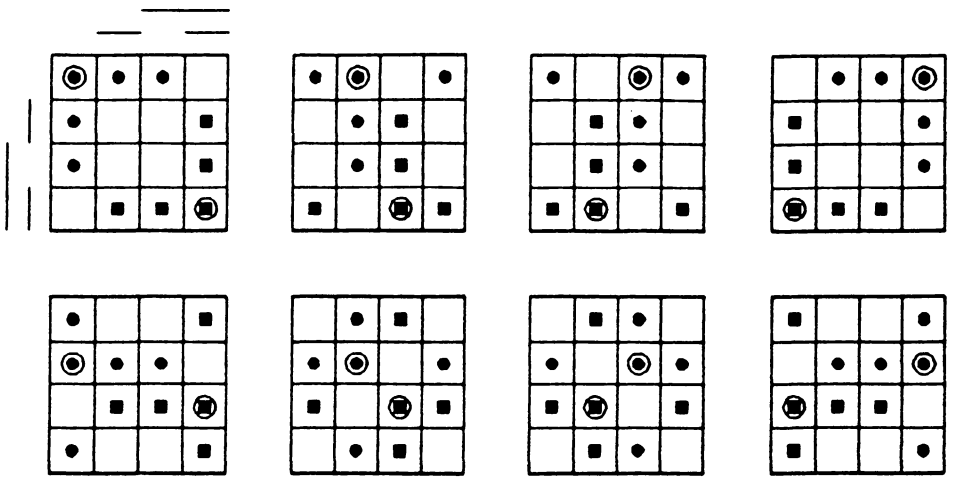
Let s be a state of an n-variable logic function. Let $d(a,b)$ be the distance of two binary codes (number of bits where they differ). Then the set of neighbors r of s is the set $N = \{r \ni d(s,r) = 1\}$.

In the Karnaugh map representation of a boolean function, neighbor states are in adjacent squares (considering corresponding squares in opposite edges to be adjacent). Neighbors in a binary matrix follow a well determined set of patterns, as shown in Figure 5 [11]. This patterns have as a basic unit a 4 variable matrix. The symmetry of the patterns and their natural extension for larger matrices are apparent.

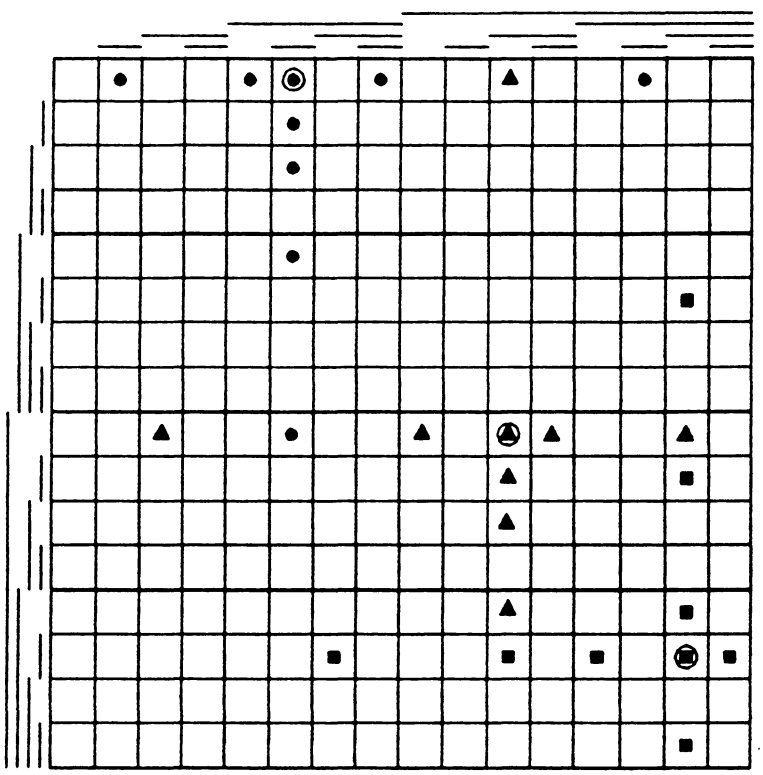
2.2.2 K-cells

A k-cell (or binary cube) is a set of 2^k states, such that each state in the set has exactly k neighbors within the set. In other words, a k-cell is a group of 2^k states, in which the index of any state differs from that of exactly k other states by one bit. It follows that for an n variable function a k-cell can be represented by $(n-k)$ variables. This concept is illustrated in Figure 6. Note the convenient representation of the states: a 1-state is represented by a square with a dot, a D-state is represented by a hatched square, and 0-states are left blank.

Any k-cell can be represented by the ordered pair (g,k), where g is the generic k-cell index (g-index), and k the definite k-cell index (k-index).



(a) Four variables



(b) Eight variables

Figure 5. Adjacency patterns in the Marquand chart.

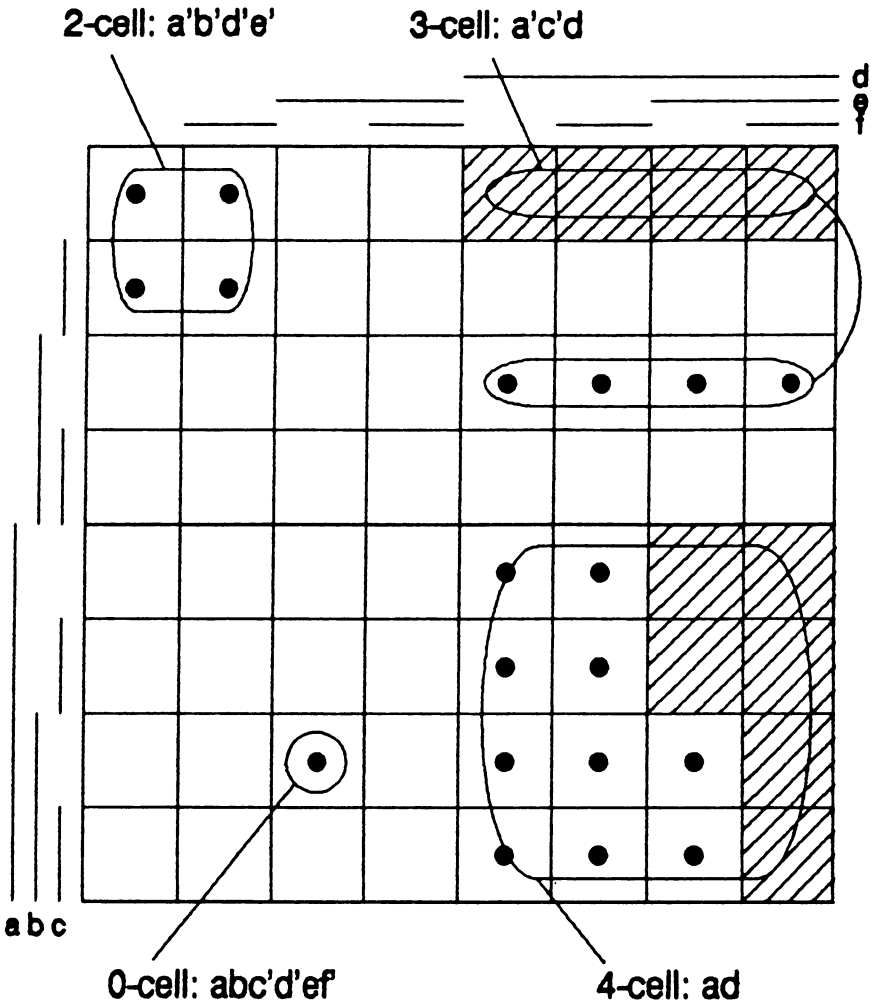


Figure 6. Examples of k-cells in a 6-input boolean function.

The g-index gives the input dependency of the k-cell, and is defined as:

$$g = (g_{n-1}, \ \ldots \ , g_1, g_0),$$

$g_i =$ 0, if k-cell does not depend on input variable i

1, otherwise (variable i is present in the term)

The k-index indicates which input variables are complemented or not, and is defined as:

$$k = (k_{n-1}, \ \ldots \ , k_1, k_0),$$

$k_i =$ 0, if i is complemented or if the k-cell does not depend on it.

1, otherwise (variable i is not complemented)

As an illustration, let $f(a,b,c,d)$ be a 4-variable boolean function. Then, we have:

$$ab\bar{c} = (1110, 1100),$$
$$\bar{b}\bar{d} = (0101, 0000), \ldots$$

An important relationship between g-index, k-index and s-index (binary index of a state) is given by:

$$(k - index) \ = \ (s - index) \text{ AND } (g - index)$$

This equation can be used to generate a k-cell for a given k that covers state s. Also given a k-cell (g,k), the above relationship can be used to test if it covers a state s.

2.2.3 Other Definitions

2.2.3.1 Free state:

A 1-state which has not yet been covered by any of the k-cells selected in the minimal form.

2.2.3.2 Bound state:

A 1-state which has been covered by a k-cell selected in the minimal form, or a D-state.

2.2.3.3 Maximally covered state:

A free state s is maximally covered by a given k-cell k, if k exists in all the outputs

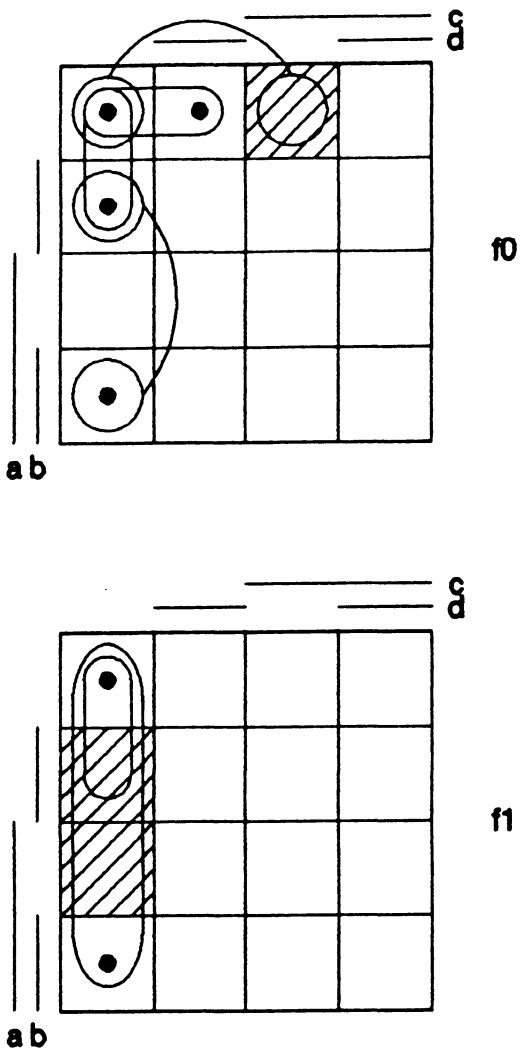


Figure 7. Example of a 4-input, 2-output boolean function.

where s exists. In Figure 7, 1-cell $\bar{a}\bar{c}\bar{d}$ is a maximal cover for free state 0^8, since it exists in both outputs where free state 0 exists. The 2-cell $\bar{c}\bar{d}$ is not a maximal cover for free state 0.

2.2.3.4 Maximal k-cell:

or prime implicant in a single-output function, is a k-cell which is not a subcell of any other k-cell, both existing in the function. The 2-cell $\bar{c}\bar{d}$ in the second output of the function in Figure 7, is maximal.

2.2.3.5 Maximal subcell:

(in a multiple-output function) is a subcell of a single-output maximal k-cell, that it is the highest order k-cell shared in any other output. In Figure 7 the 1-cell $\bar{a}\bar{c}\bar{d}$ is not a maximal k-cell in the second output, but it is a maximal subcell because is the highest-order k-cell shared in both outputs, covering free state 0.

2.2.3.6 Multiple-output Maximal k-cell:

or multiple-output prime implicant, is either a maximal k-cell of any single output or a maximal subcell for the multiple-output function.

---

[8] When bit-level manipulations are not necessary, the s-index of a state will be expressed as its decimal equivalent.

2.2.3.7 Reference state:

Free state, such that the maximal k-cells covering it do not cover those k-cells covering another free state within the same output.

2.2.3.8 Dominating state:

Free state for which, within a given output, a subset of the k-cells covering it, cover all the k-cells covering another free state (dominated state). A dominating state cannot be a reference state, since when a k-cell is chosen to cover the dominated state, the dominating state will be automatically covered.

For example, the free states 1 and 4 in Figure 7 are reference states. Free state 0 is not a reference state, because it dominates free state 1.

2.2.3.9 Corresponding reference state:

Is the state in a different output than the reference state's output, that has the same index as the reference state. This state can be free, bound or may not exist at all in a given output.

2.2.3.10 Corresponding k-cell:

Analogous to a corresponding reference state. If a k-cell is shared in two outputs, in each covering a free state, this k-cell is said to be interfering between the two outputs.

2.2.3.11 Cost functions:

1. **PLA area cost function:**

   The cost of a PLA can be computed in terms of the real state that it uses in the integrated circuit. This is given by:

   $$\text{PLA area cost} = \text{product terms} \times (\text{used inputs} + \text{outputs})$$

2. **Gate-input cost function:**

   (for a two level multiple-output AND-OR implementation) Each term (k-cell) requires an AND-gate, with as many inputs as the number of literals present in the term. The output of each AND-gate is one input line to the OR-gate. Then, the cost for a k-cell is equal to the number of 1's in its g-index, plus one, if the OR-gate exists.

For an $(n-1)$-cell (a one literal term), there is no AND-cost, so the cost of that k-cell is just the OR-cost (equal to 1). If for an output, one k-cell covers all free states in that output, there is no OR-cost. Then, the cost of that k-cell is just the AND-cost (equal to the number of literals).

When a k-cell is selected for an output, its cost in any other output is now just the OR-cost, as the outputs can share the same AND gate. Thus, for a k-cell that is already selected, its cost is equal to 1.

2.2.4 Algorithm Statement

Rule 1: **Selection Rule**

A multiple-output maximal k-cell K_i covering a reference state s in a in an output f, may be included in the minimal expression of f, if K_i is the only k-cell covering s in f.

Rule 2: **Elimination Rule**

If K_i is not the only k-cell covering s in f, an alternate k-cell K_j covering s in f can be eliminated, provided that:

K_j does not cover a free state not covered by K_i in f;

AND

from ALL the outputs where it exists if:

a. K_j is not of a lower cost than K_i;

AND

b. in any other output f' where corresponding reference state s' exists, corresponding k-cell K_j' does not cover a free state not covered by corresponding k-cell $K_{i'}$ in f'.

OR ONLY from output f if:

c. $K_{i'}$ has already been selected in any other output f'.

Rule 3: **Branching Rule**

If no k-cells can be selected or eliminated for any of the possible reference states, a reference state s with least number of alternate k-cell covers is selected. Each of those alternate k-cells is selected one at a time. For each of these selection branches, rules 1 and 2 are applied as before until all states are covered or the cost of the branch exceeds a cost cutoff value at that point. The minimum form(s) are given by the branches with the least cost.

2.2.5 Computer Implementation

The computer implementation of the algorithm given in [12] uses several techniques to reduce the number of prime implicants generated. K-cell generation begins with the higher order k-cells first. The idea behind this is that larger

implicants are likely to cover more states. As a state is maximally covered, there is no further generation of implicants covering that state.

The k-cells are generated by a loop that generates all possible g-indexes for a given k (Combinations of k bits-at-1 within an n-bit word, where n is the number of variables). Let N_{max} be the maximum number of non-zero states for any output of the function. Then, the highest order possible for a k-cell to exist in the function (K_m) is given by:

$$K_m = \text{int}\,(\log_2 N_{max})$$

Then, for k varying from K_m to 0, all g-indexes for each order are generated, until all the free states are maximally covered by at least one prime implicant. Each k-cell is generated by the bit-wise AND of the current g-index with each of the remaining free states. The new k-cell is stored only if it has not been previously generated (by a neighbor state) and if it actually exists in at least one output of the function. The free states covered by this k-cell are marked as bound only if they are maximally covered. If that is not the case, a new set of k-cells of the next lower order will be generated, until all the states are covered.

Concurrency is introduced into the algorithm by applying rules 1 and 2 (elimination/selection) after each order of g-index generation. At this point only maximally covered states are considered. The advantage of selecting k-cells at this stage is that not only maximally covered states, but any free state covered by the selected k-cells will be converted into don't cares. This produces a

cardinality reduction at each iteration. If a branching situation arises while k-cells are still being generated, no action is taken, and the branching is delayed until all the states have been maximally covered.

This implementation of the algorithm shows a clear advantage over the traditional Quine-McCluskey method [13-15]. But, when compared with ESPRESSO-II, the latter has shown a speed advantage of at least one order of magnitude. Since ESPRESSO-II generally gives a very good approximation of the solution, the extra computation time still cannot be justified.

Chapter 3

A New Implementation

The combination of Svoboda's weight index and Vora's extensions offers the possibility of a significant increase in the efficiency of prime implicant generation and identification. A new implementation is proposed next, in terms of a set of improvements. These improvements result in a greater execution speed without compromising the mathematical rigor of the method.

3.1 Data Structures

The data structures have been completely redesigned. The program uses bit vectors to represent states and k-cells, with their associated information. As this scheme uses only the necessary amount of bits to represent each item, the memory

requirements are greatly reduced. Also, it facilitates the handling of functions with large number of inputs and outputs. The overhead of retrieving the information from the bit vectors is minimized by generating all the necessary extraction masks at the initialization phase.

For an n-input, m-output function, the bit vector representing a state has the following information:

- An n-bit field for the state of the index.
- An n-bit field with existence flags for the neighbors.
- A 2m-bit field with a two bit code for each output.

The bit vector representing a k-cell has a similar structure:

- An n-bit field for the definite index (k-index)
- An n-bit field for the generic index (g-index)
- An m-bit field with "exists" flags.
- An m-bit field with "selected" flags.

This structure allows efficient implementation of set and bit-wise operations. For example, a fundamental operation in the generation of maximal k-cells is the maximal cover test. A free state *s* is maximally covered by a k-cell (*g*,*k*), if (*g*,*k*) exists in all outputs where *s* exists as a free state. The test is simply a bit-wise AND of the output existence flags:

$$\bar{s}_{out} \text{ AND } (g,k)_{out}$$

If the result is zero, then (g,k) exists in all outputs where s exists and is a maximal cover.

For example, given a 5-input, 3-output function, the state (00101 11-) is maximally covered by the 3-cell (-01-- 110).

3.2 Memory Allocation

The memory requirements for the minimization of large functions can have a strong influence on the performance of the program. Besides an efficient internal representation for the data, it is desired that memory accesses within a given module span only a limited address space. This would reduce paging activity when working with large sets of data.

The use of an exponential buddy system [17] provides an efficient memory management scheme. A modification to this scheme allows block sizes of $h_k = (2^k + 2^{(k-1)})/2$, besides the usual $h_k = 2^k$.

A hashing scheme allows fast searches for a block of a given size. As memory blocks are made available for reuse, they are collected in free lists, one for each

block size. The hash value for a memory block is, therefore, a function of its size. It is computed only once, (during allocation) and stored on a header field. When a block is in a free list, the header field is used for storing the pointer to the next block in the list.

3.3 Generation of Prime Implicants

As stated before, the most expensive operation in the exact logic minimization process is the generation of prime implicants. Svoboda's weight index allows the concurrent generation and selection of prime implicants, which can substantially reduce the number of generated primes. The ability to take advantage of that concurrency strongly depends on the k-cell generation sequence.

In the previous implementation of the algorithm [12], the prime implicant generation process was k-cell order oriented. That is, the entire set of g-indexes for a given k (order) is generated in each iteration. For each g-index, a k-cell is generated with each free state, and then tested for existence and primality. This process still allows a great deal of redundancy. In the new implementation the k-cell generation will be state oriented, i.e., the prime implicants are generated for each free state. The basic idea here is that the order of the largest possible prime implicant can be very different for each of the free states. So, by generating each

k-cell from the reference state and its neighbors, many higher order k-cells that do not exist in the function will not be generated.

The concept of weight, as defined by Svoboda, is fundamental for the new k-cell generation sequence. The weight of a free state is defined as the number of its neighbor states (both 1 and D-states). The extra information that the weight of a state provides, is processed by a new set of heuristics, as outlined below. In this discussion, a function of n inputs and m outputs is assumed. Also, if a state exists in more than one output, it is considered separately for each of those outputs. For example, the state (1101 101) would be expanded into (1101 100) and (1101 001).

3.3.1 Identification of Neighbors

First, for each free state in the function, all existing neighbors are identified and stored (one bit for each within the state data structure). Each state will have at most n neighbors, but in general less than n will actually exist. The identification is done by comparing each free state with all other states, until the n neighbors are found. This procedure has a complexity of $O(n_f^2)$, where n_f is the number of free states (a state is counted once for each function where it exists).

The complexity of the process can be reduced by observing that once we identify a free neighbor, the bit corresponding to the only input where they differ can be

set on both states. Thus, the number of neighbors that need to be checked for each free state is reduced at each iteration. Besides, as all the neighbors of a state are identified, it can be excluded from the list of states (for this procedure). This process is illustrated in the detailed example in Chapter 4.

3.3.2 Essential 0-cells and 1-cells

Once the neighbors have been identified, there are two special cases that can be taken care of:

1. If a free state *s* has no neighbors, *s* itself is an essential 0-cell, and it must be a part of any minimum cover. Therefore *s* is marked as a don't care, and selected immediately.

2. If a free state *s* has only one neighbor, check if *s* is maximally covered by the 1-cell covering both states. If it is, then the 1-cell is essential and can be selected immediately, and state *s* is marked as a don't care. Otherwise, it is appended to the list of k-cells covering *s*. Note that if the neighbor of *s* is also free, it is processed accordingly.

The above procedure is outlined in the following pseudocode fragment:

```
Begin

    foreach free state s,
```

```
if number_of_neighbors(s) = 0, then
    g = 2**n - 1;       /* ... g-index
    k = s;              /* ... k-index
    put (g, k) in minimum form;
    set(s, DCare); /* essential 0-cell
elseif number_of_neighbors(s) = 1, then
    sn = neighbor(s);
    g = ones_complement( xor(s, sn) );
    k = and(g, s);
    if maximally_covers((g, k), s), then
        Put (g, k) in minimum form;
        set(s, DCare);
        if sn is free, then
            set(s, DCare);
        endif;
    else  /* just store the k-cell as a cover of s
        Store (g, k) in list of covers of s;
        Store ((2**n - 1), s) in list of covers of s;
        set(s, Bound);
        If sn is free, then
            Store (g, k) in list of covers of s;
        endif;
    endif;
```

```
        endif;
    endfor;

End;
```

As an example, let $s = (0111)$ be a free state of a single-output boolean function f. Let $n = (0101)$ be another state of f, and the only neighbor of s. Then the 1-cell covering both states is easily obtained as

$$\begin{aligned} \text{g-index} &= \overline{(0111)\ \text{XOR}\ (0101)} \\ &= \overline{(0010)} \\ &= (1101) \end{aligned}$$

and

$$\begin{aligned} \text{k-index} &= \text{g-index AND s-index} \\ &= (1101)\ \text{AND}\ (0111) \\ &= (0101) \end{aligned}$$

In binary cube representation the 1-cell is (01-1), which obviously covers both states.

3.3.3 Generation of k-cells

The next step is the generation of prime implicants until all free states are bound. An upper bound on the size of the largest k-cell that can exist in output j is given

by the largest integer smaller than the $\log_2$ of the number of states in output j (hereafter referred to as LNSO = $\log_2$ of the number of states in an output). Also, an upper bound on size of the largest possible k-cell that covers a state is given by the number of neighbors for that state (hereafter referred to as NNS).

The prime implicant generation process begins by initializing the order k with the smallest of both upper bounds specified above. Then, for each free state, k-cells are generated for decreasing values of k . The generic indexes are computed with the same method used in Vora's implementation [12]. That procedure moves k 1's in n (number of inputs) positions, in all possible combinations.

A further reduction of redundancy can be achieved at this point: As a state is associated with a neighbor to form a k-cell covering it, the input where they differ will be a don't care in that k-cell. Since the g-index of a k-cell gives its input dependency, the bits corresponding to existing neighbors will be 0's in the g-index. This method is better illustrated with an example.

> Let $f(a,b,c,d,e)$ be a 5-input single-output boolean function. Let rs = (01100) be a free state in f, with neighbors: (01101), (01000), and (11100).
>
> The state data structure stores neighbors by setting the bit where they differ from the reference state. In this case, (10101).
>
> Suppose that NNS $\leq$ LNSO. Then, k is initialized to 3.

The g-index for 3-cells is generated by moving $5 - 3 = 2$ 1's within the five bit positions, which gives $\binom{5}{3} = 10$ possible combinations.

Using the fact that the bits corresponding to neighbors must be 0's in the g-index (input don't cares), the number of combinations is then reduced to just one: g-index = (01010).

The above is valid for k = NNS. When the k-cell generated does not exist, or it does not maximally cover the reference state, then lower order k-cells must be generated. This is also true when LNSO < NNS.

When k < NNS, lower order k-cells can be generated directly from the g-index for k = NNS simply by excluding one or more neighbors each time. This is done by changing from 0 to 1 the bit(s) corresponding to the excluded neighbor(s) in the g-index for k = NNS.

Following the previous example, if k-cell (01010, 01000) exists, it is a maximal cover for rs, because the function is single output. Now, assuming it does not exist in f, k is decremented to 2.

The 2-cells are generated by systematically excluding the 3 neighbors, one at a time:

Reference state index: (01100)

Original g-index (k = 3): (01010)

Excluding neighbor: (01101)

Generated 2-cell (g,k): (01011,01000)

Excluding neighbor: (01000)

Generated 2-cell (g,k): (01110,01100)

Excluding neighbor: (11100)

Generated 2-cell (g,k): (11010,01000)

Therefore, the number of generated 2-cells was reduced from $\binom{5}{2} = 10$ to just 3.

Lower-order k-cells are generated until a maximal cover is found. Note that it is necessary to generate all maximal k-cells of the current order covering the reference state. The selection/elimination rules will choose the minimum cost k-cell at a later stage.

In general, the number of possible prime implicants covering a given state in an n-input function is 2^n. This is given by the number of ways that 0 or more neighbors of that state can or cannot exist. When the techniques introduced above are used, that number is reduced to 2^{NNS}. Or better, if LNSO < NNS, the count is

$$\binom{NNS}{NNS} + \binom{NNS}{NNS-1} + \cdots + \binom{NNS}{LNSO}$$

In the last expression, the first term corresponds to 0-cells, the second to 1-cells, and so on, until the LNSO-cells, which are the largest possible. Furthermore, any redundancy in the generation of g-indexes is totally eliminated.

3.4 Selection of the minimum cover

The selection of the prime implicants that form part of the minimum cover is by direct application of the selection/elimination and branching rules, given before. The major improvements in this stage come from the chosen data structures in combination with optimized C code.

3.5 Portability

The computer program is written completely in ANSI C. Unix compatibility is ensured by conditional compilation, where necessary. Both 16- and 32-bit architectures are supported.

Chapter 4

A detailed example

The following example illustrates the steps involved in the minimization process for both ESPRESSO-II and the new implementation of the Svoboda-Nadler-Vora algorithm. As it will be seen, ESPRESSO-II produces an approximate cover that is one cube larger than the absolute minimum. Also, as ESPRESSO-II is a PLA minimizer, the PLA-area cost function is used for the exact minimization. The binary matrix for the 5-input 3-output boolean function used in the example is given in Figure 8.

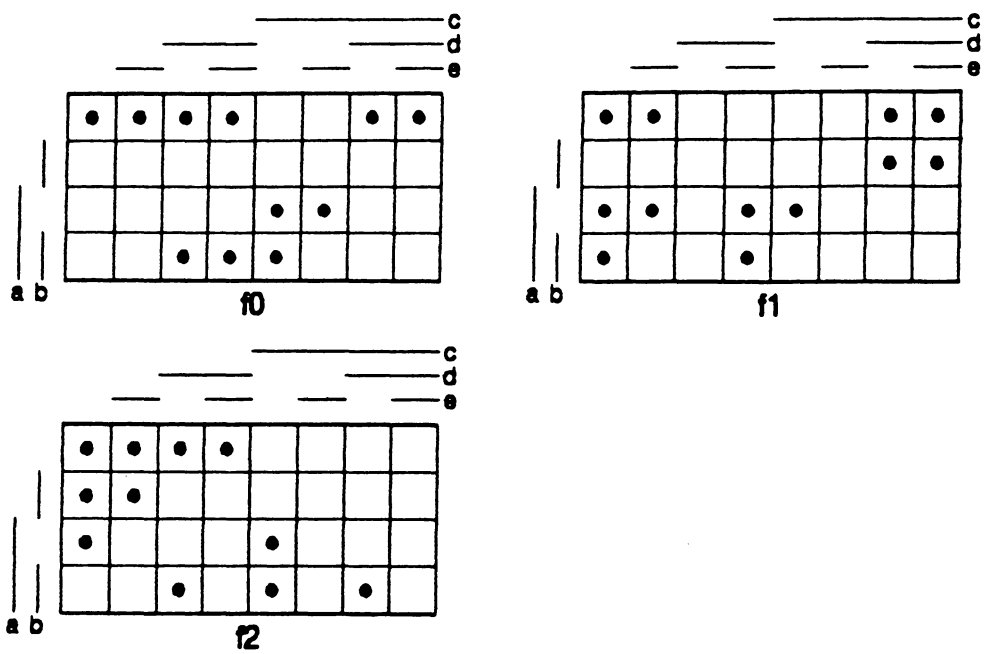


Figure 8. Binary matrix for the example function.

4.1 ESPRESSO-II solution

ESPRESSO-II accepts as input the cover of the on-set and dc-set of an incompletely specified boolean function. It manipulates the cover in its matrix representation. In that matrix, each row corresponds to a cube and each column is a input or output variable. In this example, each stage of the minimization is mapped into a binary matrix, which is a more convenient representation for the visualization of the process.

After the data input and initialization processes, ESPRESSO-II eliminates any output sharing from all the cubes[9] by creating a copy of the cube for each output where it exists. Then the cubes are sorted according to the MINI strategy, previous to the first EXPAND. Figure 9 shows the order in which the cubes will be expanded.

The expansion of the cubes follows the indicated order. However, only cubes that have not been covered by any other cube are considered for expansion. Figure 10 shows the expanded cubes. A state with a small square marks the original cube, before expansion.

---

[9] Note that the cubes in the initial cover are all minterms (0-cubes). This is particular of this example and its is not necessary in the general case.

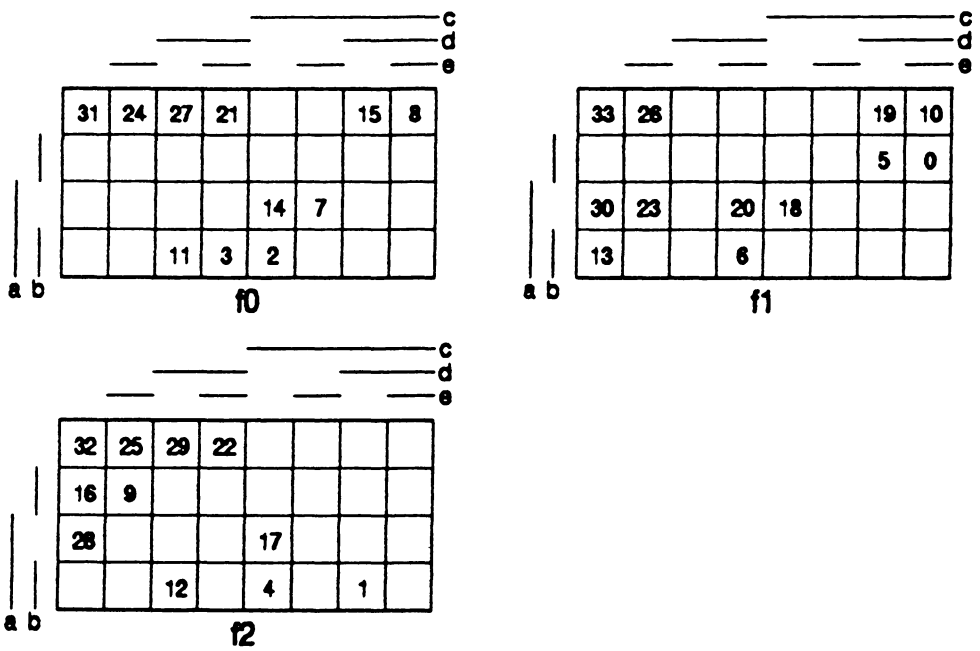


Figure 9. Cubes prior to first EXPAND.

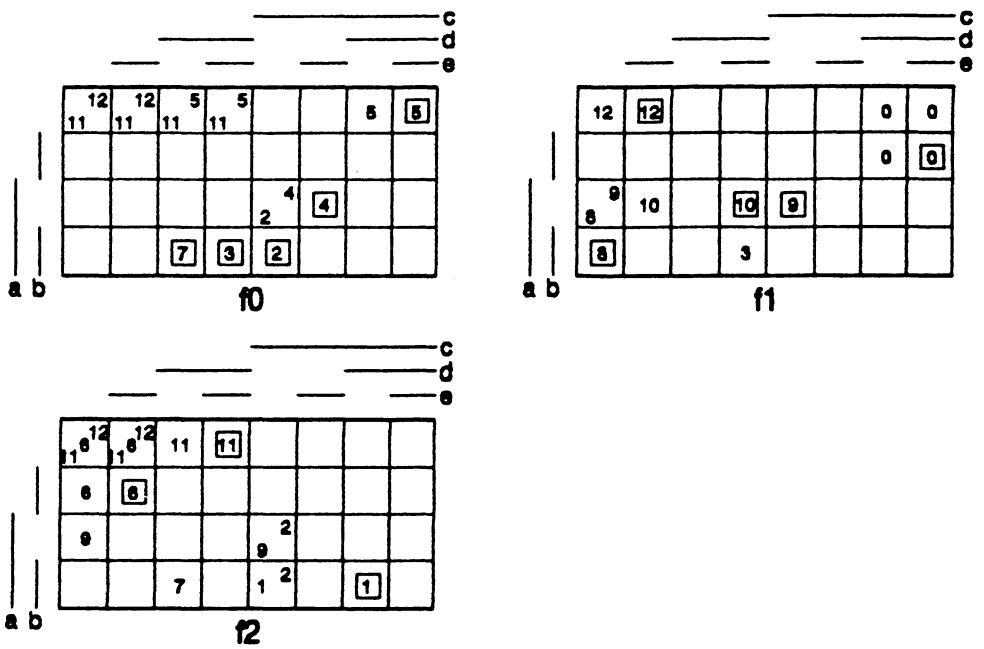


Figure 10. Cubes after first expansion.

The next step is a redundancy check, but in this case there are no redundant cubes. This can be seen directly in Figure 10. Note that all of the expanded cubes cover at least one 1-state not covered by any other cube.

Before entering the main loop ESPRESSO-II identifies the essential primes. As explained before, Sasao's method is used to check if a cube is essential or not. Figure 11 illustrates both cases:

1. Cube $\bar{a}cd$ in output 1

   The only cube that is at distance 1 or less from $\bar{a}cd$ is $\bar{a}\bar{b}d$ in output 0. Then, we have:

   (0-11- 010) $\bar{a}cd$

   (00-1- 100) $\bar{a}\bar{b}d$

   (0011- 110) Consensus

   Since the consensus does not cover $\bar{a}cd$, then it is essential.

2. Cube $abc\bar{e}$ in input 2

   Two cubes are at distance 1 or less from $abc\bar{e}$:

   (111-0 001) $abc\bar{e}$

   (1-100 101) $ac\bar{d}\bar{e}$

   (11100 101) Consensus

   (111-0 001) $abc\bar{e}$

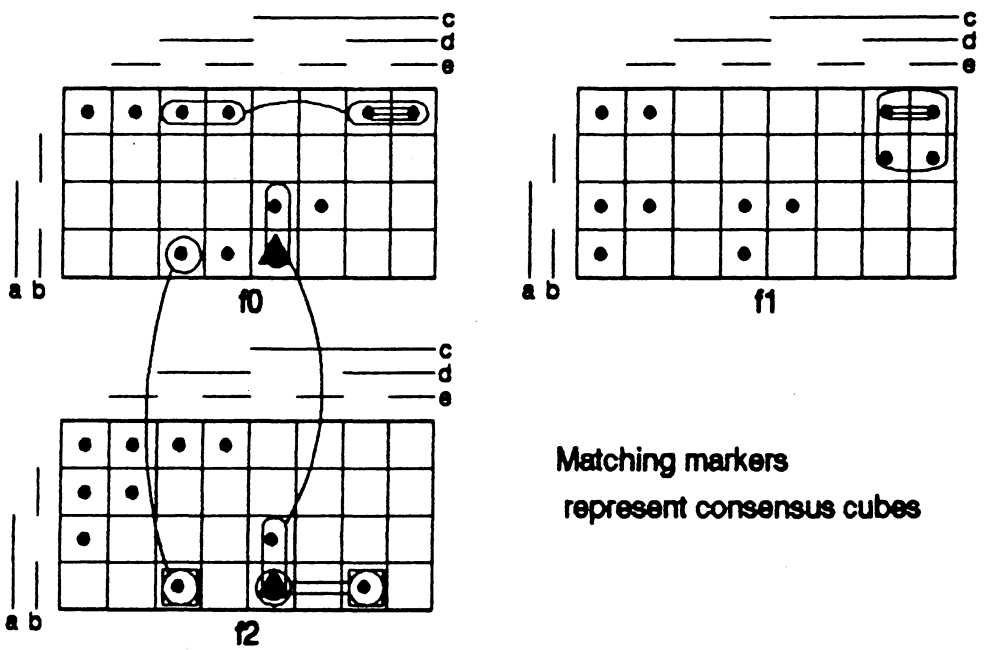


Figure 11. Identifying essential primes.

(11010 101) $ab\bar{c}d\bar{e}$

(11-10 001) Consensus

The union of both consensus cubes cover all the minterms covered by $abc\bar{e}$. Therefore, it is non-essential.

All the other cubes are processed similarly. Once the essential primes are identified, they are moved to the don't care set. Then the remaining cubes are sorted in descending size order, in preparation for the reduction step. This is illustrated in Figure 12.

REDUCE replaces each cube by the smallest cube possible contained in it, so that the set of cubes still covers all the minterms of the original cover. Basically, this eliminates the overlaps that may exist among cubes. The reduced cover is shown in Figure 13, along with the MINI sort order of the cubes prior to the next expansion.

The second expansion yields the cover shown in Figure 14. Again, there are no redundant cubes in the expanded cover. Since the cost of the cover did not improve, the main loop terminates. For this example, the LAST_GASP routines do not yield any improvements either. Therefore, the essential primes are moved back into the on-set. The resultant cover is given in Figure 15

The last step attempts to make the structure of the PLA as sparse as possible. MAKE_SPARSE is implemented in two steps. The first one, Lower Outs, elim-

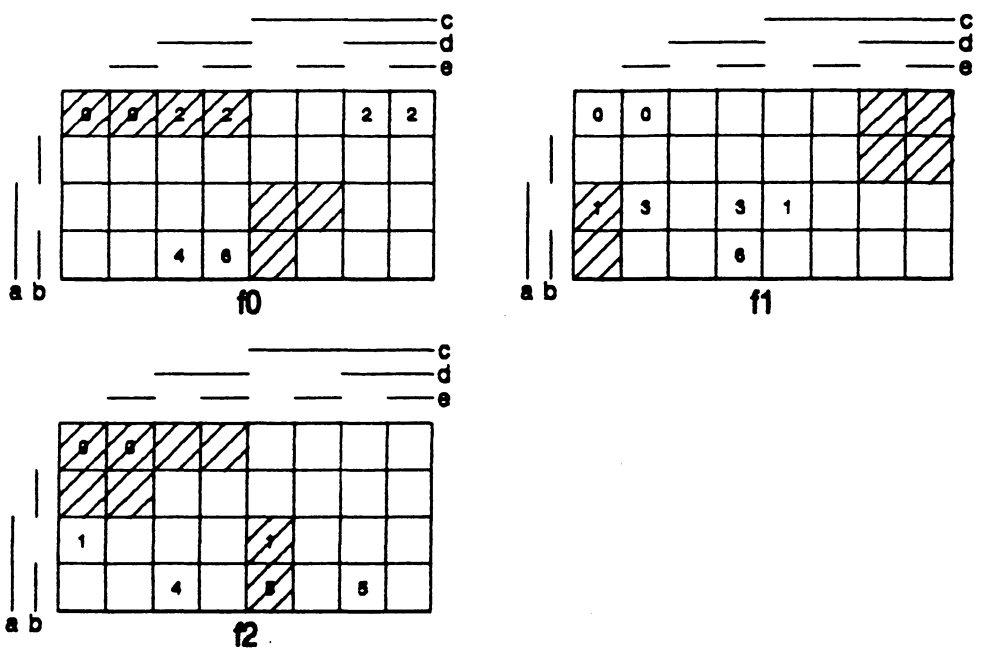


Figure 12. Move essential primes to the don't care set and sort for REDUCE.

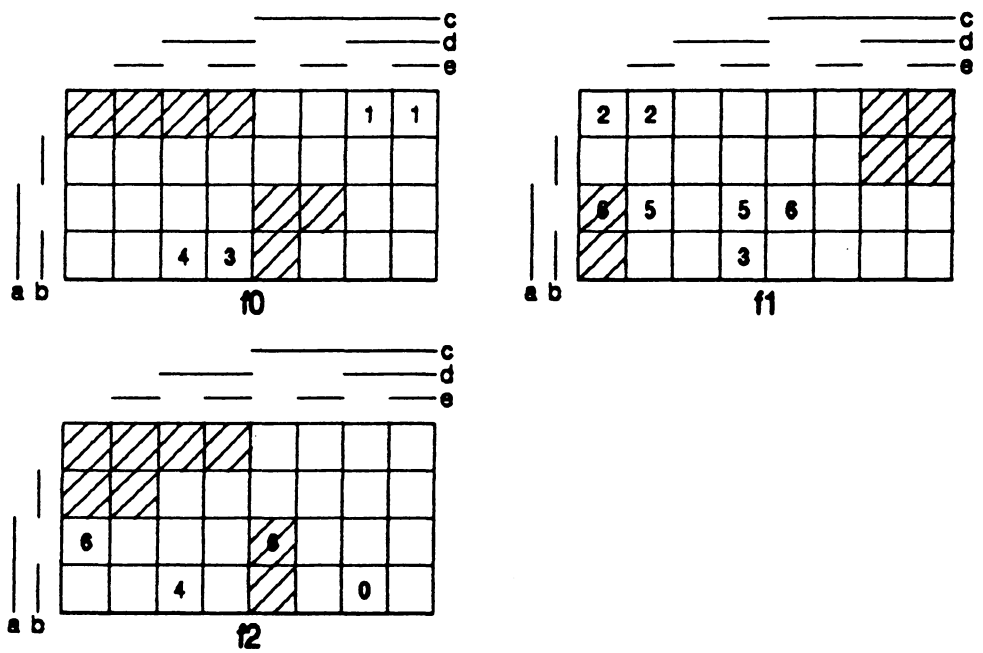


Figure 13. Reduced cover and MINI sort prior to second expansion.

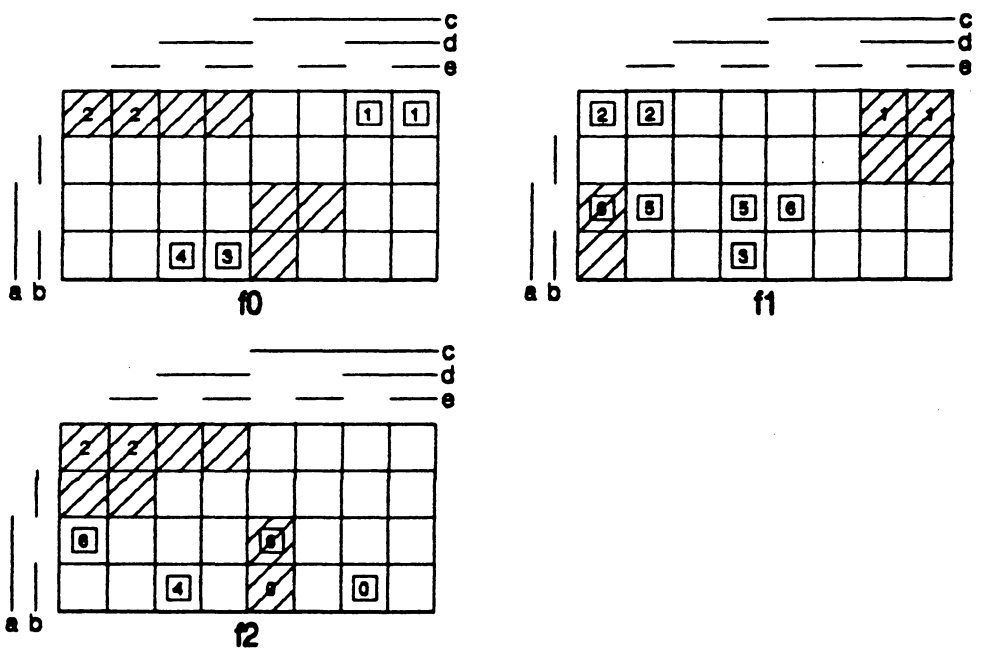


Figure 14. Result of the second expansion.

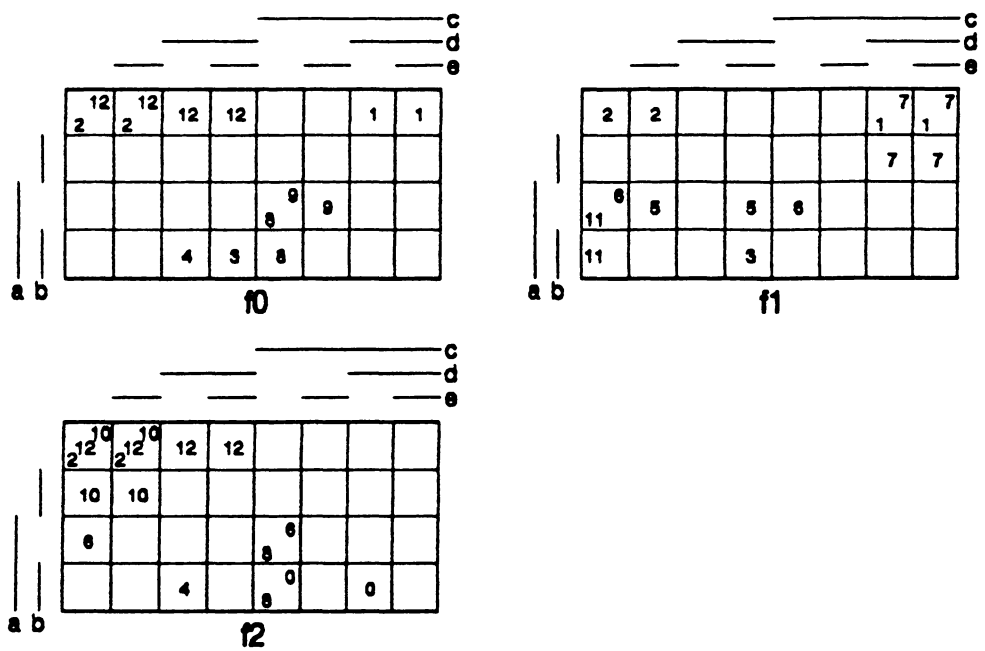


Figure 15. Essential primes are moved back to the cover.

inates some overlap in the output variables. The cubes affected by the first step are no longer primes. This allows a possible expansion of the input part of those cubes, which is exactly what the second step, Raise_In, attempts to do. The effects of both procedures are illustrated in Figure 16.

The cover produced by RAISE_IN is the solution found by ESPRESSO-II. Note that the cover has 13 cubes. In the next section, the exact solution is found to have only 12 cubes. These results are discussed and analyzed in Chapter 5.

4.2 Svoboda-Nadler-Vora solution

4.2.1 Initialization

First, the program reads and sets up the data structure for the truth table of the function. Then, it initializes the masks for retrieving the data from the bit vectors, the data structure for the minimum cover, and several other globals.

4.2.2 Identification of neighbors

The next step is the identification of the neighbors for each of the free states. This is carried out as specified in section 3.3.1. For each existing neighbor of a

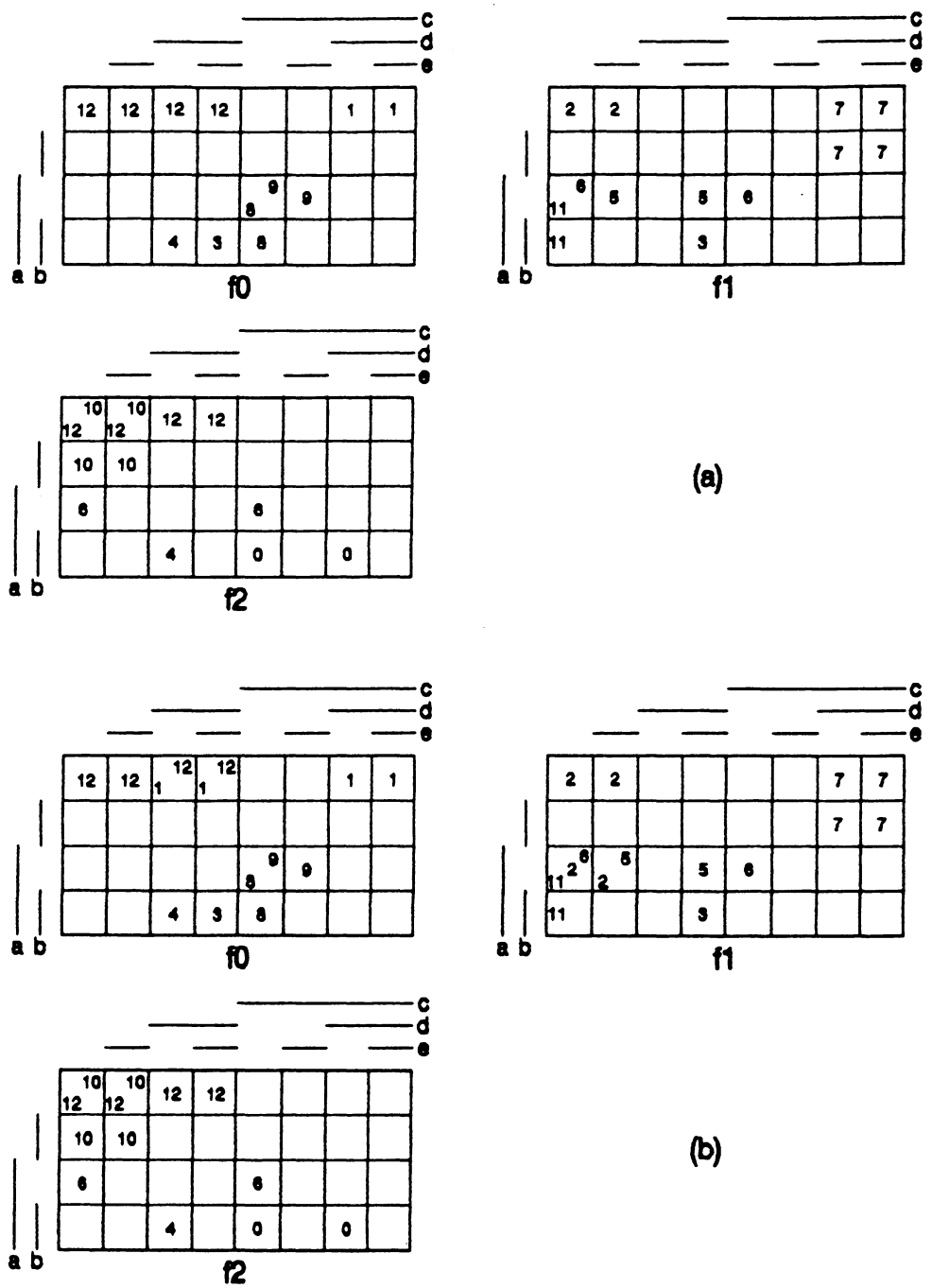


Figure 16. MAKE_SPARSE procedures: (a) Lower_Outs, (b) Raise_In.

state s, a bit is set in the bit vector of s at the position corresponding to the variable where both states differ. The result is shown in the binary matrix in Figure 17. There, the convention of representing each neighbor with a stroke is used, as suggested in [11]. After all the free states have been processed, they are sorted by increasing number of neighbors in each of the outputs.

4.2.3 Essential 0- and 1-cells.

This procedure uses only the free states with 1 or no neighbors. All the others are ignored at this stage. The binary matrix in Figure 18 summarizes the results obtained here.

1. **Output 0.**

   a. State 26:

      The 1-cell (1101- 100), formed with neighbor 27 is not a maximal cover because it does not exist in output 2. Therefore,

      Store 1-cell (1101-, 100)

      Store 0-cell (11010 101)

      Bound state 26.

   b. State 27:

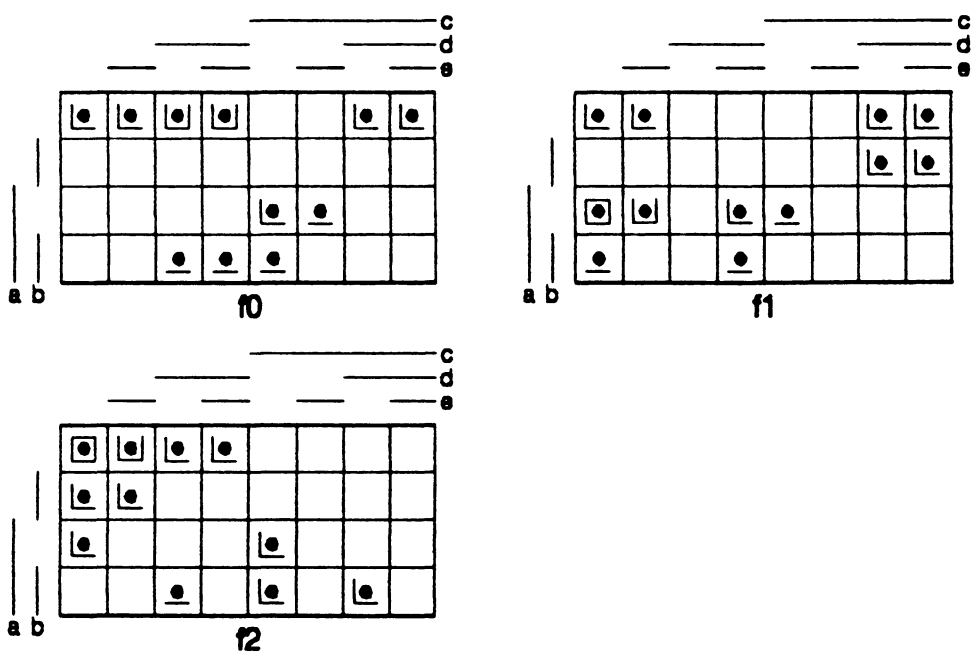


Figure 17. Identification of neighbors.

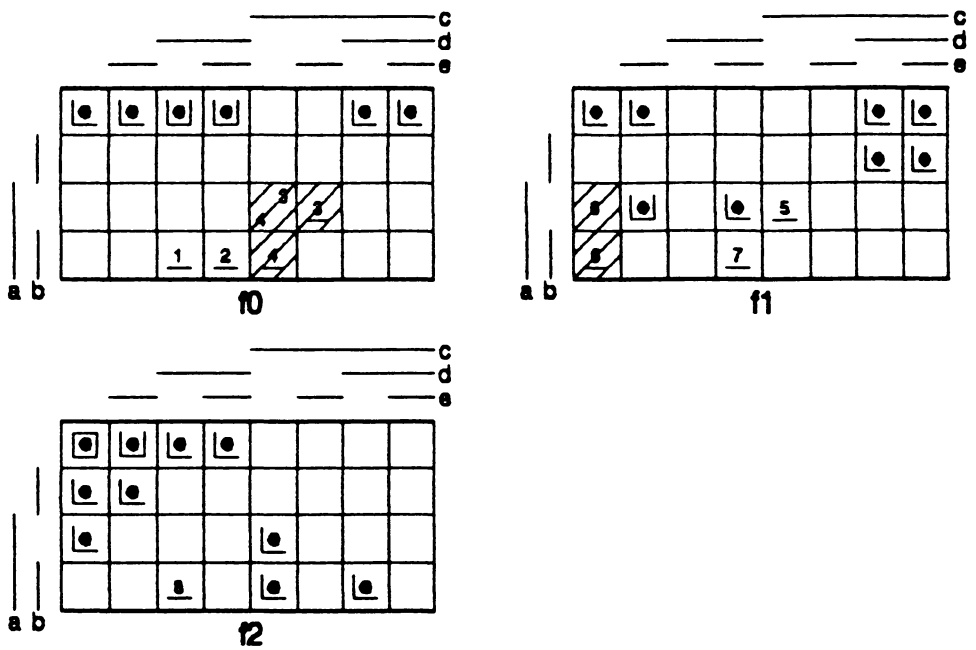


Figure 18. Essential 0- and 1-cells

The 1-cell (1101- 100), formed with neighbor 26 is not a maximal cover because it does not exist in output 1. Also, it is already stored (from state 26). Therefore,

Store 0-cell (11011 110)

Bound state 27.

c. State 21:

The 1-cell (1010- 010), formed with neighbor 20 is a maximal cover. Therefore,

Store and select 1-cell (1010- 010)

New don't cares: 21, 20.

d. State 28:

The 1-cell (1-100 101), formed with neighbor 20 is a maximal cover. Therefore,

store and select 1-cell (1-100 101)

New don't cares: 28.

2. **Output 1.**

a. State 20:

Attention Patron:

Page 72 is missing from
all copies

Attention Patron:

Page 73 is missing from
all copies

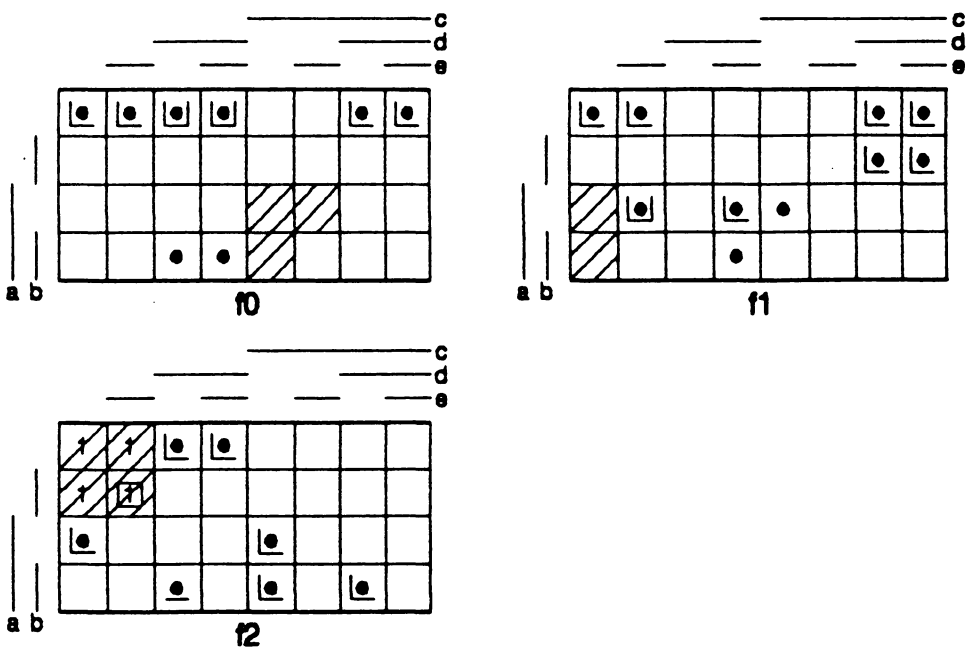


Figure 19. Coverage after generation of maximal k-cells.

(000-0 101): Stored.

b. State 6: (NNS = 2)

- 2-cells:

  (00-1- 100): not a maximal cover. Stored.

- 1-cells:

  (0011- 110): maximal cover. Stored. State bound.

  (00-10 100): Stored.

c. State 1: (NNS = 2)

- 2-cells:

  (000-- 101): not a maximal cover. Stored.

- 1-cells:

  (0000- 111): maximal cover. Stored. State bound.

  (000-1 101): Stored.

d. State 7: (NNS = 2)

- 2-cells:

  (00-1- 100): not a maximal cover. Stored.

- 1-cells:

  (0011- 110): maximal cover. Stored. State bound.

  (00-11 100): Stored.

e. State 2: (NNS = 3)

- 3-cells:

  (00--- 000): Does not exist.

- 2-cells:

  (000-- 101): maximal cover. Stored. State bound.

  (00-1- 100): Stored.

  (00--0 100): Does not exist.

f. State 3: (NNS = 3)

- 3-cells:

  (00--- 000): Does not exist.

- 2-cells:

  (000-- 101): maximal cover. Stored. State bound.

  (00-1- 100): Stored.

  (00--1 100): Does not exist.

2. **Output 1.**

a. State 0: (NNS = 2)

- 2-cells:

  (-000- 010): not a maximal cover. Stored.

- 1-cells:

(0000- 111): maximal cover. Stored. State bound.

(-0000 011): Stored.

b. State 6: (NNS = 2)

- 2-cells:

  (0-11- 010): not a maximal cover. Stored.

- 1-cells:

  (0011- 110): maximal cover. Stored. State bound.

  (0-110 010): Stored.

c. State 14: (NNS = 2)

- 2-cells:

  (0-11- 010): maximal cover. Stored. State bound.

d. State 1: (NNS = 2)

- 2-cells:

  (-000- 010): not a maximal cover. Stored.

- 1-cells:

  (0000- 111): maximal cover. Stored. State bound.

  (-0001 010): Stored.

e. State 7: (NNS = 2)

- 2-cells:

  (0-11- 010): not a maximal cover. Stored.

- 1-cells:

  (0011- 110): maximal cover. Stored. State bound.

  (0111- 010): Stored.

f. State 15: (NNS = 2)

- 2-cells:

  (0-11- 010): maximal cover. Stored. State bound.

g. State 19: (NNS = 2)

- 2-cells:

  (1-0-1 000): Does not exist.

- 1-cells:

  (100-1 010): maximal cover. Stored. State bound.

  (1-011 010): Stored.

h. State 17: (NNS = 3)

- 3-cells:

  (-00-- 000): Does not exist.

- 2-cells:

  (100-- 010): Does not exist.

(-000- 010): maximal cover. Stored. State bound.

(-00-1 010): Does not exist.

3. **Output 2.**

a. State 7: (NNS = 2)

- 2-cells:

  (000-- 101): maximal cover. Stored. State bound.

b. State 8: (NNS = 2)

- 2-cells:

  (0-00- 001): maximal cover. Stored.

  Since it is the only cover for the state, it is essential. Therefore,

  (0-00- 001) is selected for the minimum cover.

  Covered States: 0, 1, 8, 9.

c. State 20: (NNS = 2)

- 2-cells:

  (1--00 000): Does not exist.

- 1-cells:

  (10-00 011): not a maximal cover. Stored.

  (1-100 101): not a maximal cover. Stored.

- 0-cell:

  (10100 111): maximal cover. Stored. State bound.

d. State 28: (NNS = 2)

- 2-cells:

  (1-1-0 000): Does not exist.

- 1-cells:

  (111-0 001): not a maximal cover. Stored.

  (1-100 101): maximal cover. Stored. State bound.

e. State 16: (NNS = 2)

- 2-cells:

  (-0-00 000): Does not exist.

- 1-cells:

  (10-00 011): not a maximal cover. Stored.

  (-0000 011): maximal cover. Stored. State bound.

f. State 2: (NNS = 2)

- 2-cells:

  (000-- 1-1): maximal cover. Stored. State bound.

g. State 30: (NNS = 2)

- 2-cells:

  (11--0 000): Does not exist.

- 1-cells:

  (111-0 001): not a maximal cover. Stored.

  (11-10 011): not a maximal cover. Stored.

4.2.5 Minimization.

The minimization stage consists of the direct application of the k-cell selection/elimination and branching rules. If a free state *s* is not a reference state it is covered immediately. This is because when its dominated state is covered, *s* is automatically covered.

1. **First iteration:**

   Figure 20 shows the progress of the minimization after the first iteration.

   a. **Output 0.**

      1) State 27:

         State 27 is a reference state.

         None of the k-cells can be eliminated because each covers a free state not covered by the others.

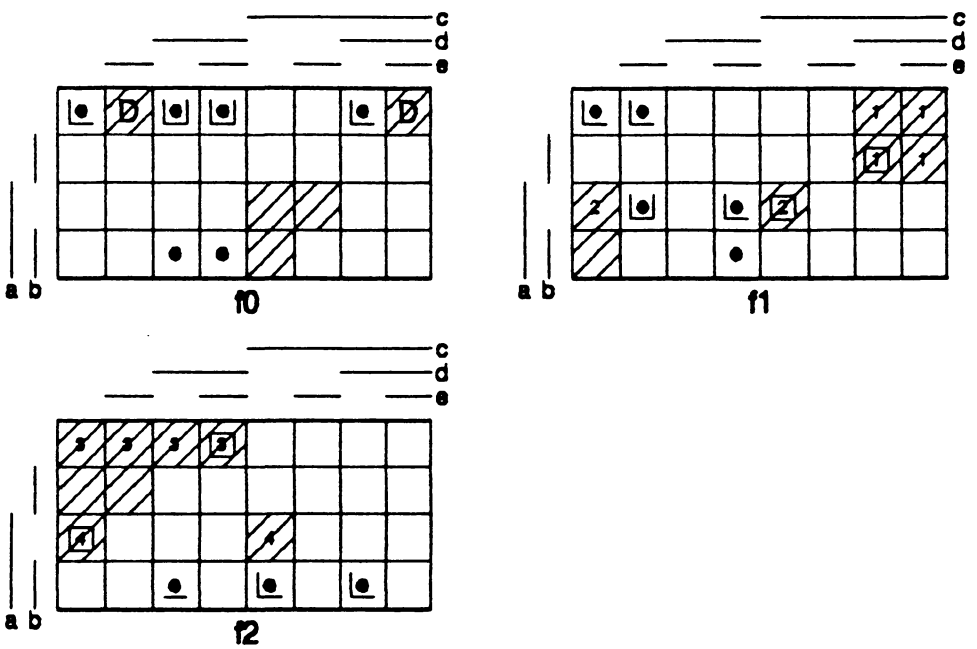


Figure 20. Coverage after the first iteration.

2) State 0:

State 0 is a reference state.

K-cell $\bar{a}\bar{b}\bar{c}$ eliminates $\bar{a}\bar{b}\bar{c}\bar{e}$ from all the inputs.

3) State 6:

State 6 is a reference state.

K-cell $\bar{a}\bar{b}d$ eliminates $\bar{a}\bar{b}d\bar{e}$ from all the inputs.

4) State 1:

State 1 dominates state 0. It is covered immediately.

5) State 7:

State 7 dominates state 6. It is covered immediately.

6) State 26:

State 26 is a reference state.

None of the k-cells covering it can be eliminated yet.

7) State 2:

State 2 is a reference state.

None of the k-cells covering it can be eliminated yet.

8) State 3:

State 3 dominates state 2. It is covered immediately.

b. **Output 1.**

1) State 27:

State 27 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 0:

State 0 is a reference state.

None of the k-cells covering it can be eliminated yet.

3) State 6:

State 6 dominates state 14. It is covered immediately.

4) State 14:

State 14 is a reference state.

K-cell $\bar{a}cd$ eliminates $\bar{a}cd\bar{e}$ from all the outputs.

K-cell $\bar{a}cd$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 6, 7, 14, 15.

5) State 1:

State 1 is a reference state.

K-cell $\bar{b}c\bar{d}$ eliminates $\bar{b}\bar{c}\bar{d}e$ from all the outputs.

6) State 19:

State 19 is a reference state.

None of the k-cells covering it can be eliminated yet.

7) State 17:

State 17 is a reference state.

None of the k-cells covering it can be eliminated yet.

8) State 20:

State 20 is a reference state.

K-cell $a\bar{b}\bar{d}\bar{e}$ eliminates $a\bar{b}c\bar{d}\bar{e}$ from all the outputs.

K-cell $a\bar{b}\bar{d}\bar{e}$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 20 (16 is already covered).

c. **Output 2.**

1) State 28:

State 28 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 3:

State 3 is a reference state.

K-cell $\bar{a}\bar{b}\bar{c}$ eliminates $\bar{a}\bar{b}\bar{c}e$ from all the outputs.

K-cell $\bar{a}\bar{b}\bar{c}$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 2, 3 (0 and 1 are already a covered).

3) State 16:

State 16 is a reference state.

K-cell $a\bar{b}\bar{d}\bar{e}$ eliminates $\bar{b}\bar{c}\bar{d}\bar{e}$ only from output 2. This is because $a\bar{b}\bar{d}\bar{e}$ was already selected in another output.

K-cell $a\bar{b}\bar{d}\bar{e}$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 16, 20.

4) State 30:

Attention Patron:

Page 87 is missing from
all copies

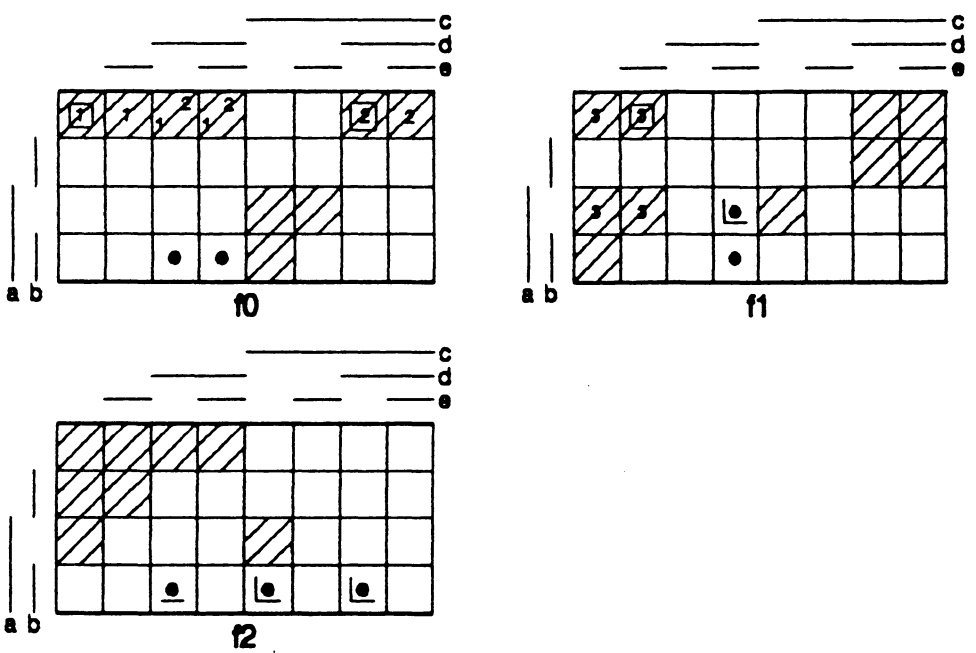


Figure 21. Coverage after the second iteration.

3) State 26:

State 26 is a reference state.

None of the k-cells covering it can be eliminated yet.

4) State 6:

State 6 is a reference state.

K-cell $\bar{a}\bar{b}d$ eliminates $\bar{a}\bar{b}cd$ from all the outputs.

K-cell $\bar{a}\bar{b}d$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 6, 7 (2 and 3 are already covered).

b. **Output 1.**

1) State 27:

State 27 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 0:

State 0 dominates state 1. It is covered immediately.

3) State 19:

State 19 is a reference state.

None of the k-cells covering it can be eliminated yet.

4) State 17:

State 17 is a reference state.

None of the k-cells covering it can be eliminated yet.

5) State 1:

State 1 is a reference state.

K-cell $\bar{b}\bar{c}\bar{d}$ eliminates $\bar{a}\bar{b}\bar{c}\bar{d}$ from all the outputs.

K-cell $\bar{b}\bar{c}\bar{d}$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 1, 17 (0 and 16 are already covered).

6) State 30:

State 30 is a reference state.

None of the k-cells covering it can be eliminated yet.

7) State 26:

State 26 is a reference state.

None of the k-cells covering it can be eliminated yet.

8) State 28:

State 28 is a reference state.

None of the k-cells covering it can be eliminated yet.

c. **Output 2.**

1) State 30:

State 30 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 26:

State 26 is a reference state.

None of the k-cells covering it can be eliminated yet.

3) State 28:

State 28 is a reference state.

None of the k-cells covering it can be eliminated yet.

3. **Third iteration:**

The progress of the minimization after this iteration is given in Figure 22.

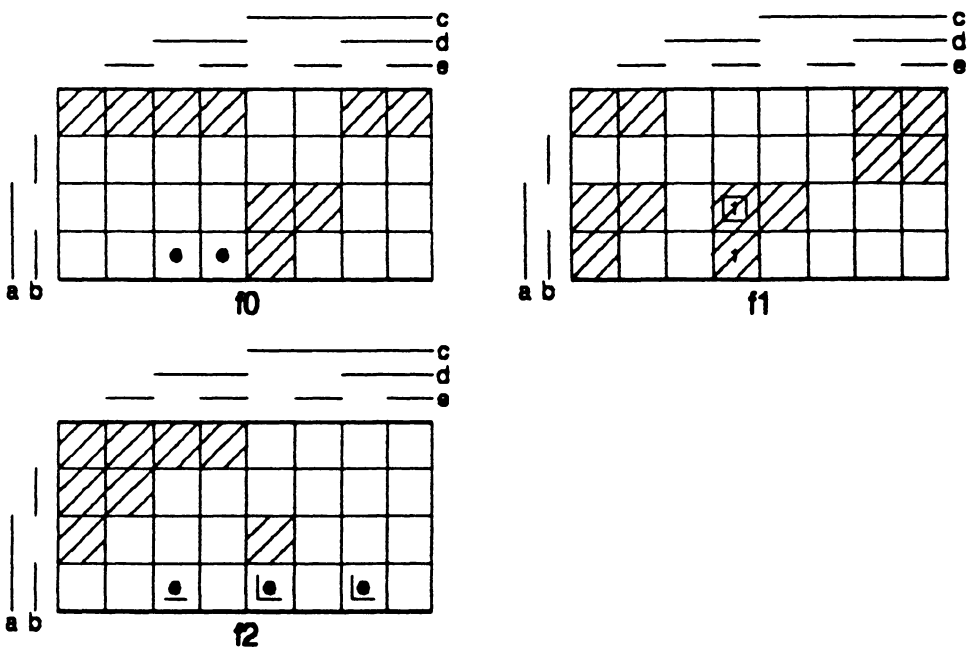


Figure 22. Coverage after the third iteration.

a. **Output 0.**

1) State 27:

State 27 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 26:

State 26 is a reference state.

None of the k-cells covering it can be eliminated yet.

b. **Output 1.**

1) State 27:

State 27 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 19:

State 19 is a reference state.

K-cell $a\bar{c}de$ eliminates $a\bar{b}\bar{c}e$ from all the outputs.

K-cell $a\bar{c}de$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 19, 27.

Attention Patron:

Page 94 is missing from
all copies

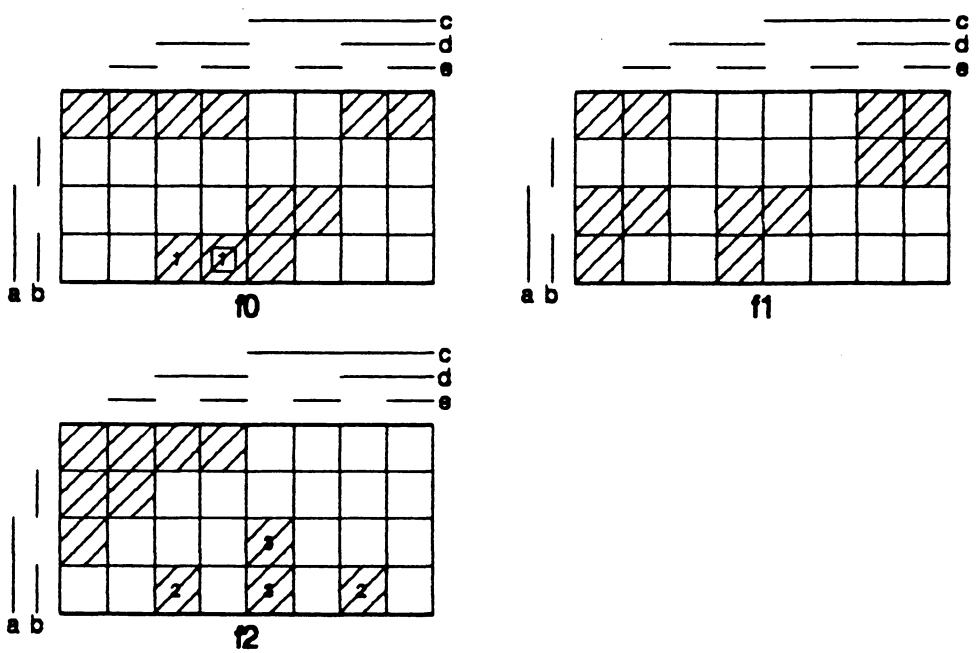


Figure 23. Cover after the fourth iteration

State 27 is a reference state.

K-cell $ab\bar{c}d$ eliminates $ab\bar{c}de$ from all the outputs.

K-cell $ab\bar{c}d$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 26, 27.

b. **Output 1.**

1) There are no more free states in this output.

c. **Output 2.**

1) State 30:

State 30 is a reference state.

None of the k-cells covering it can be eliminated yet.

2) State 26:

State 26 is a reference state.

K-cell $abd\bar{e}$ eliminates $ab\bar{c}d\bar{e}$ from all the outputs.

K-cell $abd\bar{e}$ is now a unique cover. Therefore it can be selected as part of the minimum cover.

States Covered: 26, 30.

3) State 28:

Attention Patron:

Page 97 is missing from
all copies

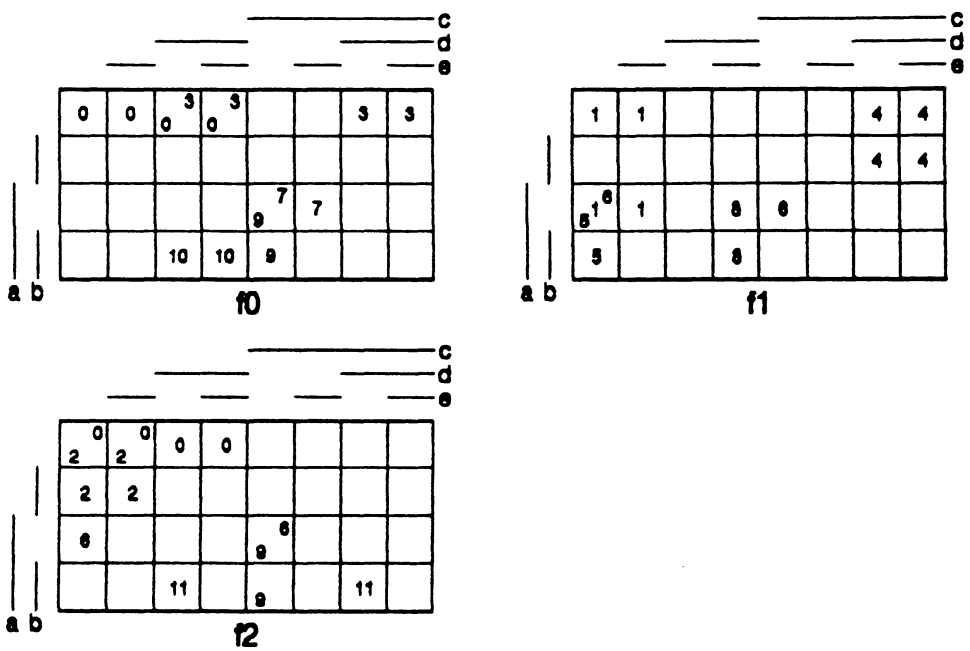


Figure 24. Absolute minimum cover

Chapter 5

Results and Conclusions

The new implementation of the Svoboda-Nadler-Vora algorithm (SNV) addresses several limitations inherent to exact logic minimization. The main objective of the techniques introduced in Chapter 3 is to reduce the cost of the generation of prime implicants without compromising the rigor of the method. The concept of weight introduced by Svoboda in the original algorithm proves to be very useful in achieving that goal.

Contrary to such traditional methods as Quine-McCluskey's, the SNV algorithm does not blindly generate all possible prime implicants. Instead, it begins with the highest possible k-cell within an output, and continues generating lower order k-cells until all the free states are bound. This guarantees the primality of the cover, because for a given state a k-cell is generated only if the state is not maximally covered by a higher order k-cell. The new implementation goes a step

Attention Patron:

Page 100 is missing from
all copies

Attention Patron:

Page 101 is missing from
all copies

Table 1. Comparison of the cost of prime implicant generation.

Function	in	out	terms	min	free	QM	Vora	New
a	4	3	9	7	13	143	135	30
c	5	3	6	6	11	176	174	18
p1	4	3	9	7	17	187	172	31
p2	5	3	20	12	34	884	832	72
br52	6	1	12	6	45	2835	1250	30
f0adr4	8	1	120	15	120	29640	8624	15
sadr	8	5	23	10	55	5115	4820	154
adr4	8	5	256	75	632	161160	(151490)	10925

computed by observing that for the other multiple-output functions the k-cell count is around 94% of the QM's count.[10]

The data for QM considers that for each free state in the function all the possible k-cells are generated, beginning with 0-cells and incrementing k until it reaches the highest value possible (as computed by Vora's program). The data for the new implementation was obtained directly from the program by placing a counter in the points where the k-cells are generated.

In terms of efficiency SNV cannot compete with ESPRESSO-II since it is bound by the complexity of the exact minimization process. For some small problems the new implementation of SNV found the minimum cover faster than ESPRESSO-II. However, as soon as the problems get moderately large, the time spend by SNV grows significantly. For example, the 5-input 3-output function given in chapter 4 was minimized by the SNV program in 82 seconds, while it took ESPRESSO-II 93 seconds to find the solution.[11] On the other hand, for ADR4, an 8-input 5-output PLA taken from [4], ESPRESSO-II minimized the 256 terms to 75 in 24 seconds. The SNV program took 1969 seconds.

---

[10] A more extensive comparison was not possible because Vora's program runs out of memory in a VAX 11/785 when minimizing anything larger than a 5-input 3-output function. Note that the memory requirements are substantially reduced in the new implementation by a complete redesign of the data structures. The new program was tested with functions of up to 12 inputs and 7 outputs in a MS-DOS machine.

[11] Both programs are coded in C, and were run under MS-DOS in a 10 Mhz AT-class computer.

The example given in chapter 4, shows a case where ESPRESSO-II fails to find the absolute minimum form. ESPRESSO-II has three minimization objectives: The number of product terms in the cover, the number of literals (1's and 0's) in the input part of the cover and the number of literals (1's) in the output part. The cover found by ESPRESSO-II has 49 input literals, 17 output literals and 12 cubes. The minimum cover found by the SNV algorithm has 43 input literals, 15 output literals and twelve cubes. Since ESPRESSO-II uses a PLA-area cost function, the same function was selected for the SNV program. The minimum cover uses 7.7% less PLA area than the solution found by ESPRESSO-II.

In the example, the first expansion produced a near minimal solution, which remained basically unchanged throughout the minimization. This reinforces the importance of the order in which the cubes are expanded, and the dependence of EXPAND on that order. Note that ESPRESSO-II was not able to get away from a locally minimal solution. This is probably due to the cube ordering dependence of REDUCE and to the merge with single cube containment strategy (explained in section 2.1.1.5) used in the implementation of the unate recursive paradigm. This strategy cannot detect more complex interaction within a set of cubes.

While the new implementation of the Svoboda-Nadler-Vora algorithm still suffers from the high complexity of exact logic minimization, it provides an improved prime implicant generation process. One issue that could be improved in a future work is the test of existence of a k-cell. This test is in the inner loop of the prime

implicant generation procedure and still takes a significant percentage of the time spent at that stage. This test is done by searching the states covered by the new k-cell to see if they exist. A possibility to be explored is the use of the complement of the function (as ESPRESSO-II does) to check if a new k-cell is indeed an implicant of the function. That is, it may be faster to check if the new k-cell covers a 0-state. Of course, this process should maintain the guarantee of an absolute minimum cover.

Appendix A

Program listing

A.1 Header files

```
/*** mm.h ***/

/*************************************************************************
* mm.exe 1.2
***************
*
*    Input :    File with the truth table of a multi-output
*               boolean function in the Berkeley's VLSI CAD Tools
*               standard format for PLA description.
*
*    Output:    Absolute minimum cover(s) for the given function
*               found by the Svoboda-Nadler-Vora algorithm.  By default
*               the output goes to stdout, using the same format as the
*               input.
*               Since the algorithm is not function-implementation
*               dependent, any cost function could be used.  The two
*               functions available in this release are:
*                    - PLA area
*                    - Gate/input count
*
*    Usage:          mm [options] < input file >
*
*    Options:
*               ?    Print a help screen in stdout.
*               -a   give ALL possible minimum forms.
*               -c[N]  Specify cut-off value N for the cost func.
*               -g   Use gate/input count cost function.
*               -h   Same as "?".
*               -k[N]  Specify stack size factor (N*10).
*               -o[fn] Optional output file name.
*               -p   Use PLA area cost function.
```

```
*           -s      Print an execution summary.
*           -t      Print an execution trace.
*
*     Author:       Cesar Augusto Duenas M.
*     Supervisor:   Dr. Morton Nadler
*
*                   Virginia Tech,  September 1989.
*
*************************************************************************/


/*** Compiler and Machine Dependencies ***/

#ifdef  MSDOS
#define ANSIC
#endif

#define INTS    (8 * sizeof(int))   /* # of bits per integer */
#if (INTS = = 16)
#define LOG2_INTS   4               /* log(INTS) ---- base 2 */
#else
#define LOG2_INTS   5
#endif

/** Register assignments **/

#define REG1 register
#define REG2 register

#ifdef MSDOS
#define REG3
#define REG4
#define REG5
#define REG6
#else
#if (defined(VAX) || defined (ULTRIX) || defined (VMS))
```

```
#define REG3 register
#define REG4 register
#define REG5 register
#define REG6 register
#endif
#endif


/*** Message defines ***/


#define COPYRIGHT \
    "MM Version 1.2, Multiple Output Boolean Function Minimization\n\n"
#define MSG_USAGE \
    "Usage:\tmm -[chko(g|p)(s|t)] file\n\tmm ? for help"
#define MSG_BADCHR "mm:\tInvalid character in truth table"
#define MSG_CANTOP "mm:\tCannot open input file"
#define MSG_CHKC   "\nmm:\tCut-off must be a positive integer"
#define MSG_MISSDAT "mm:\tMissing data in input file:"
#define MSG_NOMEM  "mm:\tMemory request failed"
#define MSG_NOPLA  "mm:\tThere is no data to minimize in file "
#define MSG_STK    "mm:\tOut of stack space. Use option k to expand it"


/*** Constants ***/


/** Boolean Constants **/
#define FALSE  0
#define TRUE   !FALSE


/** Codes for cost functions **/
#define NCOSTF 2
#define PLA    0
#define GATE   1


/** Names for major routines (used in profiling) **/
```

```
#define NFUNC     10
#define READ_TT    1
#define COUNT_N    2
#define ESSEN01    3
#define MAX_KCELL  4
#define MINIMIZE   5
#define BRANCH     6
#define CNT_NEIG   7
#define SEL_ELIM   8
#define IS_REF     9

/** Codes for the selection conditions word (selcond) **/
/*  CXi = = TRUE  = = >   k-cell ki satisfies condition X */
#define CA1 0x01
#define CB1 0x02
#define CC1 0x04
#define CA2 0x10
#define CB2 0x20
#define CC2 0x40


/***  Header files  ***/

#include  < stdio.h >
#include  < ctype.h >
#include  < string.h >

#ifdef ANSIC
#include  < stdlib.h >
#endif

#include "mm_bitv.h"

#ifdef ANSIC
#include "mm_prot.h"
```

```
#else
#include "mm_decl.h"
#endif


/*** Macros ***/

#define Error(M,L) (fprintf(stderr, "%s %s\n", M, L), exit (1))
#define BYTE(W) ((W) & 0xff)

#if (INTS == 16)
#define Ones(I) ( n_1s[BYTE(I)] + n_1s[BYTE(I >> 8)] )
#else
#define Ones(I) ( n_1s[BYTE(I)]      + n_1s[BYTE(I >> 8)] + \
            n_1s[BYTE(I >> 16)] + n_1s[BYTE(I >> 24)] )
#endif

/*** Global variables ***/

#ifndef EXT
#define EXT extern
#endif

EXT const char *fn_name[NFUNC]
#ifdef MAIN
 = { "Total Proc. Time", "Read Truth Table",
   "Count Neighbors ", "Essen 0/1-cells ",
   "Generate K-cells", "Minimize       ",
   "Branch         ", "Count Neighbors ",
   "Select/Eliminate", "Check Reference " }
#endif
;

EXT const long  exp2[]   /* Table: (2**0) ... (2**30) */
#ifdef MAIN
 = { 1L, 2L,      4L,      8L,      16L,      32L,      64L,
```

```
      128L,    256L,    512L,    1024L,    2048L,    4096L,
      8192L,   16384L,  32768L,  65536L,   131072L,  262144L,
      524288L, 1048576L, 2097152L, 4194304L, 8388608L, 116777216L,
      33554432L, 67108864L, 134217728L, 268435456L, 536870912L, 1073741824L }
#endif
;

EXT const int  n_1s[]   /* Table: Number of 1 bits in a byte */
#ifdef MAIN
 = { 0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4,1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,
   1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
   1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
   2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
   1,2,2,3,2,3,3,4,2,3,3,4,3,4,4,5,2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,
   2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
   2,3,3,4,3,4,4,5,3,4,4,5,4,5,5,6,3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,
   3,4,4,5,4,5,5,6,4,5,5,6,5,6,6,7,4,5,5,6,5,6,6,7,5,6,6,7,6,7,7,8 }
#endif
;

EXT union {
      struct {
         int   max_i,   /* Index for last word of bitv */
              *offi,    /* bit offset of each input    */
              *offn,    /* bit offset of each neighbor */
              *offo;    /* bit offset of each output   */
         bitv  input,   /* Mask: all bits of in field  */
               neigh,   /* Mask: idem. neighbor field  */
               out,     /* Mask: idem. outputs  field  */
               hout,    /* Mask: MSB for all outputs   */
               lout,    /* Mask: LSB for all outputs   */
              *imask,   /* Mask: for individual input  */
              *nmask,   /* Mask: for individual neighb */
              *omask;   /* Mask: for individual output */
         } mterm;       /* Descriptor for a bit vector */
      struct {
```

```
        int   max_i,    /* Index for last word of bitv */
            *offk,      /* bit offset of each input    */
            *offg,      /* bit offset of each neighbor */
            *offo;      /* bit offset of each output   */
        bitv  k,        /* Mask: all bits for k-index  */
              g,        /* Mask: all bits for g-index  */
              out,      /* Mask: idem. outputs  field  */
              hout,     /* Mask: MSB for all outputs   */
              lout,     /* Mask: LSB for all outputs   */
             *kmask,    /* Mask: for individual input  */
             *gmask,    /* Mask: for individual neighb */
             *omask;    /* Mask: for individual output */
        } kcell;        /* Descriptor for a bit vector */
    } masks;

#define Mterm masks.mterm
#define Kcell masks.kcell

EXT ttbl    tt;             /* Truth table  (state  list) */
EXT mform   mf,             /* Minimum form (k-cell list) */
            cv;             /* Current cover(k-cell list) */
EXT bitv    tmp_stack;      /* Bit Vectors for temp. use  */
EXT int     stacki,         /* Counter: used stack cells  */
            trace,          /* Flag for execution trace   */
            print_all,      /* Flag: print 1 or all m.f.  */
            verbose,        /* Flag: amount info printed  */
            debug,          /* Flag: print maximum info   */
            n_in,           /* No. of input   variables   */
            n_out,          /* No. of output  variables   */
            npterms,        /* No. of product terms read  */
            nbra,           /* No. of all branches taken  */
            nsol,           /* No. of minimal forms found */
            n_free,         /* No. of states yet to cover */
            Bits,           /* sizeof bit vector in bits  */
            Bytes,          /* sizeof bit vector in bytes */
            Ints,           /* sizeof bit vector in ints  */
```

```
        costf,          /* code for the cost function */
        fn_call[NFUNC],  /* No. of calls to each funct */
        return_branch,   /* TRUE: return from branch() */
        exceded_cost;    /* TRUE: current cost higher  */
EXT long    cut_off,        /* cut-off value for cost fn. */
        mf_cost,        /* Cost of the minimum form   */
        tot_cost,       /* Cost of current min. form  */
        last_cost,      /* Cost of last minimal form  */
        max_cost,       /* Current cost funct cut-off */
        terms,          /* No. of sel.  product terms */
        in_cost;        /* # of sel var (inv + n-inv) */
EXT char    name[50];       /* string w/ name of function */
EXT double  fn_time[NFUNC];  /* Accum. proc. time for func */
EXT FILE    *sol;           /* temp. file (for min forms) */
```

```
/*** mm_bitv.h ***/

/* Find position of a bit within a bit vector */
#define WORD(bit)   ((bit) > > LOG2_INTS)
#define BIT(bit)    (1 < < ((bit) & (INTS-1)))

/* # of ints needed to allocate a bit vector with "nbits" bits */
#define N_INT(nbits)    (((nbits) % INTS) ? WORD(nbits) + 1 : WORD(nbits))

/* bit vector type */
typedef unsigned *bitv;

/***
 *
 *      The two main data structures are the "state" and the "k_cell".
 *
 *      By a state, we mean a minterm of a boolean function, which can
 *  be 0-state, a 1-state or a D-state (don't care).  The binary index
 *  (index, for short) of a state is simply the binary representation
 *  of a minterm (as used in a truth table).
 *      Let s be the index of a given state. Let d(a,b) be the distance
 *  of two binary codes (# of bits where they differ).  Then, the set of
 *  "neighbors" of s is the set  N = { n, such that  d(s,n) = 1 }.
 *      The data structure that contains the information associated with
 *  each state is implemented as a struct type:
 */

typedef struct STATE {
        bitv    st,     /* Pointer to state data  (bit vector) */
                kc;     /* Array of ptr to k_cells covering st */
        int     nk,     /* Number of k_cells in array kc above */
                nn,     /* Number of neighbors for this state  */
                w;      /* Status of the state (FREE/DC/BOUND) */
        } state;

/*
```

```
*
*     A k-cell (binary cube) is a set of exp2(k) (2 to the power k)
*  states, such that each state in the set has exactly k neighbors.
*     Any k-cell can be represented by the ordered pair (g,k), where g
*  is the generic k-cell index (g-index), and k is the definite k-cell
*  index (k-index).  The g-index gives the input dependencies of the
*  k-cell, and is defined as:
*
*           g = ( g(n-1), ... , g(1), g(0) ) ,
*
*           g(i) =  0, if the k-cell does not depend on input i
*                   1, otherwise (var i is present in minterm)
*
*     The k-index indicates which input variables are complemented or
*  not, and it's defined as:
*
*           k = ( k(n-1), ... , k(1), k(0) ) ,
*
*           k(i) =  0,  if input variable i is complemented, or
*                       if the k-cell does not depend on this input.
*                   1,  otherwise (var is not complemented)
*
*
*     Let  f(A, B, C, D)  be a boolean function.  Using lowercase letters
*  to denote complement, we have:
*
*                   AbC  =  (1110, 1010) ,
*                   bd   =  (0101, 0000) ,  etc.
*
*     This representation of a k-cell is also implemented as a bit vector.
*
***
*
*  The fields in the bit vector for a state are:
*
*     | 2 * N_INT(nout) | N_INT(nvar) | N_INT(nvar) |
```

```
*     +-------------------------------------------------+
*     |     outputs     |  neighbors  |     index     |
*     +-------------------------------------------------+
*     |MSB                                          LSB|
*
*  where:
*        index       Binary index of the state
*        neighbors   Bit = 1 if the respective neighbor exists
*        outputs     For each output we have a two bit code:
*                      00  = >  zero state
*                      01  = >  don't care  (also covered free state)
*                      10  = >  one  state  (free)
*                      11  = >  bound state (we have covers for this one
*                                  but none has been selected yet)
*
***
*
*  The fields in the bit vector for a k-cell are:
*
*     | 2 * N_INT(nout) | N_INT(nvar) | N_INT(nvar) |
*     +-------------------------------------------------+
*     |     outputs     |  g-index    |   k-index     |
*     +-------------------------------------------------+
*     |MSB                                          LSB|
*
*  where:
*        k-index     Definite cell binary index
*        g-index     Generic  cell binary index
*        outputs     For each output we have a two bit code:
*                      x0  = >  Does not exist in this output.
*                      01  = >  Exists in this output.
*                      11  = >  Selected for minimum form.
*
***/

/*** Macros ***/
```

```
/* Memory Allocation */
#define New_cell()  (bitv) Malloc(Bytes)
#define New_state() New_cell()

/* Manipulation of individual bits */
#define Set_bit(v,b)    ((v)[WORD(b)] |=  BIT(b))
#define Clr_bit(v,b)    ((v)[WORD(b)] &= ~BIT(b))
#define Tst_bit(v,b)    ((v)[WORD(b)] &   BIT(b))

/* Status of a state */
#define FREE   0
#define DCARE -1
#define BOUND  1

/* Change status of a state in an output */
#define Zero(v,i)   {Clr_bit(v,Mterm.offo[i]); Clr_bit(v,Mterm.offo[i]+1);}
#define Free(v,i)   {Clr_bit(v,Mterm.offo[i]); Set_bit(v,Mterm.offo[i]+1);}
#define Dcare(v,i)  {Set_bit(v,Mterm.offo[i]); Clr_bit(v,Mterm.offo[i]+1);}
#define Bound(v,i)  {Set_bit(v,Mterm.offo[i]); Set_bit(v,Mterm.offo[i]+1);}

/* Test status of a state in an output */
#define isState(v,i) (Tst_bit(v,Mterm.offo[i])  || Tst_bit(v,Mterm.offo[i]+1))
#define isFree(v,i)  (!Tst_bit(v,Mterm.offo[i]) && Tst_bit(v,Mterm.offo[i]+1))
#define isDcare(v,i) (Tst_bit(v,Mterm.offo[i]) && !Tst_bit(v,Mterm.offo[i]+1))
#define isZero(v,i) !isState(v,i)

/* Copy g-index into k-index */
#define GtoK(u,v) \
 {REG1 I=n_in; for(;I<*Kcell.offo;++I) if(Tst_bit(v,I)) Set_bit(u,(I-n_in));}

/* Copy neigbor flags into s-index */
#define NtoS(u,v)   GtoK(u,v)
```

```
/* Change status of a k-cell in an output */
#define Select(v,i)     {REG1 I = Kcell.offo[i] + 1 ; Set_bit(v,I);}
#define DelKcell(v,i)   Clr_bit(v, Kcell.offo[i])

/* Test status of a k-cell in an output */
#define isSelect(v,i)   Tst_bit(v, Kcell.offo[i] + 1)
#define isKcell(v,i)    Tst_bit(v, Kcell.offo[i])

/* Test if a k-cell covers a state */
#define Covers(k,s)     !in_Comp(k, s)
#define NotCovers(k,s)  in_Comp(k, s)

/* Bit Vector wise operations */
#define BitV_Op(op) {REG1 I = Kcell.max_i; do (op); while (I-- > 0);}
#define Count1(v,n) {n=0 ; BitV_Op(n += Ones((v)[I]))}
#define Clear(v)     BitV_Op((v)[I] = 0)
#define Copy(u,v)    BitV_Op((u)[I] = (v)[I])
#define Not(u,v)     BitV_Op((u)[I] = ¬(v)[I])
#define And(u,v,w)   BitV_Op((u)[I] = (v)[I] & (w)[I])
#define Nand(u,v,w)  BitV_Op((u)[I] = ¬((v)[I] & (w)[I]))
#define Or(u,v,w)    BitV_Op((u)[I] = (v)[I] | (w)[I])
#define Nor(u,v,w)   BitV_Op((u)[I] = ¬((v)[I] | (w)[I]))
#define Xor(u,v,w)   BitV_Op((u)[I] = (v)[I] ≠ (w)[I])
#define Diff(u,v,w)  BitV_Op((u)[I] = (v)[I] & ¬(w)[I])
#define Outs(u,v)    BitV_Op((u)[I] = (v)[I] | ((v)[I]>>1))
#define Nempty(v,i)  BitV_Op((i)  |= (v)[I])
#define isSelAny(v,i) (i)=0; \
                BitV_Op((i)  |= (v)[I] & Kcell.hout[I])


/* Operations for a stack of temporary storage for bit vectors */
#define GetTemp(v)  {if(!--stacki)Error(MSG_STK,"");(v)=(tmp_stack += Ints);}
#define FreeTemp(v) (tmp_stack -= Ints, ++stacki)

/*** Type definitions ***/
```

```
typedef struct tbl  {
        int     size,   /* Allocated   # of states */
                ns,     /* Actual number of states */
               *nso,    /* # of states in each out */
               *ndo,    /* # of dcares in each out */
               *out;    /* Index to 1st st  in out */
        state  *tt;     /* Array of state pointers */
        } ttbl;         /* truth table data type   */

typedef struct minform {
        int     size,   /* Allocated   # of k-cells */
                nkc;    /* Actual number of k-cells */
        bitv    kc;     /* Array of k-cell pointers */
        } mform;        /* minimum form data type   */
```

```
/*** mm_decl.h ***/

/********** Function declarations for UNIX C **********/

#ifdef MAIN
  void  maximal_kcells (), minimize (),
        print_results (),  read_truth_table (),
        count_neighbors(), essential_0_1_cells (),
        usage (), parse_cmd_line ();
#else
  void  parse_input (),      minterms (),
        setup_descriptor (), profile (),
       *Malloc (),           *Calloc (),
        Mfree (),           prn_mem (),
        move_dcares (),      get0_cell ();
  int   is_in_out (),       Comp (),
        gk_Comp (),          in_Comp (),
        maximally_covers (), add_state (),
        add_to_CFS (),       add_to_state_covers (),
        add_to_min_form ();
#endif

  double proc_time ();
  void   profile   ();

#ifdef FMIN
  void  minimize (),
        check_corresponding (),
        store_min_form();
  int   intcmp (),
        select_or_eliminate(),
        is_reference();

 /* Function declarations for cost functions */

  void PLA_store_selection();
```

```
void gate_store_selection();

void  PLA_initialize();
void  gate_initialize();

#ifdef FIND
const void (*store_selection[])()
            = { PLA_store_selection,
              gate_store_selection };

const void (*initialize[])()
            = { PLA_initialize,
              gate_initialize };


void PLA_check_cost  ();
void gate_check_cost ();

const int (*check_cost[])()
            = { PLA_check_cost,
              gate_check_cost };


int PLA_check_coverage  ();
int gate_check_coverage ();

const int (*check_coverage[])()
            = { PLA_check_coverage,
              gate_check_coverage };


void PLA_branch  ();
void gate_branch ();

const void (*branch[])()
            = { PLA_branch,
```

```
        gate_branch };

#endif
#endif
```

```
/*** mm_prot.h ***/

/*********** Function prototypes for ANSI C ***********/

 double proc_time (void);
 void   profile  (double time, int funct);
 int    k_Comp   (bitv u,  bitv v);

#ifdef MAIN
 void  maximal_kcells     (void),
       minimize           (void),
       print_results      (void),
       read_truth_table   (char *fname),
       count_neighbors    (void),
       essential_0_1_cells (void),
       prn_mem            (void),
       usage              (void),
       parse_cmd_line  (int argc,char *argv[]);

 /* initialization functions */
 void  PLA_initialize  (void);
 void  gate_initialize (void);

 const void (*initialize[]) (void)
            = { PLA_initialize,
                gate_initialize };

#else
 void  parse_input       (FILE *in),
       minterms          (FILE *),
       setup_descriptor  (void),
       profile           (double time, int funct),
      *Malloc            (int size),
      *Calloc            (int num_elem, int elem_size, int *asize),
       Mfree             (char *block),
       move_dcares       (int out),
```

```
        get0_cell           (bitv kc, bitv st);
  int   is_in_out           (int i,   bitv v),
        Comp                (bitv u,  bitv v),
        in_Comp             (bitv u,  bitv v),
        gk_Comp             (bitv u,  bitv v),
        maximally_covers    (bitv kc, bitv s, int out, int *n),
        add_state           (bitv s),
        add_to_CFS          (int  ki, int out, bitv rs, int k),
        add_to_state_covers (int ki, state *s),
        add_to_min_form     (bitv kc);
#endif

#ifdef MINI

  void  minimize            (void),
        store_min_form      (void),
        check_corresponding
           (state *rs, int out, bitv k1, bitv k2, int *selcond);
  int   intcmp              (int *i1, int *i2),
        is_reference        (state *rs, int out, int last),
        select_or_eliminate (state *rs, int out, int *elimination);

 /* Function prototypes for cost functions */

#if MINI == 1

  void  PLA_select  (bitv k, int out, int dcare);
  void  gate_select (bitv k, int out, int dcare);

  const void (*select[NCOSTF]) (bitv k, int out, int dcare)
              = { PLA_select,
                  gate_select };

  int PLA_check_coverage  (int out, bitv k1, bitv k2, int *selcond);
  int gate_check_coverage (int out, bitv k1, bitv k2, int *selcond);
```

```
const int (*check_coverage[]) (int out, bitv k1, bitv k2, int *selcond)
            = { PLA_check_coverage,
              gate_check_coverage };

void PLA_check_cost  (int out, bitv k1, bitv k2, int *selcond);
void gate_check_cost (int out, bitv k1, bitv k2, int *selcond);

const void (*check_cost[]) (int out, bitv k1, bitv k2, int *selcond)
            = { PLA_check_cost,
              gate_check_cost };

void PLA_branch  (void);
void gate_branch (void);

const void (*branch[]) (void)
            = { PLA_branch,
              gate_branch };

#endif
#endif
```

A.2 Main functions

```
/*** mm.c ***/
/*************************************************************
 Functions in this file: main, parse_cmd_line, print_results.
 *************************************************************/

#define MAIN
#define EXT

#include "mm.h"

#define NOT_SEL -1

/*** Globals for this module ***/

static char fout[80],   /* Name of solution output file */
            fname[80]; /* string with input file name  */



/*****************************************************************************
 * main
 **********
 *
 *     Called by   :   none.
 *
 *     Calls       :   parse_cmd_line, read_truth_table,
 *                     initialize[costf], count_neigbors,
 *                     essential_0_1_cells, maximal_kcells,
 *                     minimize, print_results.
 *
 *     Modifies    :   none.
 *
```

```
*     Description :  Get options from the command line and call the
*                    I/O and minimization routines acordingly.
*
*     Returns     :  0  on success.
*
*************************************************************************/


main (argc, argv)

int argc;
char *argv[];

{

  /* Get options and file names from the command line */
   parse_cmd_line (argc, argv);

  /* Read data and prepare data structures */
   read_truth_table (fname);
   initialize[costf] ();
   count_neighbors ();

  /* Take care of some special cases */
   essential_0_1_cells ();

  /* Generation of multiple-output prime implicants */
   maximal_kcells ();

  /* Open a temporary file for storage of the solution */
   if (print_all && !(sol = tmpfile ()))
      Error ("mm: Cannot open work file","");

  /* Find all absolute minimum covers */
   minimize ();
```

```
    print_results ();

    /* Normal termination */
    if (trace)
     {
        int  i;
        for (i = 1 ; i < NFUNC ; *fn_time + = fn_time[i+ + ]);
        if (verbose)
           for (i = 1 ; i < NFUNC ; + + i)
              printf("# %s:\t %4.2f sec\t= > %4.2f\%\n", fn_name[i],
                       fn_time[i], 100 * (fn_time[i] / *fn_time));
        printf("# %s:\t %4.2f sec\n", *fn_name, *fn_time);
        (void) fflush(stdout);
     }
    if (print_all) fclose (sol);
    return 0;

}     /* end main */



/*****************************************************************************
*   parse_cmd_line
*********************
*
*     Called by   :   main.
*
*     Calls       :   usage.
*
*     Modifies    :   cut_off, fout, trace, fname.
*
*     Description :   Parses the command line, setting the option
*                     flags and file names as necessary.
*
*     Returns     :   A code for the cost function (defined in mm.h).
*
```

```
*************************************************************************/


void parse_cmd_line (argc, argv)

REG2 int   argc;   /* # of command line arguments */
REG3 char *argv[]; /* List of cmd. line arguments */

{
  REG1 char *arg;          /* Current argument */

  /* Display identification message */
  fputs (COPYRIGHT, stderr);
  cut_off = -1;
  costf   = NOT_SEL;

  /* Get options from command line */
  while (--argc > 0  &&  *(*++argv) == '-')
    for  (arg = ++*argv ; *arg ;)
      switch (*arg)
       {
         case '?':        /* Thou shalt help the needy ! */
         case 'h':
         case 'H':
                usage ();
                break;
         case 'a':        /* Give all possible solutions */
         case 'A':
                print_all = TRUE;
                ++arg;
                break;
         case 'c':        /* Cost function cut-off value */
         case 'C':
                if (arg[1])
                 {
                   if(!(cut_off = atol (arg + 1)))
```

```
            Error (MSG_USAGE, MSG_CHKC);
        }
        while (*++arg);
        break;
case 'd':       /* Debug: Print some more */
case 'D':       /*       info in stderr. */
        debug   = TRUE;
        trace   = TRUE;
        verbose = TRUE;
        ++arg;
        break;
case 'g':       /* Gate/input count cost function */
case 'G':
        if (costf == NOT_SEL)
          costf = GATE;
        else
          fputs ("Warning: option -g ignored\n",stderr);
        ++arg;
        break;
case 'k':       /* Modify stack size by factor N */
case 'K':
        if (arg[1])
        {
          if(!(stacki = atoi (arg + 1)))
            Error (MSG_USAGE, MSG_CHKC);
        }
        while (*++arg);
        break;
case 'o':       /* Output file name */
case 'O':
        strcpy (fout, (arg + 1));
        while (*++arg);
        break;
case 'p':       /* PLA area cost function */
case 'P':
        if (costf == NOT_SEL)
```

```
                costf = PLA;
            else
                fputs ("Warning: option -p ignored\n",stderr);
            ++arg;
            break;
        case 's':       /* Execution summary */
        case 'S':
            trace   = TRUE;
            verbose = FALSE;
            ++arg;
            break;
        case 't':       /* Execution trace */
        case 'T':
            trace   = TRUE;
            verbose = TRUE;
            ++arg;
            break;
        default :
            fprintf (stderr, "mm: Illegal option %c\n", *(arg++));
            Error   ("\n", MSG_USAGE);
    }

  /* Thou shalt help the needy ! */
   if (**argv == '?') usage();

  /* Get input file name from the command line */
   if (argc == 1) strcpy (fname, *argv);

  /* Set default options */
   if (costf == NOT_SEL) costf = PLA;

}    /* end parse_cmd_line */

/*****************************************************************************
```

```
* print_results
*********************
*
*     Called by  :  main.
*
*     Calls      :  none.
*
*     Modifies   :  none.
*
*     Description :  Writes all the minimal forms to the standard
*                    output, along with other information on the
*                    minimization.
*
*     Returns    :  none.
*
*****************************************************************************/


void print_results (void)


{
   REG1 bitv  kc;
   REG2 FILE *out;         /* output file handle  */
      int   nkc,         /* # k-cells in output */
            ns  =  0;     /* # of solutions      */


   GetTemp(kc);


  /* If no output file name given, use stdout */
   if (*fout) { if (!(out  =  fopen (fout, "w"))) Error (MSG_CANTOP,fout); }
   else              out  =  stdout;


   if (trace && verbose)
    {
      fputc('\n', out);
      fprintf (out, "#*****************     mm 1.2     *****************\n");
```

```
    fprintf (out, "\n# Multiple Output Minimization of %s\n", name);
    fprintf (out, "# Cost function in use:  ");
    switch (costf)
     {
       case PLA :  fputs ("PLA area\n", out);
                   break;
       case GATE:  fputs ("Gate-input count\n", out);
                   break;
     }
    if (cut_off > = 0)
       fprintf (out, "# Initial cut-off value:  %lu\n", cut_off);
    fprintf (out, "\n# No. of minterms generated:          %d\n", tt.ns);
    fprintf (out,  "# No. of k-cells  generated:          %d\n", cv.nkc);
    fprintf (out,  "# No. of leaves in minimization tree: %d\n", nbra);
    fprintf (out,  "# No. of solution leaves found:       %d\n", nsol);
    prn_mem ();
}

if (print_all)  /* Print all solutions as stored in sol */
 {
    rewind (sol);
    while ( + + ns  < = nsol)
     {
       fread (&nkc,  sizeof nkc,  1, sol);

       if (trace)
         fprintf(out,"\n# Minimal form %d  (COST = %lu)\n\n",ns,mf_cost);
       fprintf (out,".na %s\n.i %d\n.o %d\n.p %d\n",name,n_in,n_out,nkc);

       while (nkc--)
        {  REG1 int g = n_in, k = 0;

           if (fread (kc, sizeof(int) , Ints, sol) != Ints)
              if (feof (sol)) Error("mm: premature eof",  "@prnr");
              else            Error("mm: file read error", "@prnr");
```

```
            /* Translate the k and g indexes to binary representation
              and construct the input matrix for this k-cell */
            do
               fputc (Tst_bit(kc,g) ? (Tst_bit(kc,k) ? '1':'0') :'-', out);
            while (++g, ++k < n_in);
            fputc (' ', out);

            /* Construct the output vector for this k-cell */
            do
               fputc (Tst_bit(kc, g)  ? '1':'0', out);
            while ((g += 2) < Bits);
            fputc('\n', out);
          }
        fputs(".e\n", out);
      }
  }
else
  {
    if (trace)
       fprintf(out,"\n# Minimal form %d  (COST = %lu)\n\n",ns,mf_cost);
    fprintf (out,".na %s\n.i %d\n.o %d\n.p %d\n",name,n_in,n_out,mf.nkc);

    qsort(mf.kc, mf.nkc, Bytes, k_Comp);
    kc = mf.kc;
    while (mf.nkc--)
     { REG1 int g = n_in, k = 0;

       /* Translate the k and g indexes to binary representation
         and construct the input matrix for this k-cell */
       do
          fputc (Tst_bit (kc,g) ? (Tst_bit (kc,k) ? '1':'0') :'-', out);
       while (++g, ++k < n_in);
       fputc (' ', out);

       /* Construct the output vector for this k-cell;
        * consider only outputs where kc is "selected".
```

```
        */
        g++;
        do
            fputc (Tst_bit (kc,g)  ? '1':'0', out);
        while ((g += 2) < Bits);

        fputc('\n', out); kc += Ints;
      }
     fputs(".e\n", out);
   }
  if (*fout) fclose (out);

  FreeTemp(kc);

}     /* end print_results */
```

```
/*** mm_read.c ***/

/*********************************************************************
 Functions in this file: read_truth_table,  parse_input, minterms.
                         setup_descriptors, disjoint_tt.
 *********************************************************************/

#include "mm.h"


#define NTEMP 20     /* Number of temporary cells */

/*** Macros ***/

#define Get_cells(n)       (bitv) calloc ((n), Bytes)
#define Skip_spaces(c,fp)  while (isspace((c) = getc(fp)))
#define Echo(c, fi, fo) { \
   do putc((c), (fo)); while (((c) = getc(fi)) != EOF && (c) != '\n'); \
      putc('\n', (fo));                                          }

void disjoint_tt (void);

/*** Globals within this module ***/

static int head, line = 1;

/*************************************************************************
 * read_truth_table
 **********************
 *
 *     Called by   :   main.
 *
 *     Modifies    :   fname, tt.
 *
 *     Calls       :   proc_time, profile, parse_input, tt_new.
 *
```

```
*     Description :   Read the input file given in fname and parse
*                     it according to the Berkeley's VLSI CAD Tools
*                     standard format for PLA description.
*
*     Returns     :   none.
*
*************************************************************************/


void  read_truth_table (fname)

char *fname;

{
   FILE   *in;     /* input file handle */
   double  time;   /* execution time recorder */

  /* If an input file is specified open it, else, use stdin */
   if (*fname)
    { if (!(in = fopen (fname, "r"))) Error (MSG_CANTOP, fname); }
   else
      in = stdin;

  /* Initialize truth table data structure */
   tt.tt = (state *) Calloc (10, sizeof (state), &(tt.size));

   time = proc_time ();
   npterms = -1;          /* Check if this value is given */
   parse_input (in);
   fclose(in);

   if (trace)
    {
      profile (time, READ_TT);
      printf ("# inputs:\t%d\n# outputs:\t%d", n_in, n_out);
      if (npterms = = -1) puts ("\n");
```

```
      else              printf ("\n# terms:\t%d\n",npterms);
   }

}     /* end read_truth_table */



/*************************************************************************
 * parse_input
 ****************
 *
 *     Called by   :  read_truth_table.
 *
 *     Calls       :  setup_descriptors, minterms.
 *
 *     Modifies    :  n_in, n_out, npterms, name, tt.
 *
 *     Description :  Parses the truth table and associated data.
 *                    Sort the states in the truth table using in
 *                    ascending output order, don't cares first.
 *                    Finally, set the necessary indexes to access
 *                    the states efficiently.
 *
 *     Returns     :  none.
 *
 *************************************************************************/


void parse_input (in)

REG2 FILE *in;  /* input file handle */

{
  REG1 char ch;          /* Current input character  */
       char no_i = TRUE,  /* True while .i  not found */
            no_o = TRUE;  /* True while .o  not found */
```

```
REG2 int  i, j;
REG3 state *Ttt, *ttmp;


for (;;)
   switch (ch = getc (in))
    {
      case '.':             /* control ".parameters" */
              switch (ch = getc (in))
               {
                case 'i':  /* # of input variables */
                       if(no_i)
                        {
                          fscanf (in, "%d", &n_in);
                          no_i = FALSE;
                          if (!no_o) setup_descriptor();
                        }
                       else
                          fprintf (stderr,"mm: extra .i ignored");
                       break;
                case 'o':  /* # of output variables */
                       if(no_o)
                        {
                          fscanf (in, "%d", &n_out);
                          no_o = FALSE;
                          if (!no_i) setup_descriptor();
                        }
                       else
                         fprintf (stderr, "mm: Extra .o ignored");
                       break;
                case 'p':  /* # of produc terms */
                       fscanf(in, "%d", &npterms);
                       break;
                case 'n':  /* name of the function */
                       fscanf(in,"%*s");
                       while ((ch=getc(in)) != EOF &&
                                    ch != '\n' && isspace(ch));
```

```
                if(ch != EOF && ch != '\n')
                 {
                   REG1 char *na = name; *na++ = ch;
                   while ((ch=getc(in)) != EOF && ch != '\n')
                      *na++ = ch;
                   *na = '\0';
                 }
                ungetc(ch, in);
                break;
            case 'e':   /* end of data */
                goto End_of_Data;
            default:
                putc ('.', stdout);
                Echo (ch, in, stdout);
                line++;
          }
          break;
case  EOF:
          goto End_of_Data;
case '\n':              /* A blank line */
          line++;
case  ' ':              /* White space  */
case '\t':
case '\f':
case '\r':
          break;
case  '#':              /* echo comments */
          Echo (ch, in, stdout);
          line++;
          break;
default:
          ungetc(ch, in);
          if (no_i || no_o)
           {
             putc ('#', stdout);
             Echo (ch, in, stdout);
```

```
                line++;
                break;
              }
            minterms(in);
        }
  /*********************/
 /**/ End_of_Data: /**/
/*********************/
  /* Check if all data is present */
  if (no_i || no_o)  Error (MSG_MISSDAT,"");

  /* Sort the states in the truth table and set the necessary
    indexes to access them efficiently */
  Ttt = ttmp = (state *) Calloc (tt.ns, sizeof (state), &(tt.size));

  for (i = 0, j = head ; i < tt.ns ; ++i, j = (tt.tt + j)-> nk, ++Ttt)
     Ttt-> nk  = Ttt-> nn = 0, Ttt-> w = (tt.tt + j)-> w,
     memcpy (ttmp + i, tt.tt + j, 2 * sizeof(bitv));

  Mfree ((char *) tt.tt);
  tt.tt = ttmp;

  for (tt.out[0]= i= j= 0 ; ++i < n_out ; tt.out[i] = (j += tt.nso[i-1]));
  disjoint_tt ();

}     /* end parse_input */



/*****************************************************************************
 * setup_descriptor
 ************************
 *
 *     Called by  :  parse_input.
 *
 *     Calls      :  none.
```

```
*
*     Modifies   :  tt, masks, tmp_stacks.
*
*     Description :  Initializes the bit vector descriptor and some
*                    other variables.
*
*     Returns    :  none.
*
*************************************************************************/


void setup_descriptor (void)

{
   REG1 bitv m;
   REG2 int  i, j;
        int  ni2    = (2 * n_in),
             nfields = n_out + ni2;

   /* Initialize the stack size if not given in command line */
   if (stacki) stacki = NTEMP * stacki;
   else        stacki = NTEMP;

   /* Allocate arrays of indexes for the states within each output */
   tt.nso = (int *) calloc (3 * n_out, sizeof(int));
   tt.ndo = tt.nso + n_out;
   tt.out = tt.ndo + n_out;

   /* size bit vector in bits */
   Bits  = n_out + nfields;
   Bytes = (Ints = N_INT(Bits)) * sizeof (int);

   /* Compute the LSB position for each variable */
   Kcell.offk = (int *) calloc (nfields, sizeof(int));
   Kcell.offg = Kcell.offk + n_in;
   Kcell.offo = Kcell.offg + n_in;
```

```
    for (i = j = 0 ; i < nfields ; i++, j++)
        Kcell.offk[i] = (i < ni2) ? j : j++;
    Kcell.max_i = WORD(Kcell.offk[i - 1]);

/*-------------------------------------------------------------*
 * Generate masks to extract different parts of the bit vectors
 *-------------------------------------------------------------*/

    Kcell.kmask  = (bitv *) calloc (nfields, sizeof (bitv));
    Kcell.gmask  = Kcell.kmask + n_in;
    Kcell.omask  = Kcell.gmask + n_in;
    Kcell.k      = m  = Get_cells (5 + nfields + stacki);
    Kcell.g      = m += Ints;
    Kcell.out    = m += Ints;
    Kcell.hout   = m += Ints;
    Kcell.lout   = m += Ints;
   *Kcell.kmask  = m += Ints;

    for (i = 0 ; i < n_in ; Kcell.kmask[++i] = (m += Ints))
        Set_bit (m, i), Set_bit (Kcell.k, i);

    for (; i < ni2 ; Kcell.kmask[++i] = (m += Ints))
        Set_bit (m, i), Set_bit (Kcell.g, i);

    for (j = 0 ; i < Bits ; Kcell.omask[++j] = (m += Ints))
        Set_bit (m, i),         Clr_bit (Kcell.hout, i),
        Set_bit (Kcell.out, i), Set_bit (Kcell.lout, i), i++,
        Set_bit (m, i),         Set_bit (Kcell.hout, i),
        Set_bit (Kcell.out, i), Clr_bit (Kcell.lout, i), i++;

   /* The last stacki cells are for a stack of temporary storage */
        tmp_stack = m;

}       /* end setup_descriptor */
```

```
/*****************************************************************************
*  minterms
**************
*
*     Called by   :  parse_input.
*
*     Calls       :  add_state.
*
*     Modifies    :  none.
*
*     Description :  Expand the input don't cares in the truth
*                    table given in the input file to get the
*                    minterm representation of the function.
*
*     Returns     :  none.
*
*****************************************************************************/


void minterms (in)

FILE *in;  /* input file handle */

{
   REG1 int   i;
   REG2 char ch;
   REG3 bitv s;
      int  nd  =  0, not0  =  FALSE,
         *dc  =  (int *) calloc (n_in, sizeof (int));

   GetTemp (s); Clear (s);
  /* Get the inputs */
   for(i  =  0 ; i  <  n_in ; i++ )
      switch (ch  =  getc (in))
       {
```

```
        case '\n':              /* A blank line */
                line++;
        case ' ':               /* White space  */
        case '\t':
        case '\f':
        case '\r':
                i--;
                break;
        case '0':               /* Bit is clear already */
                break;
        case '1':               /*  Non-inverted input  */
                Set_bit (s,i);
                break;
        case '-':               /* Don't Care inputs */
        case 'x':               /* Get bit positions */
        case 'X':
                dc[nd++] = i;
                break;
        default:
                goto Bad_Char;
    }

/* Get the outputs */
 for(i = 0 ; i < n_out ; i++)
    switch (ch = getc(in))
    {
        case '\n':              /* A blank line */
                line++;
        case ' ':               /* White space  */
        case '\t':
        case '\f':
        case '\r':
                i--;
                break;
        case '0':               /* Bits are clear already */
                break;
```

```
        case '1':           /* Free state code = >  10 */
                Free(s,i);
                not0 = TRUE;
                break;
        case '-':           /* Don't care code = >  01 */
        case 'x':
        case 'X':
                Dcare(s,i);
                not0 = TRUE;
                break;
        default:
                goto Bad_Char;
    }

if (debug)
 {
   fprintf (stderr,"READ:\tLine %d, \tstate %X\n",line, *s);
   (void) fflush(stderr);
 }

/* Extract minterms */
if (not0)
   if (nd)
      for (i = (int) exp2[nd] ; i-- > 0 ;)
       { REG1 int j;
         for (j = 0; j < nd ; j++)
            if((i > > j) % 2)   Set_bit (s, dc[j]);
            else                Clr_bit (s, dc[j]);
         head = add_state(s);
       }
   else
      head = add_state(s);

free (dc); FreeTemp(s);
return;
```

```
 /*******************/
/**/ Bad_Char: /**/
/*******************/
   fprintf (stdout,"# Line %d ignored, \'%c\' unknown\n",line, ch);
   free (dc); FreeTemp(s);

}     /* end minterms */



/*****************************************************************************
* disjoint_tt
****************
*
*     Called by   :  parse_input.
*
*     Calls       :  none.
*
*     Modifies    :  tt.
*
*     Description :  Separates don't cares from free states in
*                    each output. The don't cares are moved to
*                    the beginning of the set of states for that
*                    output.
*
*     Returns     :  none.
*
*****************************************************************************/

void disjoint_tt (void)
{
   static state ss;
         int   s, fs, last, ndc, out;

   for (out = 0; out < n_out ; + + out)
    {
```

```
        last = tt.nso[out] + (fs = s = tt.out[out]);
        ndc  = fs + tt.ndo[out];
        for (; s < last ; ++s)
        {
          if ((tt.tt + s)-> w != DCARE) continue;
          /* Swap state s with first free state if it is not already there */
          if (s != fs)
          {
            memcpy (&ss,        (tt.tt + s),  sizeof (state));
            memcpy ((tt.tt + s), (tt.tt + fs), sizeof (state));
            memcpy ((tt.tt + fs), &ss,         sizeof (state));
          }
          if (++fs >= ndc) break;
        }
      }

}     /* end disjoint_tt */
```

```
/*** mm_cell.c ***/
/*****************************************************************
 Functions in this file: essential_0_1_cells, maximal_kcells,
                         maximally_covers,   get0_cell.
 *****************************************************************/

#include "mm.h"

extern const void (*select[NCOSTF])(bitv, int, int);

static double time;     /* execution time recorder */

/*****************************************************************************
 *  essential_0_1_cells
 ************************
 *
 *      Called by   :   main.
 *
 *      Calls       :   proc_time, profile, move_dcares, select[costf],
 *                      get0_cell, maximally_covers, add_to_min_form,
 *                      add_to_state_covers.
 *
 *      Modifies    :   cover, tt.
 *
 *      Description :   Initialize the minimum form data structure.
 *                      For all outputs,
 *                      if a free state rs has no neigbors, rs itself
 *                          is an essential 0-cell and it is selected
 *                          immediately, making rs into a don't care.
 *                      if rs has a unique neighbor ns, if the 1-cell
 *                          kc formed by both states maximally covers
 *                          rs, then kc is essential.  kc is selected
 *                          and rs is made a don't care.  Else, kc is
 *                          appended to the cover list for rs.
 *                          (if ns is free, it is processed as such.)
 *
```

```
*     Returns    :  none.
*
*************************************************************************/


void essential_0_1_cells ()

  {
   REG1   int    out;  /* index to current output */
   REG3   int    n;    /* index to neighbor of rs */
   REG4   int    r;    /* index to ref state (rs) */
   REG2   state *rs;   /* pointer to  ref.  state */
   REG5   bitv   kc,   /* pointer to 0- or 1-cell */
                 s;    /* Intermediate  results   */
   static bitv   inot; /* Used to mask-out inputs */
   static int    ki,   /* position of new  k-cell */
                 last; /* index to last st. in  i */

  /* Initialization */
   time = proc_time ();
   GetTemp(kc); GetTemp(s); GetTemp(inot);
   Not (inot, Mterm.input);

  /* Initialize Minimum Form and K-cell list data structure */
   cv.kc = (bitv) Calloc (10 * Ints, sizeof(int), &(cv.size));
   cv.size = (int) (cv.size / Ints); /* Scale for # of kcells */

  /* Get all free states in each output */
   for (out = 0 ; out < n_out ; out++)
    {
      last = tt.nso[out] + (r = tt.out[out]);
      for (r += tt.ndo[out] ; r < last ; ++r)
       { /* If state is Bound or Don't care, get next */
         if ((rs = tt.tt + r)-> w) continue;


         if (debug)
```

```
  {
    fprintf (stderr,"ESSN:\tState %d,\tNfree %d\n", r, n_free);
    (void) fflush(stderr);
  }

/* Process only states with 0 or 1 neighbors */
 switch ((tt.tt + r)-> nn)
  {
    case 0:
          get0_cell (kc, rs-> st);
          ki = add_to_min_form (kc);      /* save 0-cell */
          if (add_to_state_covers (ki, rs))
             add_to_CFS (ki, out, rs-> st, 0);
          select[costf] (&(unsigned)ki, out, FALSE);
          Dcare (rs-> st, out);
          rs-> w = DCARE;
          --n_free;
          break;
    case 1:
          /* 1-cell has all literals except the one
            corresponding to the conflicting input */
          Diff (kc, Kcell.g, rs-> st);
          GtoK (kc, kc); Or (s, rs-> st, inot);
          And  (kc, kc, s);              /* k = g AND s */

          if (maximally_covers(kc, rs-> st, out, &n))
           {
             ki = add_to_min_form (kc);  /* save 1-cell */
             if (add_to_state_covers (ki, rs))
                add_to_CFS (ki, out, rs-> st, 1);
             Dcare (rs-> st, out);
             rs-> w = DCARE;
             --n_free;
             if ((tt.tt + n)-> w != DCARE)
              {
                if (add_to_state_covers (ki, tt.tt + n))
```

```
                    add_to_CFS (ki,out,(tt.tt+n)->st,1);
                Dcare ((tt.tt + n)-> st, out);
                (tt.tt + n)-> w = DCARE;
                --n_free;
              }
            select[costf] (&(unsigned)ki, out, FALSE);
          }
        else
          {
            ki = add_to_min_form (kc);  /* save 1-cell */
            if (add_to_state_covers (ki, rs))
              add_to_CFS (ki, out, rs-> st, 1);
            if ((tt.tt + n)-> w != DCARE
              && add_to_state_covers (ki, tt.tt + n))
                add_to_CFS (ki,out,(tt.tt+n)->st,1);
            get0_cell (kc, rs-> st);
            ki = add_to_min_form (kc);   /* save 0-cell */
            if (add_to_state_covers (ki, rs))
              add_to_CFS (ki, out, rs-> st, 0);
            Bound (rs-> st, out); rs-> w = BOUND;
          }
        break;
      default: /* The states are sorted by # of neighbors;
                * so, there are no more "lone" states.
                */
        goto No_More;
    }
  }
 /*********************/
/**/  No_More:   /**/
/*********************/
 /* Sort out don't cares */
  move_dcares (out);
}

FreeTemp (inot); FreeTemp (s); FreeTemp (kc);
```

```
  if (trace)  profile (time, ESSEN01);

 }   /* end essential_0_1_cells */


/************************************************************************
 * maximal_kcells
 ***********************
 *
 *     Called by   :  main.
 *
 *     Calls       :  proc_time, profile,
 *                    move_dcares, select [costf].
 *
 *     Modifies    :  cover, tt.
 *
 *     Description :  Generates the multiple output maximal k-cells;
 *                    that is, the k-cells that maximally cover all
 *                    the free states in the multiple output function
 *                    sense.
 *                    If a free state has a unique cover, that k-cell
 *                    is an essential prime and therefore, selected
 *                    immediately.  All states covered by it are made
 *                    don't cares.
 *
 *     Returns     :  none.
 *
 ************************************************************************/

void maximal_kcells ()

{
  REG1  int   i;     /* index for general use            */
  REG2  long  notxst; /* TRUE: current kcell does not exist */
```

```
REG3   bitv   kc;    /* pointer to the current k-cell (kc) */
REG4   state *rs;    /* pointer to the reference state     */
REG5   bitv   s, u,  /* Bit vector for general use         */
              g;     /* Bit vector for g-index of a k-cell */
static bitv   inot,  /* Used to mask-out input part        */
              nnot;  /* Complement of neighbor flags       */
static int    last,  /* index to last state in output out  */
              nkc,   /* current num. of generated k-cells  */
              ng,    /* (# of neighbors of rs) - order     */
              nso,   /* Number of states in a given output */
              order, /* current order for the k-cells      */
             *gbit,  /* bits in g-index that can change    */
             *ghi,   /* positions of 1 bits in g-index     */
             *LNSO,  /* log2 (or # of states in an output) */
             *sstack; /* Stack:  states covered by a k-cell */
      int     out,   /* index to current output            */
              r,     /* index to reference state (rs)      */
              n,     /* index to neighbor of ref. state    */
              ki,    /* index to position of new k-cell    */
              bit,   /* bit being moved within g-index     */
             *ssp;   /* Stack pointer for stack of states  */
      long    nscov; /* num. of states covered by a k-cell */

time = proc_time ();
GetTemp(s); GetTemp(kc); GetTemp(inot);
GetTemp(g); GetTemp(u);  GetTemp(nnot);
Not (inot, Mterm.input);

/* Initialize arrays LNSO and g */
gbit = n_out + (LNSO = (int *) calloc (n_out + n_in, sizeof (int)));

/* Initialize weights in all bit positions of g-index to -1 */
ghi = memset ((int *) malloc(n_in * sizeof(int)),-1, n_in * sizeof(int));

for (nso = out = 0 ; out < n_out ; out++)
 {
```

```
    /* Get the maximum # of states in any output */
    if (nso < (i = tt.nso[out])) nso = i;
    /*
     * Upper bound on the order of k-cells for each output:
     *      LNSO[out] = int (log2 (# of states in out))
     */
    while (i > > = 1) + + LNSO[out];
  }

/* Allocate space for stack of states covered by a k-cell */
 sstack = (int *) calloc (nso, sizeof (int));

/*
 * Compute the maximal covers for all the free states.
 */
 for (out = 0 ; out < n_out ; out+ + )
  {
    last = tt.nso[out] + (r = tt.out[out]);
    for (r + = tt.ndo[out] ; r < last ; + + r)
     {
       /* If state already bound, get next */
       if ((rs = tt.tt + r)-> w) continue;
       /*
        * The highest order of a k-cell covering a state s
        * in output i is the minimum of LNSO and the number
        * of neighbors of s.
        */
        order = min (LNSO[out], rs-> nn);
       /*
        * If a neighbor of rs does not exist, its
        * corresponding bit in the g-index must always
        * be a 1 (the k-cell depends on that input).
        */
        Diff (nnot, Mterm.neigh, rs-> st);
       /*
        * ng: Number of bits that must be 1 in the g-index
```

```
*    besides the ones for non-existent neighbors.
*    Since the only bits that can be either 1 or 0
*    are the ones corresponding to neighbors:
*        ng  =  # neighbors - order.
*    (minus 1 because it is used as array index)
*/
ng = rs-> nn - order - 1;
/*
* Positions of bits in the g-index
* that correspond to neighbors of rs.
*/
for (i = *Kcell.offg, bit = 0 ; i < *Kcell.offo ; i++)
   if (Tst_bit (rs-> st, i))  gbit[bit++] = i;

if (debug)
 {
   fprintf (stderr,"MAXK:\tState %d,\tNfree %d\n", r, n_free);
   (void) fflush(stderr);
 }

do /* while (order >= 0) AND state not bounded */
 {
   nscov = exp2[order];
   for (bit = *ghi = 0 ; bit >= 0 ;)
    {
      if (order)
       { /* g-index */
         Copy (kc, nnot);
         for (i = bit ; i < ng ; i++)
            ghi[i + 1] = ghi[i] + 1;
         for (i = 0 ; i <= ng ; i++)
            if (ghi[i] >= 0)  Set_bit (kc, gbit[ghi[i]]);
        /* k-index = g AND s */
         GtoK(kc, kc); Or(s, rs-> st, inot); And(kc, kc, s);
       }
      else /* 0-cells are easier to generate */
```

```
{ /* g-index */
   Copy (kc, Kcell.g);
  /* k-index = s-index */
   And (s, rs-> st, Mterm.input); Or (kc, kc, s);
}
nkc = cv.nkc; ki = add_to_min_form (kc);
if (nkc != cv.nkc)
  /* If new k-cell, test its existence */
   switch (order)
    {
      case 0: /* 0-cell always exists and
               * it is a maximal cover
               */
              Outs (s, rs-> st); And (s, s, Mterm.lout);
              Or (cv.kc + ki, kc, s); /* Output of kc */
              if (add_to_state_covers (ki, rs))
                 add_to_CFS (ki,out,rs-> st,0);
              Bound (rs-> st, out); rs-> w = BOUND;
              goto GET_NEXT_G;
      case 1: /* 1-cell always exists, because the
               * generation of g-indexes is limited
               * by the existing neighbors of rs.
               */
              if (maximally_covers(kc, rs-> st, out, &n))
               { Bound (rs-> st, out); rs-> w = BOUND;}
              Copy (cv.kc + ki, kc);
              if (add_to_state_covers (ki, rs))
                 add_to_CFS (ki,out, rs-> st,1);
              if ((tt.tt + n)-> w != DCARE &&
                add_to_state_covers (ki, tt.tt + n))
                 add_to_CFS(ki,out,(tt.tt+n)-> st,1);
              goto GET_NEXT_G;
      default:
              notxst = nscov;
             /* Output of kc */
              Outs(u, rs-> st); And (u, u, Kcell.lout);
```

```
            /* g-index of kc */
            Clear (g); GtoK (g, kc);
            nso = tt.nso[out] + (i = tt.out[out]);
            /* Find 2**(order) states covered by kc */
            for (ssp=sstack ; notxst && i < nso ; ++i)
             {
               And (s, g, (tt.tt + i)->st);
               if (Covers (kc, s))
                { /* (k = g AND s) => covered! */
                  Outs (s, (tt.tt + i)->st);
                  And  (u, u, s);
                  *ssp++ = i;  /* Push state */
                  --notxst;
                }
             }
            /* u has the intersection of the output
             * part of all states covered by kc
             */
            Or (kc, kc, u); Copy (cv.kc + ki, kc);
    }
else
 { /* If k-cell is not new, use it as stored */
   notxst = FALSE; ssp = sstack;
   Copy (kc, cv.kc + ki);
 }
if (notxst)
   --cv.nkc;
else
 {
   REG1 int *sp = ssp;
  /* if kc exists, store only if maximal cover.
   *  A k-cell kc maximally covers a state s  iff
   *  kc exists in all the outputs where s exists.
   */
   Outs (s, rs-> st); And (s, s, Kcell.lout);
   Diff (u, s, kc);   Count1 (u,i);
```

```
            if (!i) { Bound (rs-> st, out); rs-> w = BOUND; }

            /* Add kc to the cover list of free states it covers */
            if (add_to_state_covers (ki, rs))
               add_to_CFS (ki, out, rs-> st, 1);
            while (sp-- > sstack)
               if ((tt.tt + *sp)-> w != DCARE
                 && add_to_state_covers (ki, tt.tt + *sp))
                  add_to_CFS (ki,out,(tt.tt + *sp)-> st, 1);
          }
         /**************************/
        /**/   GET_NEXT_G:    /**/
       /**************************/
         /* Prepare for next g-index if necessary */
         for (bit=ng ; bit >= 0 && ++ghi[bit] > order+bit ; --bit);
       }
     } /* while (order >= 0) AND state not bounded */
    while (ng++, order-- && !rs-> w);

    if (rs-> nk == 1)
     {
       /* If rs has a unique cover, that k-cell is
        * an essential prime and it can be selected
        * immediately.
        */
       select[costf] (rs->kc, out, FALSE);
       while (ssp-- > sstack)
          if ((tt.tt + *ssp)-> w != DCARE)
           {
             Dcare ((tt.tt + *ssp)-> st, out);
             (tt.tt + *ssp)-> w = DCARE;
             --n_free;
           }
     }
  }
/* Sort out don't cares */
```

```
      move_dcares (out);
    }

  free (LNSO); free (sstack);
  FreeTemp(s); FreeTemp(kc); FreeTemp(inot);
  FreeTemp(g); FreeTemp(u);  FreeTemp(nnot);

  if (trace)  profile (time, MAX_KCELL);

}     /* end maximal_kcells */



/*****************************************************************************
*  maximally_covers
*********************
*
*     Called by   :  essential_0_1_cells, find_max_kcells.
*
*     Calls       :  none.
*
*     Modifies    :  none.
*
*     Description :  Get the index of the neighbor of s,  and set the
*                    output part of the k-cell to the intersection of
*                    the output parts of s and its neighbor.
*                    Then, the maximal cover test simply checks if
*                    there is an output where s exists and the k-cell
*                    does not; i.e., it takes the set difference of the
*                    output parts of s and the k-cell. If the result is
*                    not empty, then s is not maximally covered.
*
*     Returns     :  TRUE,  if kc maximally covers s
*                    FALSE, otherwise.
*                    The index of the neighbor state of s (in n).
*
```

```
**************************************************************************/

int maximally_covers (kc, s, out, n)

bitv kc, s;
int  out, *n;

  {
   REG1 int i;
   REG2 bitv u, v;

   GetTemp(u); GetTemp(v);

  /* Get the s-index of the neighbor of s */
  /* s-index(v) = s-index(s) XOR NOT g-index(kc) */
   Diff (v, Kcell.g, kc); NtoS (v, v);
   Xor (v, v, s);

  /* Get a pointer to the neigbor state */
   for (*n = tt.out[out] ; in_Comp ((tt.tt + *n)-> st, v) ; ++*n);

  /* the 1-cell exists in the outputs that both states exist */
   Outs (v, s);                    /* output part of ref. st. */
   And  (v, v, Kcell.lout);
   Outs (u, (tt.tt + *n)-> st);       /* output part of neighbor */
   And  (u, u, v);                  /* intersection of outputs */
   Or   (kc, kc, u);                /* Save output part of kc  */

  /* A k-cell kc maximally covers a state s  iff
     kc exists in all the outputs where s exists */
   Diff   (u, v, kc);
   Count1 (u,i);

   FreeTemp(v); FreeTemp(u);
```

```
  /* (i != 0)  = > there is an output where s exists and kc does not */
   return !i;

  }   /* end maximally_covers */



/*************************************************************************
*  get0_cell
*************
*
*     Called by   :  essential_0_1_cells.
*
*     Calls       :  none.
*
*     Modifies    :  none.
*
*     Description :  Compute k-, g-index, and output of a 0-cell
*
*     Returns     :  none.
*
*************************************************************************/

void get0_cell (kc, st)
REG1 bitv kc, st;

 {
   REG2 bitv s;
   GetTemp(s);

   Copy (kc, Kcell.g);          /* g-index */

   And (s,  st, Mterm.input);
   Or  (kc, kc, s);             /* k-index  =  s-index */

   Outs (s, st);
```

```
    And (s, s, Mterm.lout);
    Or  (kc, kc, s);            /* Output part */

    FreeTemp (s);

}   /* end get0_cell */
```

```
/*** mm_mini.cc ***/
/*********************************************************
Functions in this file: minimize,     select_or_eliminate,
                        is_reference, check_corresponding,
                        store_min_form.
*********************************************************/

#define MINI 1

#include "mm.h"

 static int level = 0;      /* TRUE: first execution of minimize */
 static int dc    = FALSE;  /* TRUE: new don't cares generated   */

/*****************************************************************************
 *  minimize
 **********************
 *
 *     Called by   :   main, branch.
 *
 *     Calls       :   is_reference,   select_or_eliminate,
 *                     store_min_form, branch.
 *
 *     Modifies    :   return_branch, nbra, max_cost.
 *
 *     Description :   For each non-dominant  (reference)  free state,
 *               .     rules 1 and 2 are applied on its covers, trying
 *                     to eliminate alternate k-cells or select it when
 *                     only one is left. This continues until all states
 *                     are covered or when a set of k-cells covering a
 *                     state cannot be reduced to a unique cover.
 *                     In the first case, the algorithm terminates and
 *                     the solution is stored.
 *                     In the second case, rule 3 is applied to build a
 *                     solution tree. Each branch is followed until its
 *                     cost excedes the current minimum cost.
```

```
*                   If it doesn't, it is a minimum form.
*
*     Returns     :  none.
*
*****************************************************************************/


void minimize ()


{
  REG1  int    out, r;
        int    elimination,        /* TRUE: > = 1 k-cell eliminated */
               last;               /* index to last state in out   */
        long   kcell_selected = 0; /* Number of selected  kcells   */
  REG2  state *rs;                 /* prospective reference state  */
  static double time;              /* execution time recorder      */

  if (!level) { time = proc_time (); }

  do  /* while (kcell_selected && !exceded_cost */
   {
    /* for each output out, for each state in out */
     for (out = 0, kcell_selected = 0 ; out < n_out ; out++)
      {
        last = tt.nso[out] + (r = tt.out[out]);
        for (r += tt.ndo[out] ; r < last ; ++r)
         {
           rs = tt.tt + r;
          /* if not a reference state, try next rs */
           if (rs-> w == DCARE || !is_reference (rs,out,last))
              continue;
           if (debug)
            {
              fprintf (stderr,"MINI:\tState %d,\tNfree %d\n", r, n_free);
              (void) fflush(stderr);
            }
```

```
            /*
             * Look for a cover for rs in out while
             *   the current cutoff cost value is not exceded, and
             *   at least one alternate cover has been eliminated.
             */
            do
             {
               /* if a kcell was selected, increment count and end loop */
                if (select_or_eliminate (rs, out, &elimination))
                 { kcell_selected++; dc = TRUE; break; }
             }
            while (!exceded_cost && elimination);
          }
         /* Sort out don't cares */
          if (dc) { move_dcares (out); dc = FALSE; }
      }
  }
 while (!exceded_cost && kcell_selected);

/* if cutoff cost value has been exeded, finish branch */
 if (exceded_cost)
  {
    if (trace & verbose) fprintf(stderr, "e ");
    nbra++;
    return;
  }

/* if not all states are covered, apply branching rule */
 return_branch = FALSE;
 if (n_free)  { ++level; branch[costf] (); --level; }

/* if a new minimum cost has been found, update cut-off value */
 if (!exceded_cost)
  {
    if (tot_cost < max_cost)  max_cost = tot_cost;
```

```
      if (!return_branch)       store_min_form ();
    }

  if (trace && !level)
   { fputc ('\n', stderr); profile (time, MINIMIZE); }

}     /* end minimize */



/*****************************************************************************
*  is_reference
*******************
*
*     Called by   :  minimize.
*
*     Calls       :  none.
*
*     Modifies    :  tt, n_free.
*
*     Description :  Checks if the current state rs  is a reference
*                    state, that is,  if it does not dominate other
*                    free state(s).
*                    A free state S1  dominates another state S2 if
*                    the set of k-cells covering S2  is a subset of
*                    the set of k-cells covering S1, within a given
*                    output.
*                    Since a dominating state will be automatically
*                    covered when a k-cell is selected to cover its
*                    dominated state, it can be safely converted to
*                    a don't care.
*
*     Returns     :  TRUE,  if rs is a reference state.
*                    FALSE, if rs dominates another free state.
*
*****************************************************************************/
```

```
int is_reference (rs, out, last)

state *rs;   /* current state */
int   out,   /* output for rs */
      last;  /* index to last state in out */

{
  state  *ds;
  int     dom,            /* TRUE: rs is dominant (! ref) */
          d;
  unsigned nkr   = rs-> nk;
  bitv    kr,             /* k-cells covering rs */
          kr0   = rs-> kc,
          krlast = kr0 + nkr,
          kd,             /* k-cells covering ds */
          kdlast,
          v;

  GetTemp(v);

  for (d = tt.out[out] + tt.ndo[out] ; d < last ; ++d)
  {
    dom = FALSE;
    if ((ds = tt.tt + d)-> w == DCARE || ds == rs) continue;
    for (kr = kr0 ; kr != krlast ; kr++)
    { /* For each cover kr of state rs ... */
      Clear (v); GtoK (v, (cv.kc + *kr)); And (v, v, (tt.tt + d)-> st);
      if (Covers ((cv.kc + *kr), v))
      { /* If kr covers ds ... */
        dom = TRUE;
        /* Test if any of the covers of ds is not a cover of rs */
        for (kdlast = (kd = ds-> kc) + ds-> nk ; kd != kdlast ; kd++)
          if (!lfind (kd, kr0, &nkr, sizeof(int), intcmp))
          { dom = FALSE; break; }
```

```
            if (dom) goto GotIt; /* rs dominates ds  */
            else     break;      /* check next state */
          }
        }
    }
  /**************/
 /**/ GotIt: /**/
/**************/
   FreeTemp (v);

   if (dom)
    { /* If rs is a dominating state, make it a don't care */
      Dcare (rs-> st, out); rs-> w = DCARE; dc = TRUE; --n_free;
      return FALSE;
    }
   else    /* It is a reference state */
      return TRUE;

}     /* end  is_reference */



/*****************************************************************************
*   select_or_eliminate
*************************
*
*     Called by   :   minimize.
*
*     Calls       :   check_cost[costf],     select[costf],
*                     check_coverage[costf], check_correspoding[costf].
*
*     Modifies    :   tt.
*
*     Description :   Applies rules 1 and 2 on a pair of k-cells.
*                     Rule 1 is satisfied when the reference state
*                     is covered by a single k-cell.  This k-cell
```

```
*                must be in the minimum form.
*                If this is not the case, attempt to eliminate
*                alternate covers by applying rule 2.
*                If the alternate k-cells are reduced to one
*                rule 1 applies as above.
*                If two or more k-cells cannot be eliminated
*                against each other, the branching rule (3)
*                must be applied.
*
*     Returns    :  TRUE,  if a k-cell was selected.
*                   FALSE, otherwise.
*
*************************************************************************/


int select_or_eliminate (rs, out, elimination)

state  *rs;          /* reference state  */
int     out,         /* current output   */
       *elimination; /* TRUE: elim. ki   */

{
  REG1 bitv k1, k2;          /* current and alternate kcells */
       bitv k0    = rs-> kc,
            klast = k0 + rs-> nk - 1;
       int  selcond,         /* selection conditions word    */
            iterate = TRUE;  /* TRUE: iterate one more time  */

  *elimination = FALSE;

   while (iterate)
      for (k1 = k0, iterate = FALSE ; k1 <= klast ; k1++)
       {
        /* If k1 has been deleted from this output, get next */
         if (!isKcell(cv.kc + *k1, out))
          { /* Delete it from state covers */
```

```
    do
      --rs-> nk, *k1 = *(klast--);
    while (!isKcell(cv.kc + *k1, out) && k1 < = klast);
    if (k1 > klast) break;
  }

/* if rs has a unique cover, select k1 (rule 1) */
 if (rs-> nk = = 1)
  { select[costf] (k1, out, TRUE); return TRUE; }

/* if more than one k-cell covers reference state (rule 2) */
 for (k2 = k0 ; k2 < = klast ;)
  {/* Only for k2 other than k1 */
    if (!isKcell(cv.kc + *k2, out) || k2 = = k1)
       { k2 + +; continue; }

   /* Check k1 and k2 for condition a */
    check_coverage[costf] (out, k1, k2, &selcond);
    if (selcond & CA1)
     {
      /* Check k1 and k2 for condition b */
       check_cost[costf] (out, k1, k2, &selcond);
       if (selcond & CB1)
        {
          REG1 int yes;
         /* Check k1 and k2 for condition c */
          check_corresponding (rs, out, k1, k2, &selcond);
          if (selcond & CC1)
           {
             isSelAny (cv.kc + *k2, yes);
             if (yes)    /* Del k2 only in THIS output */
                DelKcell(cv.kc + *k2, out);
             else        /* Del k2 in ALL outputs */
                Diff (cv.kc + *k2, cv.kc + *k2, Kcell.out);
             iterate = *elimination = TRUE;
           }
```

```
        else
        {
          isSelAny (cv.kc + *k1, yes);
          if (yes)
          {
            /* Del k2 in THIS output */
            DelKcell(cv.kc + *k2, out);
            iterate = *elimination = TRUE;
          }
        }
      }
      k2++;
    }
  else if (selcond & CA2)
  {
    /* Check k1 and k2 for condition b */
    check_cost[costf] (out, k1, k2, &selcond);
    if (selcond & CB2)
    {
      REG1 int yes;
      /* Check k1 and k2 for condition c */
      check_corresponding (rs, out, k1, k2, &selcond);
      if (selcond & CC2)
      {
        isSelAny(cv.kc + *k1, yes);
        if (yes)    /* Del k1 only in THIS output */
          DelKcell(cv.kc + *k1, out);
        else        /* Del k1 in ALL outputs */
          Diff (cv.kc + *k1, cv.kc + *k1, Kcell.out);
        iterate = *elimination = TRUE;  k1 = k2;
      }
      else
      {
        isSelAny(cv.kc + *k2, yes);
        if (yes)
        {
```

```
                    /* Del k1 in THIS output */
                     DelKcell(cv.kc + *k1, out);
                     iterate = *elimination = TRUE; k1 = k2;
                   }
                 }
               }
              k2++;
            }
           else k2++;
       }
     }
 return FALSE;

}    /* end select_or_eliminate */



/*************************************************************************
*  check_corresponding
************************
*
*     Called by  :  minimize.
*
*     Calls      :  check_coverage[costf].
*
*     Modifies   :  none.
*
*     Description :  Checks if in an output i other than f (the
*                    current output), where a corresponding
*                    reference state rs' exists, condition 1a is
*                    satisfied by corresponding k-cells k1' and
*                    k2'in output i ( = > condition 1c satisfied).
*                    Condition 1a is satisfied by k1' k2' in i if:
*                    - both k-cells exist in i and kj' does not
*                      cover a free state not covered by ki' in i.
*                    - if ki does not exist in output i then kj
```

```
*                   either does not exist in i or it does not
*                   cover a free state in i.
*
*     Returns    :  none.
*
*************************************************************************/

#define CK1  0x1
#define CK2  0x2
#define CK12 (CK1 | CK2)

void check_corresponding (rs, out, k1, k2, selcond)

state *rs;     /* reference  state */
int   out;     /* current output  */
bitv  k1,      /* primary  kcell  */
      k2;      /* alternate kcell  */
int   *selcond; /* select cond word */

{
  REG1 int     i;
       int     ck = 0,      /* TRUE: Corresponding k-cell exist */
               sc = 0;      /* Local selection conditions word  */

  *selcond |= (CC1 | CC2);

  for (i = 0 ; i < n_out ; i++)
   {
    /* if corresponding ref. state doesn't exist in i, get next output */
     if (!isState(rs-> st, i)  ||  i == out) continue;

    /* check if corresponding k-cells k1'and k2' exist in output i */
    if (isKcell(cv.kc + *k1, i)) ck |= CK1;
    if (isKcell(cv.kc + *k2, i)) ck |= CK2;

    switch (ck)
```

```
    {
      case CK1: /* if only k1' exists and it covers at least 1 free
                   state in output i, k2 does not satisfy cond. c. */
                if (check_coverage[costf] (i, k1, k1, &sc))
                   *selcond &= ¬CC2;
                break;
      case CK2: /* if only k2' exists and it covers at least 1 free
                   state in output i, k1 does not satisfy cond. c. */
                if (check_coverage[costf] (i, k2, k2, &sc))
                   *selcond &= ¬CC1;
                break;
      case CK12: /* if both k-cells exist, check if kj' covers a
                    free state not covered by ki' in output i.     */
                check_coverage[costf] (i, k1, k2, &sc);
                *selcond &= (sc < < 2) | ¬(CC1 | CC2);
    }
  }

}    /* end check_corresponding */



/*************************************************************************
*  store_min_form
*********************
*
*     Called by   :   minimize.
*
*     Calls       :   none.
*
*     Modifies    :   sol, nsol, nbra, tot_cost, last_cost.
*
*     Description :   If the new solution costs less than the previous
*                     one, discards all solutions stored so far. Then,
*                     stores the new minimal form in a temporary file.
*                     RECORD FORMAT:
```

```
*                    | # k-cells | k-cell |  ...  | k-cell |
*
*     Returns     :  none.
*
*************************************************************************/


void store_min_form (void)

{
   REG1 int  i, yes;
   REG2 bitv sk, kc;

  if (trace && verbose) fprintf (stderr, "s ");
  nbra++;

 /* if new solution is less expensive, discard previous ones  */
  if (tot_cost < last_cost)
   { nsol = 0; if (print_all) rewind (sol); }


  if (print_all)                /* Store all solutions found */
   {
     /* save cost of the mininum form */
      mf_cost = tot_cost;

     /* write the number of k-cells in this solution          */
      fwrite (&terms, sizeof(int), 1, sol);

     /* Store bit vector for each k-cell in the minimum form  */
      for (i = 0, kc = cv.kc ; i < cv.nkc; i++, kc += Ints)
       {
         isSelAny (cv.kc + i, yes);
         if (yes)  fwrite (cv.kc + i, sizeof(int), Ints, sol);
       }
  }
```

```
  else if (nsol == 0)                /* Keep only one solution */
   {
     if (last_cost != -1) Mfree ((char *) mf.kc);
     mf_cost = tot_cost;   mf.nkc = (int) terms;
     mf.kc = sk = (bitv) Calloc (Ints * mf.nkc, sizeof(int), &(mf.size));
     mf.size = (int) (mf.size / Ints);   /* Scale for # kcells */
     for (i = 0, kc = cv.kc ; i < cv.nkc ; i++, kc += Ints)
      {
        isSelAny (kc, yes);
        if (yes) { Copy (sk, kc); sk += Ints; }
      }
   }

  /* update number of solutions and current minimum cost        */
   nsol++; last_cost = tot_cost;

}     /* end store_min_form */
```

A.3 Cost function dependant routines

```
/*** mm_pla.c ***/
/*************************************************************************
Functions in this file: PLA_initialize, PLA_check_cost, PLA_branch,
                        PLA_check_coverage, PLA_select.
*************************************************************************/

#define MINI 0

#include "mm.h"

extern void PLA_select (bitv, int, int);

/*** Globals ***/

bitv uvari,  /* Bit i=1 = > inverted     input i used */
     uvarn;  /* Bit i=1 = > non-inverted input i used */


/*************************************************************************
*  PLA_initialize
*********************
*
*     Called by  :  main.
*
*     Calls      :  none.
*
*     Modifies   :  tot_cost, terms, in_cost, nsol, nbra, maxcost.
*
*     Description :  initialize the cost counters and the cost
*                    cut-off value for the minimization;
*
```

```
*     Returns    :  none.
*
*************************************************************************/

void PLA_initialize ()

{
 /* Reset cost variables */
  last_cost = -1;
  tot_cost = terms = in_cost = nsol = nbra = 0;

 /* Set cut-off value if given in command line (if given) */
   max_cost = cut_off > 0  ? cut_off : *(exp2 + n_in) * (2 * n_in + n_out);

 /* Initialize selection flags for inverted and non-inverted inputs */
  uvari = Ints + (uvarn = (bitv) calloc (2, Bytes));

}     /* end PLA_initialize */



/*************************************************************************
*  PLA_check_cost
*******************************
*
*     Called by  :  select_or_eliminate, eliminate_by_c.
*
*     Calls      :  none.
*
*     Modifies   :  none.
*
*     Description :  Compute the cost of the unused literals in ki
*                    (inverted and non-inverted).
*                    Only if ki is not selected yet, set the cost
*                    of ki to the total cost with the k-cell, minus
*                    the total cost without the k-cell.
```

```
*                Check cost condition for k1 and k2, each being
*                used as primary k-cell in turn.
*
*     Returns    :  none.
*
*************************************************************************/


void PLA_check_cost (out, k1, k2, selcond)

int  out;  /* current output  */
bitv k1,   /* primary  kcell */
     k2;   /* alternate kcell */
int *selcond;

{
  int   i;
  bitv  kc1   = cv.kc + *k1, /* k-cell in position k1  */
        kc2   = cv.kc + *k2, /* l-cell in position k2  */
        v,                   /* bit vector for scratch  */
        nin1,                /* new inputs used in k1  */
        nin2;                /* new inputs used in k2  */
  long  cost1 = 0,           /* cost of k1 */
        cost2 = 0;           /* cost of k2 */

  GetTemp(v); GetTemp(nin1); GetTemp(nin2);

 /* Initialize assuming both k-cells satify condition */
  *selcond |= (CB1 | CB2);

 /* Input cost of k-cell =  #_of_ones (g & (k & ¬uvarn | ¬k & ¬uvari) */
  Diff  (v,   kc1, uvarn);  /*  k & ¬uvarn */
  Nor   (nin1, kc1, uvari);  /* ¬k & ¬uvari */
  Or    (v, v, nin1);
  Clear (nin1);
  GtoK  (nin1, kc1);         /* get g-index */
```

```
  And  (nin1, nin1, v);     /*  g & (...)  */

  Diff  (v,   kc2, uvarn);  /*  k & ¬uvarn */
  Nor   (nin2, kc2, uvari);  /*  ¬k & ¬uvari */
  Or    (v, v, nin2);
  Clear (nin2);
  GtoK  (nin2, kc2);        /* get g-index */
  And   (nin2, nin2, v);    /*  g & (...)  */

 /*
  * if kci selected, costi = 0
  * else,           costi = (cost with kci) - (cost without kci)
  */
  isSelAny (kc1, i);
  if (!i)  {Count1 (nin1, i); cost1 = (terms + 1) * i + in_cost + n_out;}

  isSelAny (kc2, i);
  if (!i)  {Count1 (nin2, i); cost2 = (terms + 1) * i + in_cost + n_out;}

 /*
  * Condition b for ki:
  *   kj is not of a lower cost than ki
  */
  if (cost1 > cost2) *selcond &= ¬CB1;
  if (cost2 > cost1) *selcond &= ¬CB2;

  FreeTemp (nin1); FreeTemp (nin2); FreeTemp (v);

}     /* end PLA_check_cost */



/*****************************************************************************
 *   PLA_check_coverage
 ********************************
 *
```

```
*     Called by   :  select_or_eliminate, eliminate_by_c.
*
*     Calls       :  none.
*
*     Modifies    :  none.
*
*     Description :  Check condition a for k1 and k2, each being
*                    used as primary k-cell in turn.
*                    Also, check if k1 covers at least one free
*                    state in output out.
*
*     Returns     :  TRUE, if k1 covers at least one free state.
*                    FALSE, otherwise.
*
*************************************************************************/


int PLA_check_coverage (out, k1, k2, selcond)

int  out;  /* current output  */
bitv k1,   /* primary   kcell */
     k2;   /* alternate kcell */
int *selcond;

{
   int   i,
         last  = tt.nso[out] + (i = tt.out[out]),
         cov_bound = FALSE;  /* k1 covers > = 1 free st. */
   bitv  kc1   = cv.kc + *k1, /* k-cell in position k1   */
         kc2   = cv.kc + *k2, /* l-cell in position k2   */
         v;                   /* bit vector for scratch  */
   state *s;

   GetTemp(v);

   /* Initialize assuming both k-cells satify conditions a and b */
```

```
  *selcond |= (CA1 | CA2);

 /*
 * Condition a for ki:
 *   kj doesn't cover a free state
 *   not covered by ki in output out
 */

 for (i += tt.ndo[out] ; i < last ; ++i)
  {
    if ((s = tt.tt + i)-> w != BOUND) continue;

    Clear (v); GtoK (v, kc1); And (v, v, s-> st);
    if (Covers (kc1, v))
     {
       cov_bound = TRUE;          /* k1 covers a bounded state */

      /* if k1 covers a free state not covered by k2 in out ... */
       Clear (v); GtoK (v, kc2); And (v, v, s-> st);
       if (NotCovers (kc2, v)) *selcond &= ¬CA2;
     }
    else
     { /* if k2 covers a free state not covered by k1 in out ... */
       Clear (v); GtoK (v, kc2); And (v, v, s-> st);
       if (Covers (v, kc2)) *selcond &= ¬CA1;
     }
  }
 FreeTemp (v);

 return  cov_bound;  /* TRUE: k1 covers at least one free state */

}    /* end PLA_check_coverage */



/*************************************************************************
```

```
*  PLA_select
*******************
*
*     Called by   :  select_or_eliminate, apply_rule_3.
*
*     Calls       :  none.
*
*     Modifies    :  cv, tt, uvarn, uvari, tot_cost, in_cost,
*                    terms, exceded_cost, n_free.
*
*     Description :  If k has not been selected in out, compute the
*                    cost of the literals in the k-cell which have
*                    not been selected as they appear in k (inverted
*                    or not), and update the total cost acordingly.
*                    Make don't cares all the free states covered by
*                    k in out, and marks the k-cell as selected.
*                    Finally, checks if the cost cut-off value has
*                    been exceded.
*
*     Returns     :  none.
*
*************************************************************************/


void PLA_select (k, out, dcare)

bitv k;     /* Selected k-cell */
int  out,   /* Current output  */
     dcare; /* TRUE: Make covered states don't cares */

{
   REG1 int    i;
   REG2 bitv   v, u, kc = cv.kc + *k;

   GetTemp (v); GetTemp (u);
```

```
/* if kcell not selected yet, update cost variables */
isSelAny (kc, i);
if (!i)
 {
   REG1 int ni, nn;
   Clear (u); GtoK (u, kc);
   /* uvarn = uvarn | (g & k & ¬uvarn)) */
   And (v, u, kc); Diff (v, v, uvarn); Or (uvarn, uvarn, v);
   Count1 (uvarn, nn);

   /* uvari = uvari | (g & ¬(k | uvari)) */
   Nor (v, kc, uvari); And (v, v, u); Or (uvari, uvari, v);
   Count1 (uvari, ni)

   tot_cost = ++terms * ((in_cost = nn + ni) + n_out);
}

if (dcare)
 { /* Make don't cares all the free states covered by k in output out */
   int   last = tt.nso[out] + (i = tt.out[out]);
   state *s;
   Clear (u); GtoK (u, kc);
   for (i += tt.ndo[out] ; i < last ; ++i)
    {
      s = tt.tt + i; And (v, u, s-> st);
      if (Covers (kc, v) && s-> w != DCARE)
       { Dcare (s-> st, out); s-> w = DCARE; --n_free; }
    }
 }

/* Mark the k-cell as selected */
Select (kc, out);

/* Checks if cut-off value has been exceded */
exceded_cost = tot_cost > max_cost;
```

```
    FreeTemp (v);  FreeTemp (u);

}     /* end PLA_select */



/*************************************************************************
*  PLA_branch
************************
*
*     Called by   :  minimize.
*
*     Calls       :  minimize, PLA_select, store_min_form.
*
*     Modifies    :  cv, tt.
*
*     Description :  The branching rule is applied when two or
*                    more k-cells covering a free state cannot
*                    be eliminated against each other.
*                    Save the state of the minimization at this
*                    point to allow baktracking.
*                    A state s with the less number of alternate
*                    covers is choosen as the next reference state.
*                    Select one of the k-cell covering s as part
*                    of the minimum form.
*                    Call minimize(). The minimization process then
*                    proceeds until a solution is found, or the cost
*                    cut-off is exceeded.
*                    This procedure constructs a solution tree with
*                    a branch for each of the k-cells covering s.
*
*     Returns     :  none.
*
*************************************************************************/

void PLA_branch ()
```

```
{
 REG1 int    i, j, out, ncov,
             Tn_free = n_free;
      int    last, dummy;
      state *rs, *s, *ss;
      ttbl   Ttt;
      mform  Tcv;
      bitv   st, kc, rs_kc, Tuvar;
      long   min_nkc   = exp2[n_in],
             Tterms    = terms,
             Tin_cost  = in_cost,
             Ttot_cost = tot_cost;

 memcpy (Tuvar = (bitv) malloc (2 * Bytes), uvarn, 2 * Bytes);

 /* Save status of the k-cell set */
 Tcv.kc = (bitv) Calloc (Ints * (Tcv.nkc=cv.nkc), sizeof(int), &(Tcv.size));
 Tcv.size = (int) (Tcv.size / Ints); /* Scale for # of kcells */
 memcpy (Tcv.kc, cv.kc, Tcv.nkc * Bytes);

 /* Save status of the set of free states */
 if (!(Ttt.ndo = (int *) malloc (sizeof (int) * n_out)))
    Error (MSG_NOMEM, "@branch");
 memcpy (Ttt.ndo, tt.ndo, n_out * sizeof(int));

 Ttt.tt = (state *) Calloc (Tn_free, sizeof (state), &dummy);
 st = (bitv) Calloc (Tn_free, Bytes, &dummy);
 for (j = 0, s = Ttt.tt ; j < n_out ; ++j)
  {
    last = tt.nso[j] + (i = tt.out[j]);
    for (i += tt.ndo[j] ; i <= last ; ++i, ++s, st+= Ints)
     {
       ss = tt.tt + i;

       /* Save state bit vector */
```

```
        Copy(st, ss->st); s-> st = st;

        /* Save # covers, status, # neighbors */
        memcpy (&s-> nk, &ss-> nk , 3 * sizeof(int));

        /* Save covers for this state */
        s-> kc = (bitv) Malloc (ss-> nk * sizeof(int));
        memcpy (s-> kc, ss-> kc , ss-> nk * sizeof(int));

        /* Update minimum # of covers if necessary */
        if (s-> nk < min_nkc)
          { min_nkc = s-> nk ; rs = s; out = j; }
      }
  }

/* Get list of k-cells covering selected reference state */
 rs_kc = kc = (bitv) Malloc (rs-> nk * sizeof(int));
 memcpy (rs_kc, rs-> kc , rs-> nk * sizeof(int));

/* Branch while cover of rs not unique */
 for (ncov = rs-> nk ; ncov-- ; + + kc)
  {
   /*
    * Put kc in the minimum form;
    * if cost cut-off not exceded, continue minimization
    */
   PLA_select (kc, out, TRUE); move_dcares (out);
   if (!exceded_cost)  minimize ();

   /* No more k-cells covering rs = > no more branches */
   if (!ncov) break;

   /* restore cost variables  and # of bound states */
   n_free  = Tn_free;  terms = Tterms;
   in_cost = Tin_cost; tot_cost = Ttot_cost;
   memcpy (uvarn,  Tuvar,  2 * Bytes);
```

```
    /* restore minimization status */
     memcpy (cv.kc,  Tcv.kc,  Tcv.nkc * Bytes);
     memcpy (tt.ndo, Ttt.ndo, n_out * sizeof(int));
     for (j = 0, s = Ttt.tt ; j < n_out ; ++j)
      {
        last = tt.nso[j] + (i = tt.out[j]);
        for (i += Ttt.ndo[j] ; i < last ; ++i, ++s, st+= Ints)
         {
           ss = tt.tt + i;
           Copy(ss-> st, s-> st);
           memcpy (&ss-> nk, &s-> nk , 3 * sizeof(int));
           memcpy (ss-> kc,  s-> kc , s-> nk * sizeof(int));

         }
      }
    }

  /* Deallocate temporary storage */
   for (i = 0; i < Tn_free ; Mfree ((char *) (Ttt.tt + i++)-> kc));
   Mfree ((char *) (Ttt.tt)-> st);  Mfree ((char *) Ttt.tt);
   Mfree ((char *) rs_kc); Mfree ((char *) Tcv.kc);
   free (Ttt.ndo); free (Tuvar);
   return_branch = TRUE;

}     /* end PLA_branch */
```

```
/*** mm_gate.c ***/
/*****************************************************************************
Functions in this file: gate_initialize, gate_check_cost, gate_branch,
                        gate_check_coverage, gate_select.
*****************************************************************************/

#define MINI 0

#include "mm.h"

extern void gate_select (bitv, int, int);



/*****************************************************************************
 *  gate_initialize
 ***********************
 *
 *      Called by   :   main.
 *
 *      Calls       :   none.
 *
 *      Modifies    :   tot_cost, terms, nsol, nbra, maxcost.
 *
 *      Description :   initialize the cost counters and the cost
 *                      cut-off value for the minimization;
 *
 *      Returns     :   none.
 *
 *****************************************************************************/

void gate_initialize ()

{
  /* Reset cost variables */
   last_cost = -1;
   tot_cost = terms = nsol = nbra = 0;
```

```
  /* Reset cut-off value */
   max_cost = cut_off > 0  ? cut_off : *(exp2 + n_in) * (2 * n_in + n_out);

}     /* end gate_initialize */



/*****************************************************************************
*  gate_check_cost
*********************************
*
*     Called by   :   select_or_eliminate, eliminate_by_c.
*
*     Calls       :   none.
*
*     Modifies    :   none.
*
*     Description :   The cost of ki is computed as the # of literals
*                     present in the k-cell, if not selected, and as
*                     zero otherwise.
*                     If ki has more than one literal present (AND-
*                     gate exists), and  either it does not cover all
*                     free states in out or at least one other k-cell
*                     has been selected in out (OR-gate exists), add
*                     1 to cost of ki.
*                     Check cost condition for k1 and k2, each being
*                     used as primary k-cell in turn.
*
*     Returns     :   none.
*
*****************************************************************************/


void gate_check_cost (out, k1, k2, selcond)
```

```
int     out;  /* current output  */
bitv    k1,   /* primary  kcell */
        k2;   /* alternate kcell */
int    *selcond;

{
  int   i, last,
        nlit1 = 0,          /* Number of literals present in k1  */
        nlit2 = 0,          /* Number of literals present in k2  */
        kcsel = FALSE,      /* TRUE:  > = 1 kcell selected in out */
        all1  = TRUE,       /* TRUE:  k1 covers all free states  */
        all2  = TRUE;       /* TRUE:  k2 covers all free states  */
  bitv  kc1   = cv.kc + *k1, /* k-cell in position k1   */
        kc2   = cv.kc + *k2, /* l-cell in position k2   */
        v;                /* bit vector for scratch  */
  long  cost1 = 0,          /* cost of k1 */
        cost2 = 0;          /* cost of k2 */
  state *s;

  GetTemp(v);

 /* Initialize assuming both k-cells satify the condition */
  *selcond |= (CB1 | CB2);

 /* ckeck if either k1 or k2 cover all free states in out */
  last  = tt.nso[out] + (i = tt.out[out]);
  for (i += tt.ndo[out] ; i < last ; ++i)
   {
     if ((s = tt.tt + i)-> w != BOUND) continue;
     if (all1)
      {
        Clear (v); GtoK (v, kc1); And (v, v, s-> st);
        if (NotCovers (kc1, v))
         { all1 = FALSE; if (!all2) break;}
      }
     if (all2)
```

```
    {
      Clear (v); GtoK (v, kc2); And (v, v, s-> st);
      if (NotCovers (kc2, v))
       { all2 = FALSE; if (!all1) break;}
    }
 }

/* find the cost of k1 and k2 in terms of # of inputs */
 And (v, Kcell.g, kc1); Count1(v, nlit1);
 And (v, Kcell.g, kc2); Count1(v, nlit2);

/* if kcell already selected cost = 0 ; else cost = # of inputs */
 isSelAny (kc1, i); if (!i) cost1 = nlit1;
 isSelAny (kc2, i); if (!i) cost2 = nlit2;

/* Check if at least one k-cell has been selected in output out */
 for (i = 0 ; i < cv.nkc ; ++i)
   if (isSelect(cv.kc + i, out)) { kcsel = TRUE; break; }

/*
 * ADD 1 for cost of OR-in/AND-out if
 *    exists AND-gate: # literals in ki > 1) AND
 *    exists OR-gate : # k-cells selected in output out > = 1, OR,
 *                  ki doesn't cover all free states in out.
 */
 if (nlit1 > 1 && (kcsel || !all1)) cost1++;
 if (nlit2 > 1 && (kcsel || !all2)) cost2++;

/*
 * Condition b for ki:
 *  kj is not of a lower cost than ki
 */
 if (cost1 > cost2) *selcond &= ~CB1;
 if (cost2 > cost1) *selcond &= ~CB2;

 FreeTemp (v);
```

```
}     /* end gate_check_cost */



/*************************************************************************
 *  gate_check_coverage
 ********************************
 *
 *     Called by   :  select_or_eliminate, eliminate_by_c.
 *
 *     Calls       :  none.
 *
 *     Modifies    :  none.
 *
 *     Description :  Check condition a for k1 and k2, each being
 *                    used as primary k-cell in turn.
 *                    Also, check if k1 covers at least one free
 *                    state in output out.
 *
 *     Returns     :  TRUE, if k1 covers at least one free state.
 *                    FALSE, otherwise.
 *
 *************************************************************************/


int gate_check_coverage (out, k1, k2, selcond)

int     out;  /* current output  */
bitv    k1,   /* primary  kcell  */
        k2;   /* alternate kcell */
int    *selcond;

{
  int   i, last,
        cov_bound = FALSE;   /* k1 covers at least 1 bound state */
```

```
bitv   kc1   = cv.kc + *k1, /* k-cell in position k1   */
       kc2   = cv.kc + *k2, /* l-cell in position k2   */
       v;                   /* bit vector for scratch  */
state *s;

GetTemp(v);

/* Initialize assuming both k-cells satify conditions a and b */
*selcond |= (CA1 | CA2);

/*
* Condition a for ki:
*   kj doesn't cover a free state
*   not covered by ki in output out
*/

last = tt.nso[out] + (i = tt.out[out]);
for (i += tt.ndo[out] ; i < last ; ++i)
 {
   if ((s = tt.tt + i)-> w != BOUND) continue;

   Clear (v); GtoK (v, kc1); And (v, v, s-> st);
   if (Covers (v, kc1))
    {
      cov_bound = TRUE;   /* k1 covers a bounded state */

      /* if k1 covers a free state not covered by k2 in out ... */
       Clear (v); GtoK (v, kc2); And (v, v, s-> st);
       if (NotCovers (v, kc2)) *selcond &= ¬CA2;
    }
   else
    { /* if k2 covers a free state not covered by k1 in f... */
       Clear (v); GtoK (v, kc2); And (v, v, s-> st);
       if (Covers (v, kc2)) *selcond &= ¬CA1;
    }
 }
```

```
    FreeTemp (v);

    return  cov_bound;  /* TRUE: k1 covers at least one free state */

}       /* end gate_check_coverage */



/*****************************************************************************
*   gate_select
*****************************
*
*       Called by   :   select_or_eliminate, apply_rule_3.
*
*       Calls       :   none.
*
*       Modifies    :   cv, tt, tot_cost, exceded_cost, n_free.
*
*       Description :   If k has not been selected in out, compute the
*                       cost as the number of literals and adds it to
*                       the total cost of the minimization.
*                       If k covers any free state in out, and if the
*                       OR-gate exists, add 1 to the total cost.
*                       Make don't cares all the free states covered by
*                       k in out, and marks the k-cell as selected.
*                       Finally, checks if the cost cut-off value has
*                       been exceded.
*
*       Returns     :   none.
*
*****************************************************************************/


void gate_select (k, out, dcare)
```

```
bitv k;     /* Selected k-cell  */
int  out,   /* Current output   */
     dcare; /* TRUE: Make covered states don't cares */

{
  int    i, last,
         still_free = TRUE,
         cov        = TRUE;
  bitv   v, kc = cv.kc + *k;
  state *s;

   GetTemp(v);

  /* if kcell not selected yet */
   isSelAny (kc, i);
   if (!i)
    { /* If exist AND gate, add its cost */
      And (v, Kcell.g, kc); Count1(v, i);
      if (i > 1) tot_cost + = i;
      + + terms;
    }

   if (dcare)
    { /* Make don't cares all the free states covered by k in output out */
      bitv u; GetTemp (u); Clear (u); GtoK (u, kc);
      last = tt.nso[out] + (i = tt.out[out]);
      for (i + = tt.ndo[out], cov = still_free = FALSE ; i < last ; + + i)
       {
         s = tt.tt + i;
         And (v, u, s-> st);
         if (Covers (v, kc)  &&  s-> w != DCARE)
          { Dcare (s-> st, out); s-> w  = DCARE; cov = TRUE; --n_free; }
         else
          {still_free = TRUE;}
       }
      FreeTemp (u);
```

```
    }

  /* If exists OR-gate, add 1 to the total cost */
   if (cov)
      if (still_free)
         tot_cost++;
      else for (i = 0 ; i < cv.nkc ; ++i)
         if (isSelect(cv.kc + i, out))
          { tot_cost++; break;}

  /* Mark the k-cell as selected */
   Select (kc, out);

  /* Checks if cut-off value has been exceded */
   exceded_cost  =  tot_cost > max_cost;

   FreeTemp(v);

}     /* end gate_select */



/*****************************************************************************
*  gate_branch
************************
*
*     Called by   :  minimize.
*
*     Calls       :  minimize, PLA_select, store_min_form.
*
*
*     Modifies    :  cv, tt.
*
*     Description :  The branching rule is applied when two or
*                    more k-cells covering a free state cannot
*                    be eliminated against each other.
```

```
*                Save the state of the minimization at this
*                point to allow baktracking.
*                A state s with the less number of alternate
*                covers is choosen as the next reference state.
*                Select one of the k-cell covering s as part
*                of the minimum form.
*                Call minimize(). The minimization process then
*                proceeds until a solution is found, or the cost
*                cut-off is exceeded.
*                This procedure constructs a solution tree with
*                a branch for each of the k-cells covering s.
*
*     Returns    :  none.
*
*****************************************************************************/


void gate_branch ()


{
  REG1 int    i, j, out, ncov,
              Tn_free = n_free;
       int    last, dummy;
       state *rs, *s, *ss;
       ttbl   Ttt;
       mform  Tcv;
       bitv   st, kc, rs_kc;
       long   min_nkc   = exp2[n_in],
              Ttot_cost = tot_cost;

  /* Save status of the k-cell set */
  Tcv.kc = (bitv) Calloc (Ints * (Tcv.nkc = cv.nkc), sizeof(int), &(Tcv.size));
  Tcv.size = (int) (Tcv.size / Ints); /* Scale for # of kcells */
  memcpy (Tcv.kc, cv.kc, Tcv.nkc * Bytes);


  /* Save status of the set of free states */
```

```
  if (!(Ttt.ndo = (int *) malloc (sizeof (int) * n_out)))
     Error (MSG_NOMEM, "@branch");
  memcpy (Ttt.ndo, tt.ndo, n_out * sizeof(int));

  Ttt.tt = (state *) Calloc (Tn_free, sizeof (state), &dummy);
  st = (bitv) Calloc (Tn_free, Bytes, &dummy);
  for (j = 0, s = Ttt.tt ; j < n_out ; ++j)
   {
     last = tt.nso[j] + (i = tt.out[j]);
     for (i += tt.ndo[j] ; i < last ; ++i, ++s, st+= Ints)
      {
        ss = tt.tt + i;

        /* Save state bit vector */
        Copy(st, ss->st); s-> st = st;

        /* Save # covers, status, # neighbors */
        memcpy (&s-> nk, &ss-> nk , 3 * sizeof(int));

        /* Save covers for this state */
        s-> kc = (bitv) Malloc (ss-> nk * sizeof(int));
        memcpy (s-> kc, ss-> kc , ss-> nk * sizeof(int));

        /* Update minimum # of covers if necessary */
        if (s-> nk < min_nkc)
         { min_nkc = s-> nk ; rs = s; out = j; }
      }
   }

/* Get list of k-cells covering selected reference state */
 rs_kc = kc = (bitv) Malloc (rs-> nk * sizeof(int));
 memcpy (rs_kc, rs-> kc , rs-> nk * sizeof(int));

/* Branch while cover of rs not unique */
 for (ncov = rs-> nk ; ncov-- ; ++kc)
  {
```

```
/*
 * Put kc in the minimum form;
 * if cost cut-off not exceded, continue minimization
 */
 gate_select (kc, out, TRUE); move_dcares (out);
 if (!exceded_cost)  minimize ();

/* No more k-cells covering rs = > no more branches */
 if (!ncov) break;

/* restore cost and # of bound states */
 tot_cost = Ttot_cost; n_free = Tn_free;

/* restore minimization status */
 memcpy (cv.kc,  Tcv.kc,  Tcv.nkc * Bytes);
 memcpy (tt.ndo, Ttt.ndo, n_out * sizeof(int));
 for (j = 0, s = Ttt.tt ; j < n_out ; + + j)
  {
    last = tt.nso[j] + (i = tt.out[j]);
    for (i + = Ttt.ndo[j] ; i < last ; + + i, + + s, st + = Ints)
     {
       ss = tt.tt + i;
       Copy(ss- > st, s- > st);
       memcpy (&ss- > nk, &s- > nk , 3 * sizeof(int));
       memcpy (ss- > kc,  s- > kc , s- > nk * sizeof(int));

     }
  }
}

/* Deallocate temporary storage */
 for (i = 0; i < Tn_free ; Mfree ((char *) (Ttt.tt + i+ + )- > kc));
 Mfree ((char *) (Ttt.tt)- > st);  Mfree ((char *) Ttt.tt);
 Mfree ((char *) rs_kc); Mfree ((char *) Tcv.kc);
 free (Ttt.ndo);
 return_branch = TRUE;
```

```
}     /* end gate_branch */
```

A.4 Support functions

```
/*** mm_bitv.c ***/

/*********************************************************************
 Functions in this file: add_state,  Comp,     add_to_state_covers,
                         add_to_CFS, gk_Comp,  add_to_min_form,
                         is_in_out,  in_Comp,  k_Comp, move_dcares.
 *********************************************************************/

#include "mm.h"

extern const void (*select[NCOSTF]) (bitv k, int out, int dcare);

/*****************************************************************************
 * add_state
 **************
 *
 *     Called by   :  minterms.
 *
 *     Calls       :  none.
 *
 *     Modifies    :  tt.
 *
 *     Description :  Add a new minterm to the truth table for each
 *                    output where state s exists.  Only states that
 *                    are not yet in the truth table are processed.
 *                    A linked list is temporarely implemented in the
 *                    array.  This speeds up the search for existing
 *                    states, and insertion of new ones.  When the
 *                    truth table is complete, it will be sorted and
 *                    used as an array.  The fields used by pointers
 *                    will be used to store other information.  The
```

```
*                position of the head of the list is passed in
*                field nk of the first (phisical) state.
*
*     Returns    :  none.
*
*************************************************************************/

int add_state (s)

bitv  s;

{
   REG1   int    ns   = tt.ns;  /* Current number of states in list  */
   REG2   int    out, status;
   static int    head = -1;     /* Index to the head of  state list  */
   static int    tail =  0;     /* Index to the tail of  state list  */
          state  *t   = tt.tt;  /* Array of states being constructed */
          bitv   so, ss;        /* State s masked for  single output */

   GetTemp(so); GetTemp(ss);

  /* Make new list the first time ...
     Note:  (t + out)-> nk  is used as the cursor for the linked list */
   if (head = = -1) head = t-> nk = 0;

  /* If no space for new state, allocate more */
   if ((ns + n_out + 1) > = tt.size)
    {
      REG1 int i; REG2 state *ttmp, *t1;
      ttmp = (state *) Calloc(tt.size+n_out,sizeof(state),&(tt.size));
      for (i = ns, t1 = ttmp ; i-- ; t++, t1++)
         memcpy (t1, t, sizeof (state));
      Mfree ((char *) tt.tt);
      t = ttmp;
    }
```

```
/* Generate one minterm for each output where s exists */
for (out = 0 ; out < n_out ; out++)
 {
   if (status = is_in_out (out,s))
    {
     /* Previous and next states in list (for new state) */
      REG1 int sp = head;  REG2 int sn = sp;
      REG3 state *ttns = (t + ns);
     /* Mask other outputs and set low bit of output out */
      And (so, Mterm.omask[out], Mterm.lout);
      And (ss, Mterm.input, s);
      Or  (so, so, ss);

     /* Since ns is the number of states, the address (tt + ns) is
        one past the last used (but allocated). So, it can be used
        as a sentinel.  [Tail].w  is already pointing to it */
      ttns-> kc = so; /* kc is used here for sorting purposes*/
      while (Comp ((t + sn)-> kc, so) < 0)
         sn = (t + (sp = sn))-> nk;

     /* Add the state only if it is not already in the list */
      if (sn = = ns)
       { /* If at sentinel it is either a new head or a new tail */
         ttns-> kc = New_state();  Copy (ttns-> kc, so);
         ttns-> st = New_state();  Copy (ttns-> st, s );
         ttns-> nn = out;
         if (sp = = sn)   /* New head */
            head = ns++, (t + head)-> nk = ns;
         else          /* New tail */
            tail = ns++, (t + tail)-> nk = ns;
         ++tt.nso[out];
         if (status > 0)     ttns-> w = FREE;
         else ++tt.ndo[out], ttns-> w = DCARE;
       }
      else if (Comp ((t + sn)-> kc, so))
       { /* if  so  is smaller */
```

```
            ttns-> kc = New_state(); Copy (ttns-> kc, so);
            ttns-> st = New_state(); Copy (ttns-> st, s );
            ttns-> nn = out;
            if (sp = = sn)  /* New head */
              (t + (head = ns))-> nk = sn;
            else
              (t + ((t + sp)-> nk = ns) )-> nk = sn;
            (t + tail)-> nk = + + ns;
            + + tt.nso[out];
            if (status > 0)    ttns-> w = FREE;
            else + + tt.ndo[out], ttns-> w = DCARE;
          }
        else
          { /* Union of the output part (1 prevails over DC) */
            REG1 i = n_out;
            Or (ss, (t + sn)-> st, s);
            while (i--) if (Tst_bit(ss, Mterm.offo[i]+1)) Free(ss,i);
            Copy ((t + sn)-> st, ss);
           /* If it is there, check if need to change DC -> 1 */
            if (isDcare((t + sn)-> st, out) && status > 0)
             { --tt.nso[out]; ttns-> w = FREE;}
          }
      }
  }
  tt.tt = t; tt.ns = ns;
  FreeTemp(so); FreeTemp(ss);

  return head;

}    /* end add_state */
```

```
/*************************************************************************
 * add_to_min_form
 *************
```

```
*
*     Called by   :  essential_0_1_cells.
*
*     Calls       :  none.
*
*     Modifies    :  cv.
*
*     Description :  Add a k-cell to the minimum form if its not
*                    already in the list.
*
*     Returns     :  the index of the new k-cell.
*
*****************************************************************************/

int add_to_min_form (k)

bitv  k;

{
  REG1 int  i;
       int  n = Ints * cv.nkc; /* # of k-cells in "ints" */

  /* If no space for new k-cell, allocate more */
  if ((cv.nkc + 1) > = cv.size)
   {
     REG2 bitv kctmp = (bitv) Calloc (n + Ints, sizeof(int), &(cv.size));
     cv.size = (int) (cv.size / Ints); /* Scale for # of kcells */
     memcpy (kctmp, cv.kc, n * sizeof(int));
     Mfree ((char *) cv.kc);
     cv.kc = kctmp;
   }

  /* Sentinel */
  Copy (cv.kc + n, k);
  for (i = 0 ; gk_Comp(cv.kc + i, k) ; i + = Ints);
```

```
  /* If "found" sentinel, k was not in list */
  if (i == n) ++cv.nkc;

  return i;      /* Index of k */

}     /* end add_to_min_form */



/*************************************************************************
 * add_to_state_covers
 **************
 *
 *     Called by  :  essential_0_1_cells.
 *
 *     Calls      :  none.
 *
 *     Modifies   :  tt.
 *
 *     Description :  Add a k-cell to the cover list of state s.
 *
 *     Returns    :  TRUE,  if the k-cell was added.
 *                   FALSE, otherwise.
 *
 *************************************************************************/

int add_to_state_covers (ki, s)

int   ki;
state *s;

{
  REG1 int  i,
           nk = s-> nk;
  REG2 bitv kc = s-> kc;
```

```
  if (nk % 2)
   {
     REG1 bitv k1, k2;
     k1 = k2 = (bitv) Malloc ((nk + 2) * sizeof(int));
     for (i = nk ; i-- ; *k2++ = *kc++);
     Mfree ((char *) s-> kc);
     s-> kc = kc = k1;
   }

  /* Sentinel */
   kc[nk] = ki;
   while (*kc++ != ki);

  /* If "found" sentinel, k was not in list */
   if (--kc == (s-> kc + nk)) {++s-> nk; return TRUE;}
   else                       return FALSE;

}    /* end add_to_state_covers */



/*****************************************************************************
* add_to_CFS  (Corresponding Free State)
**************
*
*     Called by  :  essential_0_1_cells.
*
*     Calls      :  none.
*
*     Modifies   :  tt.
*
*     Description :  Add a k-cell to the covers of the corresponding
*                    free states of rs.
*
*     Returns    :  none.
*
```

```
**************************************************************************/

int add_to_CFS (ki, out, rs, k)

int  ki,       /* index of the k-cell */
     out;      /* output of the rs    */
bitv rs;       /* reference state     */
int  k;        /* order of k-cell ki  */

{
   REG1 int  cs, i, dc = FALSE;
   REG2 bitv kc = cv.kc + ki;

   if (k)
    {
      for (i = 0; i < n_out ; i++)
        if (isKcell (kc, i)  &&  i != out  &&  isFree (rs, i))
         { /* if kc and CFS exist in  i !=  (output of rs) */
           cs = tt.out[i] + tt.ndo[i];
           while (in_Comp ((tt.tt + cs)-> st, rs)) ++cs;
           add_to_state_covers(ki, tt.tt + cs);
         }
    }
   else
     /* 0-cells are a special case */
     for (i = 0; i < n_out ; i++)
       if (isFree(rs, i) && i != out)
        {
          /* Search for the corresponding state */
          cs = tt.out[i];
          while (in_Comp ((tt.tt + cs)-> st, rs)) ++cs;
          if ((tt.tt + cs)-> nn)
              add_to_state_covers(ki, tt.tt + cs);
          else if ((tt.tt + cs)-> w != DCARE)
           { /* if it is also a "lone" free state, cover it */
             select[costf] (&(unsigned)ki, i, FALSE);
```

```
            Dcare ((tt.tt + cs)-> st, i);
            (tt.tt + cs)-> w = DCARE;
            --n_free ;dc = TRUE;
          }
        }
  return dc;

}    /* end add_to_CFS */


/*************************************************************************
 * Comp
 **********
 *
 *    Called by  :  many.
 *
 *    Calls      :  none.
 *
 *    Modifies   :  none.
 *
 *    Description :  Compare two bit vectors u and v.
 *
 *    Returns    :  -1  if  u < v
 *                   0  if  u = v
 *                   1  if  u > v
 *
 *************************************************************************/

int Comp (u, v)

REG2 bitv u, v;

{
  REG1 i = Kcell.max_i;
```

```
    do
        if      (u[i] < v[i])  return -1;
        else if (u[i] > v[i])  return  1;
    while (i-- > 0);

    return 0;

}       /* end Comp */



/*****************************************************************************
 * gk_Comp
 ************
 *
 *      Called by   :   Maximally Covers.
 *
 *      Calls       :   Comp.
 *
 *      Modifies    :   none.
 *
 *      Description :   Compare the 2*n_in LSB's of bit vectors u and v.
 *
 *      Returns     :   -1  if  u < v
 *                       0  if  u = v
 *                       1  if  u > v
 *
 *****************************************************************************/

int gk_Comp (u, v)

bitv u, v;

{
        int  result;
    REG1 bitv a, b;
```

```
    GetTemp(a); GetTemp(b);

    Diff (a, u, Mterm.out);
    Diff (b, v, Mterm.out);
    result = Comp (a, b);

    FreeTemp(a); FreeTemp(b);

    return result;

}    /* end gk_Comp */



/*****************************************************************************
* in_Comp
************
*
*     Called by  :  Maximally Covers.
*
*     Calls      :  Comp.
*
*     Modifies   :  none.
*
*     Description :  Compare the  n_in  LSB's of bit vectors u and v.
*
*     Returns    :  -1  if  u < v
*                    0  if  u = v
*                    1  if  u > v
*
*****************************************************************************/

int in_Comp (u, v)

bitv u, v;
```

```
{
    int  result;
  REG1 bitv a, b;

  GetTemp(a); GetTemp(b);

  And (a, u, Mterm.input);
  And (b, v, Mterm.input);
  result = Comp (a, b);

  FreeTemp(a); FreeTemp(b);

  return result;

}    /* end in_Comp */



/*****************************************************************************
* k_Comp
************
*
*    Called by  :  print_results.
*
*    Calls      :  Comp.
*
*    Modifies   :  none.
*
*    Description :  Compares the k-indexes of two k-cells.
*                   It is used to qsort the k-cells in
*                   ascending lexicographical order.
*
*    Returns    :  -1  if  u < v
*                   0  if  u = v
*                   1  if  u > v
```

```
*
**************************************************************************/

int k_Comp (u, v)

bitv u, v;

{
  REG1 int  i;

  for (i = 0 ; i < n_in ; + + i)
     if (Tst_bit (u, i)) {
          if (!Tst_bit (v, i)) return  1;
     }else if ( Tst_bit (v, i)) return -1;
  return 0;

}     /* end k_Comp */



/**************************************************************************
* is_in_out
**************
*
*     Called by   :   many.
*
*     Calls       :   none.
*
*     Modifies    :   none.
*
*     Description :   If state v exists in output identifies it either
*                     as a don't care or a free (1) state.
*
*     Returns     :    0,  if v does not exist  in out i  (code = 00)
*                     -1,  if v is a don't care in out i  (code = 01)
*                      1,  if v is free (a '1') in out i  (code = 10)
```

```
*
*************************************************************************/

int is_in_out (i, v)

REG1 int  i;
REG2 bitv v;

{
  /* Get the bit offset for output i */
   i = Mterm.offo[i];

   if (Tst_bit(v, i)) return -1;   /* (check x1) */
   i++;
   if (Tst_bit(v, i)) return  1;   /* (check 1x) */

   return 0;

}     /* end is_in_out */



/*************************************************************************
* move_dcares
***************
*
*     Called by  :  essential_0_1_cells.
*
*     Calls      :  none.
*
*     Modifies   :  tt.
*
*     Description :  Move the new don't care states to
*                    the don't care set for output out.
*
*     Returns    :  none.
```

```
*
*************************************************************************/

void move_dcares (out)

int out;

{
  static state ss;
       int  s, fs, last = tt.nso[out] + (s = tt.out[out]);

  for (fs = s += tt.ndo[out] ; s < last ; ++s)
   {
     if ((tt.tt + s)-> w != DCARE) continue;
    /* Swap state s with first free state if it is not already there */
     if (s != fs)
      {
        memcpy (&ss,       (tt.tt + s),  sizeof (state));
        memcpy ((tt.tt + s), (tt.tt + fs), sizeof (state));
        memcpy ((tt.tt + fs), &ss,        sizeof (state));
      }
     tt.ndo[out]++; fs++;
   }

}    /* end move_dcares */
```

```
/*** mm_util.c ***/

/*************************************************************
 Functions in this file: count_neighbors, intcmp, proc_time,
                         profile, usage.
 *************************************************************/

#include "mm.h"

#ifdef ANSIC
#include <time.h>
#endif

int nn_Comp ();



/*****************************************************************************
 * count_neighbors
 ********************
 *
 *     Called by   :   maximal_kcells.
 *
 *     Calls       :   sort_by_neighbors.
 *
 *     Modifies    :   tt, nfree.
 *
 *     Description :   Find the neighbors for each of the free states.
 *                     A bit is set for each existing neighbor (free
 *                     or don't care).  A weight is computed for each
 *                     state as the number of neighbors.
 *                     Sort states by ascending number of neighbors.
 *                     Compute the initial number of free states.
 *
 *     Returns     :   none.
 *
 *****************************************************************************/
```

```
void count_neighbors (void)

{
  REG1 int    out;        /* index to current output */
  REG2 state *rs;         /* pointer to reference st */
  REG3 state *ns;         /* pointer to neigh. of rs */
  REG4 bitv   s;
     int    r,          /* index to ref state (rs) */
            n,          /* index to neighbor of rs */
            last,       /* index to last st in out */
            lastdc;     /* index to last dc in out */
     double time;       /* execution time recorder */

  time = proc_time ();
  GetTemp (s);

  for (out = 0 ; out < n_out ; out++)
   {
     last = tt.nso[out] + (r = tt.out[out]);
     for (lastdc = r += tt.ndo[out] ; r < last ; ++r)
      {
        rs = tt.tt + (n = r);
       /* Any state has at most n_in neighbors */
         if (rs-> nn >= n_in) continue;

       /* Search for free neighbors of rs */
        while (++n < last)
         {
           REG1 int w, b, dist;
           ns = tt.tt + n;
          /* Get input distance of states rs and ns */
           Xor (s, rs-> st, ns-> st);
           And (s, s,  Mterm.input);
           Count1(s, dist);
           if (dist != 1) continue;
```

```
            for (w = 0 ; w < Ints ; ++w)
               if (s[w])
                { for (b = 0 ; exp2[b] < s[w] ; ++b); break;}

            /* If distance > 1 they are not neighbors */
            b += (n_in + w * INTS);
            /* Set the bit in both free states */
            Set_bit (rs-> st, b); ++rs-> nn;
            Set_bit (ns-> st, b); ++ns-> nn;
          }

        /* Search for don't care neighbors of rs */
        /* This part is of constant cost for all free states */
         for (n = tt.out[out] ; n < lastdc ; ++n)
          {
            REG1 int w, b, dist;
            ns = tt.tt + n;
            /* Get input distance of states rs and ns */
            Xor (s, rs-> st, ns-> st);
            And (s, s, Mterm.input);
            Count1(s, dist);
            if (dist != 1) continue;
            for (w = 0 ; w < Ints ; ++w)
               if (s[w])
                { for (b = 0 ; exp2[b] < s[w] ; ++b); break;}

            /* If distance > 1 they are not neighbors */
            b += (n_in + w * INTS);
            /* Set the bit in free state */
            Set_bit (rs-> st, b); ++rs-> nn;
          }

      }
  }
  FreeTemp(s);
```

```
  /* Sort free states in each output by ascending number of neighbors  */
   for (out  =  n_out ; out-- ;)
    {
      qsort ( tt.tt  +  tt.ndo[out] +  tt.out[out],  /* sort free states */
            tt.nso[out] - tt.ndo[out],          /* # of free states */
            sizeof (state),                    /* size of a state  */
            nn_Comp);                           /* Compare # neigh. */

      n_free  +  =  tt.nso[out] - tt.ndo[out];
    }

   if (trace)  profile (time, COUNT_N);

 }    /* end count_neighbors */



/*****************************************************************************
 *  intcmp
 ************
 *
 *      Called by   :   is_reference.
 *
 *      Calls       :   none.
 *
 *      Modifies    :   none.
 *
 *      Description :   Compares two integers pointed to by the
 *                      arguments.
 *
 *      Returns     :   0,      if the integers are identical
 *                      non-0,  otherwise.
 *
 *****************************************************************************/

int intcmp (i1, i2)
```

```
REG1 int *i1, *i2;

 { return  *i1 - *i2; }    /* end intcmp */



/*************************************************************************
 *  proc_time
 **************
 *
 *     Called by   :   many.
 *
 *     Calls       :   none.
 *
 *     Modifies    :   none.
 *
 *     Description :   Returns the elapsed processor time in seconds
 *                     since some constant reference. It takes care
 *                     of machine dependencies.
 *
 *     Returns     :   a double with the time in seconds.
 *
 *************************************************************************/

double proc_time(void)

 {
   double time;

#ifdef ANSIC
   /* ANSI C: MS-DOS */
   time = (float) clock() / CLK_TCK;
#else
#ifdef UNIX
   /* Berkeley Unix 4.1/4.2 bsd */
```

```
    struct tms {int user, sys, cuser, csys;} buffer;
    times(&buffer);
    time = buffer.user / 60.0;
#else
#ifdef VMS
  /* VAX/VMS */
    struct tms {int user, sys, cuser, csys;} buffer;
    times(&buffer);
    time = buffer.user / 100.0;
#else
  /* Other hosts */
    time = 0.0;
#endif
#endif
#endif

    return time;

}     /* end proc_time */



/*****************************************************************************
 * profile
 ************
 *
 *     Called by   :   many.
 *
 *     Calls       :   none.
 *
 *     Modifies    :   none.
 *
 *     Description :   Add time spent for the calling
 *                     function into the total.
 *                     Increments its counter of calls too.
 *
```

```
*      Returns    :  none.
*
*************************************************************************/

void profile (time, funct)

double time;
int    funct;

{
   time = proc_time() - time;
   fn_time[funct] + = time;
   fn_call[funct]+ +;
   if (debug)
    {
      fprintf(stderr, "# %s:\t %4.2f sec\n", fn_name[funct], time);
      (void) fflush(stderr);
    }

}     /* end profile */



/*************************************************************************
* usage
************
*
*      Called by  :  parse_cmd_line.
*
*      Modifies   :  none.
*
*      Calls      :  none.
*
*      Description :  Display help screen and exit.
*
*      Returns    :  0 on exit.
```

```
 *
 ***************************************************************************/

void usage (void)
{
   puts ( "\nSyntax:\t mm [options] < file > [.EXT]\n"                    );
   puts ( "options:\t [] = optional\t < > = required\t * = default\n"   );
   puts ( "\t-a\t  Print All possible solutions (default: only one)"   );
   puts ( "\t-c[N]\t  Specify cut-off value N for the cost function"   );
   puts ( "\t\t  If N is omitted, no cut-off value is used"           );
   puts ( "\t-g\t  Use gate/input count cost function"                 );
   puts ( "\t-h\t  Displays this help screen (also ?)"                );
   puts ( "\t-k[N]\t  Stack size factor (for error: out of stack ...)" );
   puts ( "\t-o < fn > \t  Optional output file name (default: stdout)"   );
   puts ( "\t-p\t* Use PLA area cost function"                         );
   puts ( "\t-s\t  Print execution summary"                           );
   puts ( "\t-t\t  Print execution trace\n\n"                         );
   puts ( "\tOptions can be concatenated.  However, any option"       );
   puts ( "\twith arguments must be the last one in a group."         );
   puts ( "\tFor example:\n\t\t\tmm  -gc350 -s adder.pla"             );
   exit (0);
}     /* end usage */



/***************************************************************************
 *  nn_Comp
 ************
 *
 *      Called by   :   count_neighbors.
 *
 *      Modifies    :   tt.
 *
 *      Calls       :   none.
 *
 *      Description :   Compare two states by the number of neigbors.
```

```
 *
 *     Returns    :   < 0, if u has less neighbors than v.
 *                    = 0, if u and v has equal number of neighbors.
 *                    > 0, if u has more neighbors than v.
 *
 ****************************************************************************/

int nn_Comp(u, v)

state * u, * v;

{ return u-> nn - v-> nn; }      /* end sort_by_neighbors */
```

```
/*** mm_mem.c ***/

/*********************************************************
Functions in this file: Malloc, Calloc, Mfree, prn_mem.
*********************************************************/


#include "mm.h"

#ifdef UNIX
#define malloc(n) sbrk((int) n)
#endif

#define BUCKETS 64


/*** Globals within this unit ***/

union header {
      int          bucket;   /* bucket id for this block  */
      union header *next;     /* Next block in this bucket */
      } *free_list[BUCKETS]; /* Hash table for memory blocks */

static int  n_block[BUCKETS],   /* # of allocated blocks in each bucket */
           a_block[BUCKETS],   /* # of active    blocks in each bucket */
           block_size;         /* Real block size (includes header) */


/*****************************************************************************
 *  Malloc
 ************
 *
 *     Called by   :   Many.
 *
 *     Calls       :   none.
 *
```

```
 *      Modifies   :  only globals in this module.
 *
 *      Description :   Provides an efficient memory management strategy
 *                      by using a modified exponential buddy system.
 *                      The modification is that it allows block sizes
 *                      h(k) = (2**k + 2**(k-1))/2, besides h(k) = 2**k.
 *                       The memory blocks that are made available after
 *                      use, are collected in free lists, one for each
 *                      block size. This provides a hashing scheme for
 *                      quick searching of a block of a given size.
 *                       The hash value for a memory block is, therefore,
 *                      a function of its size. It is computed only once,
 *                      (during allocation) and stored on a header field.
 *                       When a block is in a free list, the header field
 *                      is used for storing the pointer to the next block
 *                      in the list. (NOTE: Neither splitting nor merging
 *                      of blocks is performed at any time)
 *
 *      Returns    :  a pointer to the new block.
 *
 ************************************************************************/

void *Malloc (size)

int size;

 {
   REG3 int n;
   REG2 int i;
   REG1 int bucket;
   REG6 int tsize = size + sizeof(union header *);
   REG5 union header *b;

/*
 * The hash number of a memory block is given by bucket, such that
 *      h(bucket-1) < tsize < = h(bucket) ,  bucket = 0, 1, ... , 63.
```

```
*
* The block size, h(bucket), is computed as follows:
* Let k = smallest integer > = log2(tsize). Using the fact that
*
*    (2**(k-1) + 2**(k))/2 = (3 * 2**(k-1))/2 = 3 * 2**(k-2)
*
* Let n = (k-2);
*
*    if tsize < = 3 * 2**(n) :  bucket    = (2 * n)
*                               h(bucket) = 3 * 2**(n)
*
*    if tsize  > 3 * 2**(n) :  bucket    = 1 + (2 * n)
*                               h(bucket) = 2**(k) = 2**(n+2) = 4 * 2**(n)
*
* That is,   block_size = 4 * 2**((bucket/2)-2), bucket even;
*            block_size = 3 * 2**((bucket/2)-2), bucket odd.
*
*/

 /* Compute smallest n, such that  (n+2) > = log2(tsize)  */
 for (i = tsize - 1, n = -1 ; (i > > = 1) > 0 ; n++);

 /* Determine the bucket and actual block_size */
 if (tsize < = (3 < < n))
  {
    bucket     = (2 * n);   /* bucket = { 0, 2, 4, ... , 62 } */
    block_size = 3 < < n;   /* h(bucket) = 3 * 2**n */
  }
 else
  {
    bucket     = (2 * n) + 1;   /* bucket = { 1, 3, 5, ... , 63 } */
    block_size = 4 < < n;       /* h(bucket) = 4 * 2**n */
  }

 /* A new memory block is allocated only if no blocks exist in the
    the free list of this bucket */
```

```
    if (b = free_list[bucket])
        free_list[bucket] = b-> next;
    else
    { /* This free list is empty => allocate new block */
        b = (union header *) malloc((unsigned) block_size);
       /* If request failed, see if a larger block exists in free lists */
        if (!b)
        {
            for(i = bucket+1, bucket = 0; i < BUCKETS; i++)
                if (free_list[i])
                {
                    bucket = i;
                    b = free_list[bucket];
                    free_list[bucket] = b-> next;
                    break;
                }
            if (!bucket)
            { /* We are out of memory! */
                fprintf(stderr,"mm: failed allocating %d bytes\n", tsize);
                exit(-1);
            }
        }
        ++n_block[bucket];
    }
    ++a_block[bucket];
    b-> bucket = bucket | 0x5500;

    return (void *) (b + 1);

}     /* end Malloc */


/*****************************************************************************
 * Calloc
```

```
************
*
*     Called by   :  Many.
*
*     Calls       :  Malloc.
*
*     Modifies    :  only globals in this module.
*
*     Description :  Analogous to the standard calloc().
*                    Uses Malloc to allocate an array of num_elem
*                    items of elem_size bytes each.  The allocated
*                    block is NOT initialized.
*                    Returns a pointer to the allocated block and
*                    its actual size.
*
*     Returns     :  a pointer to the new block, and its size (asize)
*
*************************************************************************/


void *Calloc(num_elem, elem_size, asize)

int  num_elem,  /* Number of elements */
     elem_size, /* Size of an element */
    *asize;     /* Actual array size  */
{
    char *p = (char *) Malloc (num_elem * elem_size);

   /* Take off the size of the block's header */
    *asize = (int) ((block_size - sizeof(union header)) / elem_size);
    return (void *) p;

}     /* end Calloc */
```

```
/*************************************************************************
 * Mfree
 **************
 *
 *     Called by   :  Many.
 *
 *     Calls       :  none.
 *
 *     Modifies    :  only globals in this module.
 *
 *     Description :  Checks if the pointer of the returned block is
 *                    corrupted.  If not, links block to the free list
 *                    correponding to its size.
 *
 *     Returns     :  none.
 *
 *************************************************************************/

void Mfree (block)

char *block;

{
   REG1 union header *b = (union header *) block - 1;
   REG2 int bucket = b-> bucket;

   if (((bucket & 0xFF00) != 0x5500) || ((bucket &= 0x00FF) >= BUCKETS))
    {
      fprintf(stderr, "mm: corrupt pointer returned, addr = %x, bucket = %x\n",
            b, bucket);
      exit(-1);
    }
   /* Put block in free list for this bucket */
   b-> next = free_list[bucket];
   free_list[bucket] = b;
   --a_block[bucket];
```

```
}     /* end Mfree */


/*************************************************************************
*   prn_mem
***************
*
*     Called by   :   print_results.
*
*     Calls       :   none.
*
*     Modifies    :   none.
*
*     Description :   Print a report on memory usage statistics.
*
*     Returns     :   none.
*
*************************************************************************/


void prn_mem(void)

{
   REG1 int i = BUCKETS;
   REG2 int allocated = 0;
   REG3 int active    = 0;
   REG4 int size;

   while (i-- > 0)
    {
      size = (i % 2 ? 4 : 3) < < (i/2 - 2);
      allocated + = n_block[i] * size;
      active    + = a_block[i] * size;
    }
```

```
    active    = (int) (active    / 1024.0 + 0.5);
    allocated = (int) (allocated / 1024.0 + 0.5);
    printf("# %dK bytes active out of %dK allocated\n", active, allocated);

}     /* end prn_mem */
```

Bibliography

1. S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," IBM J. Of R & D, vol. 18, pp. 443-458, Sept. 1974.

2. D. W. Brown, "A State-Machine Synthesizer - SMS," Proc. 18th Des. Auto. Conference, pp. 301-304, Nashville, June 1981.

3. J. P. Roth, "Computer Logic Testing and Verification," Computer Science Press, 1980.

4. R. K. Brayton, G. D. Hatchel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," Kluwer Academic Publishers, 1984.

5. A. Svoboda, "Some applications of contact grids," Proc. of an Int. Symp. on the Theory of Switching, Harvard U. Press, Cambridge, Mass. 1959, pp. 293-305.

6. A. Svoboda, "Logical instruments for teaching logical design," IEEE Trans. on Educ. vol E-12, no. 4, Dec. 1969, pp. 262-273.

7. A. Svoboda, "The concept of term exclusiveness and its effect on the theory of Boolean functions," J. of ACM, vol. 22, no. 3, July 1975, pp. 425-440.

8. A. Svoboda, "Ordering of Implicants," IEEE Trans. Electron. Comput., vol. EC-16, Feb. 1967, pp. 100-105.

9. A. Svoboda, "Advanced Logical Circuit Design Techniques," New York: Garland STMP Press, 1979, pp. 130-157.

10. A. Marquand, "On logical diagrams for n terms," Phil. Mag. 12, 1881, pp. 266-270.

11. M. Nadler, "Topics in Engineering Logic," Pergamon Press, Oxford, 1962, Chaps. 1,3.

12. R. H. Vora, "An Algorithm for Multiple-output Boolean Logic Minimization," M.S. Thesis, Virginia Polytechnic Institute and State University, June 1987.

13. E. J. McCluskey, "Minimization of Boolean functions," Bell Syst. Tech. J. no. 35, 1965, pp. 1417-1444.

14. E. J. McCluskey, "Introduction to the Theory of Switching Circuits," New York: McGraw-Hill, 1965.

15. I. B. Pyne and E. J. McCluskey, "The reduction of redundancy in solving prime implicant tables," IRE Trans. Electron. Computers, vol EC-11, Aug. 1962, pp. 473-482.

16. A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley, Reading, Mass., 1983.

17. V. D. Agrawal, K. Cheng, D. D. Johnson and T. Ling, "Designing Circuits with Partial Scan," IEEE Design & Test of Computers, Apr. 1988, pp. 8-15.

The vita has been removed from the scanned document