MULTIPLE-VALUED LOGIC MINIMIZATION FOR PLA SYNTHESIS

by

Richard L. Rudell

Memorandum No. UCB/ERL M86/65

5 June 1986

COVER PAGE

MULTIPLE-VALUED LOGIC MINIMIZATION FOR PLA SYNTHESIS

by

Richard L. Rudell

Memorandum No. UCB/ERL M86/65

5 June 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MULTIPLE-VALUED LOGIC MINIMIZATION FOR PLA SYNTHESIS

by

Richard L. Rudell

Memorandum No. UCB/ERL M86/65

5 June 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

Multiple-Valued Logic Minimization for PLA Synthesis

Richard L. Rudell
Electronics Research Laboratory
Electrical Engineering and Computer Science Department
University of California
Berkeley, California 94720

June 5, 1986

ABSTRACT

Multiple-valued logic minimization is an important technique for reducing the area required by a Programmable Logic Array (PLA). This report describes both heuristic and exact algorithms for solving the multiple-valued logic minimization problem. These algorithms have been implemented in a C program called Espresso-MV.

Acknowledgements

My research advisor, Alberto Sangiovanni-Vincentelli, and the other authors of the book *Logic Minimization Algorithms for VLSI Synthesis*, Robert Brayton, Gary Hachtel, and Curt McMullen provided the foundation and motivation for this work. I acknowledge their work on logic minimization for multiple-output Boolean functions as fundamental for Espresso-MV. In particular, I would like to thank Robert Brayton for many interesting and challenging discussions on logic minimization. I also thank Bob for reading an early draft of this report and providing a critical review which greatly enhanced the quality of the final report.

I would like to thank Dr. Hugo De Man and his student Marc Bartholomeus (both of the University of Leuven) for their contributions to my collection of test cases for PLA minimization, and for providing results for their program Prestol-II. I also thank Michel Dagenais of McGill University for providing me with a copy of his program McBoole for comparison.

I am grateful to Robin Wallach for (among other things) putting up with the long hours it took to complete this report. I would like to thank all of the members of the Berkeley CAD group, especially Ken Kundert, Tom Laidig, Peter Moore, Tom Quarles, Jim Reed (while he was here), Rick Spickelmier, Albert Wang and Jacob White for making life interesting while I worked on Espresso-MV. I also thank Jacob White for convincing me to attend graduate school at Berkeley.

I would like to thank Dr. Gerhard Zimmerman for introducing me to computer-aided design while I was an undergraduate at the University of Minnesota. I would also like to thank Dr. Yacoub El-Ziq for showing me a paper in 1982 which described the Espresso algorithms for logic minimization. (Little did I know then what it would lead to !) When I started graduate school at Berkeley in the fall of 1983, I took on the task of implementing the Espresso algorithms in the C language as a project suitable for my Master's degree. This program, Espresso-IIC, was completed in January of 1984. Unfortunately, this project was finished too quickly. Professor Sangiovanni then encouraged me to work harder to extend the basic algorithms into the area of multiple-valued logic functions. Perhaps if I had taken longer for Espresso-IIC I would have completed my master's several years sooner, never having written Espresso-MV.

I acknowledge and greatly appreciate the support of an IBM Graduate Research Fellowship for two of the three years it has taken to complete this work. Partial funding from DARPA under grant N00039-C-0107 and SRC under contract 82-11-008 are also acknowledged.

Table of Contents

CHAPTER 1

Introduction

Programmable Logic Arrays (PLA's) are important subsystems in digital design of integrated circuits [FlM75, LBH75]. A PLA provides a simple and regular layout strategy for Boolean equations expressed in two-level canonical form, and is usually used to implement "random" logic (random in the sense that the designer sees no regular structure in the Boolean equations). Typical examples are the control logic for a reduced-instruction set computer, or the control logic for a microcode engine. With the addition of latches for feedback, PLA's are also often used for the combinational logic in a finite-state machine. The optimization of PLA's is a useful application of Computer-Aided Design to the automatic synthesis of custom VLSI designs.

Techniques for optimizing the structure of a PLA are becoming well understood. The optimization goals are to minimize the area occupied by the PLA, and to minimize the delay through the PLA. The regular structure of a PLA means that the area of the PLA is simply proportional to the number of product terms in the array, and, to a first-order approximation, the delay through the PLA is also proportional to the number of product terms (i.e., independent of the structure of each product term). Efficient algorithms can be developed to minimize the number of product terms in the array. A complete strategy for the design of a PLA macro-cell involves: (1) logic optimization of the PLA logic equations including input variable assignment and output phase assignment [BMH84, Sas84b]; (2) optimization of the PLA layout through folding and partitioning [DeS83, HNS82]; and (3) generation of the mask geometries implementing the PLA [Mah84].

This report is concerned with the logic optimization of PLA equations, and in particular, with the extension of the Espresso-II algorithms [BMH84] to the case of multiple-valued logic functions. Recent advances in multiple-output minimization of Boolean equa-

tions have produced algorithms able to minimize large Boolean functions. This is important for VLSI designs where a PLA can have more than 50 inputs and 50 outputs. Boolean minimization is perhaps the most important logic optimization procedure for PLA's, but it is not the only one. Other potential optimizations that change the form of the logic equations include using multiple-bit decoders on the inputs and choosing the most appropriate phase for each output. A multiple-valued minimization tool is an important part of each of these optimization procedures.

Espresso-II is a collection of algorithms for the minimization of two-level binary-valued switching functions [BMH84]. Research on the Espresso algorithms began in the summer of 1981 at the IBM T. J. Watson Research Center. A program implementing these algorithms was written in APL in the summer of 1982, and a C language version (called Espresso-IIC) was completed in January of 1984. The research culminated in the publication of the monograph *Logic Minimization Algorithms for VLSI Synthesis* [BMH84] in 1984. The public domain program Espresso-IIC was made available from the University of California simultaneously with the publication of the monograph.

Some early ideas on the problem of minimizing multiple-valued Boolean functions were presented in Chapter 5 of the monograph. Multiple-valued logic has many uses in optimizing structures built from binary-valued logic. For example, it has been shown that the *input-encoding problem* can be solved by treating it as a multiple-valued minimization problem. This can be applied to the optimal state-assignment problem (for many types of finite-state machines) [De83, DeB84] or to optimal assignment of opcodes in a processor so as to minimize the instruction decode logic [De84]. Multiple-valued logic functions can also be used to represent and minimize PLA's with *multiple-bit decoders* [FlM75, Sas84b].

With a simple transformation and the addition of an appropriate *don't-care set*, a multiple-valued minimization problem can be solved with any binary-valued minimizer [BMH84, Chapter 5]. However, this technique fails to exploit any knowledge of the structure of the multiple-valued minimization problem, and hence can be inefficient. For example, the *don't-care set* can become very large, and the number of binary variables needed

equals the sum of the number of values (for all variables) in the original problem. Hence, even Espresso-IIC was unable to minimize the transformed function resulting from performing a state-assignment on a dense 93-state machine. (It should be noted that the state machine had more than 3200 transitions and the transformed function had over 100 input variables, over 100 output functions, and there were more than 5000 don't-care terms.) Hence, it was hoped that a multiple-valued minimizer would be able to solve this large problem.

Also, it is known that the multiple-output minimization problem for PLA optimization is a special case of multiple-valued minimization. Therefore, it was hoped that a better understanding of the effect of the *output part* on the multiple-output minimization problem would result from working directly with the multiple-valued variables. For these reasons, I became interested in extending the Espresso-II algorithms to the more general framework of multiple-valued logic functions.

In this report I present the extension of Espresso-II to multiple-valued logic functions, and I report my experience with the program Espresso-MV that implements these extensions. Espresso-MV was found to be more efficient than Espresso-IIC due to its more uniform treatment of the output part, and hence has replaced Espresso-IIC even for minimization of binary-valued multiple-output functions. I also demonstrate how the Espresso-II algorithms can be extended to solve the Boolean minimization problem exactly. This exact algorithm relies on a new algorithm for the *minimum cover* problem which has proven to be efficient for solving large, cyclic covering problems. I present results for a large test set of PLA examples for several different minimization algorithms including the heuristic and exact modes of Espresso-MV. The PLA examples in the test set are also graded with respect to difficulty to organize the comparisons among competing algorithms. Finally, I report on the successful multiple-valued minimization of the large state machine mentioned above.

In particular, the basic definitions of multiple-valued logic functions are presented in Chapter 2 along with the necessary extensions to the fundamental concepts of Espresso-II

for dealing with multiple-valued logic functions. The key concepts in this chapter are the extension of the Shannon Cofactor and unate functions to multiple-valued logic functions.

Background on logic optimization of PLA's, including logic minimization, input-variable assignment for two-bit decoders, and output phase assignment, is presented in Chapter 3. This provides motivation for interest in multiple-valued logic minimization as well as an introduction to the exact minimization problem.

The algorithms used for heuristic minimization are described in Chapter 4. From an outside view, the algorithms appear similar to the original Espresso-II algorithms. However, the use of multiple-valued logic simplifies the description of many of the algorithms, and hence, the algorithms are explained in detail.

The exact minimization problem is considered in Chapter 5. I show how the algorithms used by Espresso-MV can be used to create a minimization algorithm which provides the minimum solution to the minimization problem. In particular, a new algorithm for finding the minimum cover of the prime implicant table is presented that has successfully completed the covering for several functions that have appeared in the literature without a solution.

Experimental results with the C language version of the program Espresso-MV are given in Chapter 6. First, results evaluating the difficulty of the PLA test set are presented. For those examples where the exact minimizer is able to generate a solution, Espresso-MV is much faster and is produces solutions which are very close to the exact minimum. When minimizing multiple-valued functions, Espresso-MV is much more efficient than using a two-valued logic minimizer with an appropriate don't-care set. Also, Espresso-MV is more efficient than Espresso-IIC due to the uniform treatment of the output part.

Appendix A contains user documentation for the program Espresso-MV, including the command line options and file formats.

Appendix B contains results for the 145 PLA's in the Berkeley PLA Test Set. Optimum results are reported for over 100 of the PLA's, and the best known solutions are reported for the remaining problems. The examples where the minimum solution is not known can be be viewed as a challenge to any Boolean minimization program to find the minimum solution, or to find a solution better than that reported by Espresso-MV.

CHAPTER 2

Basic Definitions

The purpose of this chapter is to review the definitions that will be used in dealing with multiple-valued input binary-valued output functions, and to define the notions of Shannon cofactor, *weakly-unate* and *strongly-unate* for these types of functions. There is a wealth of data in the literature regarding these types of functions. In particular, I follow the notation and terminology of Sasao [Sas81, Sas83, Sas84b] for multiple-valued functions. Chapters 2 and 3 of *Logic Minimization Algorithms for VLSI Synthesis* [BMH84] are valuable references for these definitions in the special case of binary-valued multiple-output functions.

2.1. Multiple-Valued Functions

Let p_i for $i = 1 \cdots n$ be positive integers representing the number of values for each of n variables. Define the set $P_i \equiv \{ 0, \cdots, p_i - 1 \}$ for $i = 1 \cdots n$ which represents the p_i values that variable i may assume, and define $B \equiv \{ 0, 1, * \}$ which represents the value of the function. A **multiple-valued input, binary-valued output function**, f, (hereafter known as a **multiple-valued function**) is a mapping

$$f : P_1 \times P_2 \times \cdots \times P_n \rightarrow B$$

The function is said to have n multiple-valued inputs, and variable i is said to take on one of p_i possible values.

Each element in the domain of the function is called a **minterm** of the function.

An enumeration of all minterms with the value of the function is called a **truth table**.

The value $* \in B$ will represent a minterm for which the function value is allowed to be either 0 or 1. Hence, we allow functions which are *incompletely specified*.

An n-input, m-output switching function can be represented by a multiple-valued function of $n+1$ variables where $p_i = 2$ for $i = 1 \cdots n$, and $p_{n+1} = m$. This special case is called a **multiple-output function**. It is easily proven that the Boolean minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form [Sas78, Theorem 4.1].

As an example of a multiple-valued function, I define a function of three variables with the first variable assuming three values ($p_1 = 3$), the second variable assuming two values ($p_2 = 2$), and the third variable assuming three values ($p_3 = 3$). The function is defined by the following truth table:

X_1	X_2	X_3	*value*
0	0	0	1
0	0	1	1
0	0	2	0
0	1	0	1
0	1	1	0
0	1	2	1
1	0	0	0
1	0	1	1
1	0	2	1
1	1	0	1
1	1	1	1
1	1	2	0
2	0	0	*
2	0	1	*
2	0	2	0
2	1	0	1
2	1	1	*
2	1	2	0

Note that some of the function values are * indicating that the function value may be either 0 or 1 for these minterms.

Let X_i be a variable taking a value from the set P_i, and let S_i be a subset of P_i. $X_i^{S_i}$ represents the Boolean function

$$X_i^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \\ 1 & \text{if } X_i \in S_i \end{cases}$$

$X_i^{S_i}$ is called a **literal** of variable X_i. If $S_i \equiv \emptyset$, then the value of the literal is always 0, and the literal is called **empty**. If $S_i \equiv P_i$, then the value of the literal is always 1, and

the literal is called **full.**

In the example, $P_1 = \{ 0,1,2 \}$, and if $X_1 = 1$ then $X_1^{\{0,2\}} = 0$, and $X_1^{\{1\}} = 1$.

The **complement** of the literal $X_i^{S_i}$ (written $\bar{X}_i^{S_i}$) is the literal $X_i^{\bar{S}_i}$. The complement of a literal evaluates to 0 when the literal evaluates to 1, and vice-versa.

A **product term** (sometimes simply a **term**) is a Boolean product (or AND) of literals. If a product term evaluates to 1 for a given minterm, the product term is said to contain the minterm. If a literal in a product term is full, the product term does not depend on that variable. Without loss of generality, a product term consists of the Boolean AND of a literal for each variable.

If a literal in a product term is empty, the product term contains no minterms, and is called the **null product term** (written $\emptyset$). If all literals in a product term are full, the product term contains all minterms, and is called the **universal product term.**

A **sum-of-products** (also called a **cover**) is a Boolean sum (or OR) of product terms. If any product term in the sum-of-products evaluates to 1 for a given minterm, then the sum-of-products is said to contain the minterm.

The set X_{on} (called the **ON-set**) is the set of minterms for which the function value is 1 (i.e., $X_{on} \equiv f^{-1}(1)$). Likewise, the set X_{off} (called the **OFF-set**) is the set of minterms for which the function value is 0 (i.e., $X_{off} \equiv f^{-1}(0)$), and X_{dc} (called the **DC-set**) is the set of minterms for which the function value is unspecified (i.e., $X_{dc} \equiv f^{-1}(*)$).

An **algebraic expression** for f is a Boolean expression (written using Boolean sums and Boolean products of literals) which evaluates to 1 for all minterms of the ON-set, evaluates to 0 for all minterms of the OFF-set, and evaluates to either 0 or 1 for all minterms of the DC-set.

Proposition 2.1: An algebraic expression for f can always be written in sum-of-products form.

Likewise, it is possible to define a **sum term** as a Boolean sum of literals, and a **product-of-sums** as a Boolean product of sum terms. However, we restrict our attention to sum-of-product forms because of the next proposition:

Proposition 2.2: The minimal product-of-sums form for a function f can be derived from the minimal sum-of-products form for X_{off}.

An **implicant** of a function f is a product term which does not contain any minterm in the OFF-set of the function.

A **prime implicant** of a function f is an implicant which is contained by no other implicant of the function.

An **essential prime implicant** is a prime implicant which contains some minterm not contained by any other implicant.

In the example, $X_1^{\{0\}}X_2^{\{0,1\}}X_3^{\{0,1\}}$ is a product term (which is *not* an implicant of the function), and a sum-of-products expression for the function is:

$$X_1^{\{0\}}X_2^{\{1\}}X_3^{\{0,2\}} \cup X_1^{\{1\}}X_2^{\{0\}}X_3^{\{1,2\}} \cup$$
$$X_1^{\{0\}}X_2^{\{0\}}X_3^{\{0,1\}} \cup X_1^{\{1,2\}}X_2^{\{1\}}X_3^{\{0,1\}}$$

2.2. Operations on Product Terms and Covers

In the definitions which follow, $S = X_1^{S_1}X_2^{S_2} \cdots X_n^{S_n}$ and $T = X_1^{T_1}X_2^{T_2} \cdots X_n^{T_n}$ represent product terms, and F and G will represent sum-of-product expressions.

The **volume** of a product term S ($vol(S)$) is the number of minterms which the product term contains. (i.e., $\prod_{i=1}^{n} |S_i|$). S is said to be **larger than** T if $vol(S) > vol(T)$.

A product term S is said to **contain** a product term T ($T \subseteq S$) if $T_i \subseteq S_i$ for all $i = 1 \cdots n$. If, in addition, $S \neq T$, then S is said to **strictly contain** T ($T \subset S$). S (strictly) contains T if S (strictly) contains all of the minterms that T contains.

The **complement** of a product term S ($\bar{S}$) (computed using De Morgan's Law) is the sum-of-products $\bigcup_{i=1}^{n} \bar{X}_i^{S_i}$.

The **intersection** of product terms S and T $(S \cap T)$ is the product term $X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}$ which is the largest product term contained in both S and T. If $S_i \cap T_i = \varnothing$ for some i, then $S \cap T = \varnothing$ and S and T are said to be **disjoint**. If $S \cap T$ are not disjoint, they are said to **intersect**. Likewise, the intersection of two covers F and G is defined as the union of the pairwise intersection of the cubes from each cover.

The **supercube** of S and T $(supercube(S, T))$ is the product term $X_1^{S_1 \cup T_1} X_1^{S_2 \cup T_2} \cdots X_n^{S_n \cup T_n}$ which is the smallest product term containing both S and T. Likewise, the supercube of a cover F is the smallest product term containing every product term of F.

The **distance** between S and T equals the number of empty literals in their intersection . If the distance between two cubes is 0 they intersect, otherwise they are disjoint.

The **sharp-product** of S and T $(S \# T)$ is the null product term if S and T are disjoint. Otherwise, it is the sum-of-products:

$$S \# T = S \cap \bar{T} = \bigcup_{i=1}^{n} X_1^{S_1} \cdots X_i^{S_i \cap \bar{T}_i} \cdots X_n^{S_n}$$

$S \# T$ contains all of the minterms of S which are not contained by T.

The **consensus** of S and T $(consensus(S, T))$ is the sum-of-products:

$$\bigcup_{i=1}^{n} X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}$$

If $distance(S, T) \geq 2$ then $consensus(S, T) = \varnothing$. If $distance(S, T) = 1$ and $S_i \cap T_i = \varnothing$, then $consensus(S, T)$ is the single product term $X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}$. If $distance(S, T) = 0$ then $consensus(S, T)$ is a cover of n terms. If the consensus of S and T is nonempty, it contains minterms of both S and T. Likewise, the consensus of two covers F and G is defined as the union of the pairwise consensus of the product terms from each cover.

The **cofactor** (or **cube restriction**) of S with respect to T (S_T) is empty if S and T are disjoint. Otherwise, it is the product term $X_1^{S_1 \cup \bar{T}_1} X_2^{S_2 \cup \bar{T}_2} \cdots X_n^{S_n \cup \bar{T}_n}$. Likewise the cofactor of a cover F with respect to a cube S (F_S) is the union of the cofactor of each

cube of F with respect to S.

2.3. Positional Cube Notation

Let $X_1^{S_1}X_2^{S_2}\cdots X_n^{S_n}$ be a product term. This product term can be represented by a binary vector:

$$c_1^0c_1^1\cdots c_1^{p_1-1} \;-\; c_2^0c_2^1\cdots c_2^{p_2-1} \;-\; c_n^0c_n^1\cdots c_n^{p_n-1}$$

where $c_i^j=0$ if $j \notin S_i$, and $c_j^i=1$ if $j \in S_i$. This is called the **positional cube notation** or more simply a **cube** [Su72]. A cube is a convenient representation for a product term, and the terms cube and product term will often be used interchangeably. (For example, a prime cube is a cube which represents a prime implicant.)

The notation c_i represents the binary vector $c_i^0c_i^1\cdots c_i^{p_i-1}$, and $|c_i|$ represents the number of 1's in the binary vector. The notation $c_i \cup d_i$ refers to the bit-wise OR of two binary vectors, $c_i \cap d_i$ refers to the bit-wise AND of two binary vectors, and $\bar{c}_i$ refers to the bit-wise complement of a binary vector.

A sum-of-products will be represented by a set of cubes, also called a cover. A cover also has a natural two-dimensional matrix representation, where each row of the matrix is a cube.

Continuing with the example, the following is a cover for the function:

X_1	X_2	X_3
012	01	012
100	01	101
010	10	011
100	10	110
011	01	110

The cube representation of a product term is useful because Boolean operations on the binary vectors correspond to the useful operations on the product terms. For example, one product term contains another if and only if their corresponding cubes contain each other as bit-vectors, the intersection of two cubes is the cube which results from component-wise Boolean AND of the two cubes, and the supercube of two cubes results from the

component-wise Boolean OR of the two cubes.

For computer implementation of the algorithms, the cube provides a convenient data structure where one bit is used for each part of the cube. It is possible to perform operations on the cubes as word-wide operations (i.e., the bit-wise Boolean AND of two 32-bit vectors on most 32-bit computers) which is more efficient than manipulating the binary vectors element by element.

2.4. Generalized Shannon Cofactor and Multiple-Valued Unate Functions

In [BMH84], unate functions were defined for binary-valued functions, and several important properties of unate functions were proven. In particular, it was shown that the problems of finding the smallest cube containing the complement of a function (an important step of REDUCE), and the problem of determining whether a function is a tautology (an important step of both IRREDUNDANT and ESSENTIAL) can be answered quickly for unate functions. When these results are combined with Shannon's Theorem and the cofactor operation defined in Section 2.2, efficient recursive algorithms can be devised which attempt to split the function so as to reach a leaf where the function is unate, and then quickly determine the result for the unate function.

The basic paradigm for manipulating multiple-valued functions is to use the multiple-valued extension of the Shannon Cofactor which is called the *Generalized Shannon Cofactor* [Sas84a, Lemma 3.2]: In Proposition 2.3, F is a cover of a multiple-valued function. Recall that F_{c^i} represents the cofactor of F with respect to the cube c^i.

Proposition 2.3: Let $c^i, i = 1 \cdots m$ be a set of cubes satisfying $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \emptyset$ for $i \neq j$. Then,

$$F = \bigcup_{i=1}^{m} c^i \cap F_{c^i}$$

Remark: Using simple algebraic operations of Boolean algebra, it is east to show that the operations of tautology, complementation and computing the supercube of the complement of a cover (an important operation of REDUCE) can be computed using the properties:

$$F \equiv 1 \iff F_{c^i} \equiv 1 \text{ for } i = 1 \cdots m$$

$$\bar{F} = \bigcup_{i=1}^{m} c^i \cap \bar{F}_{c^i}$$

$$supercube(\bar{F}) = supercube(\bigcup_{i=1}^{m} c^i \cap supercube(\bar{F}_{c^i}))$$

In this section, I consider how to extend the definition of a binary-valued unate function to the multiple-valued case. I show that there are two useful extensions. The first, referred to as *weakly-unate*, preserves the important property that tautology and computing the supercube of the complement are trivial operations for weakly-unate functions. However, a weakly-unate function does not satisfy some of the other properties of binary-valued unate functions, namely, that all prime implicants of a binary-valued unate function are essential or that the complement of a binary-valued unate function is unate. Hence, I also define a *strongly-unate* function (a stronger condition on the function than weakly-unate) which preserves these two properties. It is important to note that the definitions of weakly-unate and strongly-unate coincide for the special case of binary-valued functions.

In this section, I also consider how to choose the cubes $c^1, c^2, \cdots c^m$ when partitioning the function.

2.4.1. Weakly-Unate Functions

Definition 2.1: A function is said to be *weakly-unate* in variable X_i if there exists a j such that changing the value of X_i from value j to any other value causes the function value, if it changes, to change from 0 to 1. If a function is weakly unate in all of its variables, then the function is said to be weakly unate.

If a function is weakly unate in variable X_i, then changing the value of variable X_i to value j causes the value of the function, if it changes, to change from 1 to 0. Hence, there is no need to define both unate increasing and unate decreasing functions.

Definition 2.2: A cover F is said to be *weakly-unate* in variable X_i if there exists a j such that all cubes which depend on variable X_i contain a 0 in the position j.

For example, the following cover is weakly-unate because it is weakly-unate in part 1 of variable 1, part 1 of variable 2, and part 5 of variable 3.

11111-00001-11110
01100-00011-01010
01010-00100-11111
00110-01001-11010
00001-11111-10110

Proposition 2.4: A weakly-unate cover in variable X_i is a cover for a weakly-unate function in variable X_i.

Proposition 2.5: A function f is weakly-unate in variable X_i if and only if there exists a j such that each prime implicant of f which depends on variable X_i has a 0 in part j of variable X_i. Hence, a prime cover for a weakly-unate function is also a weakly-unate cover.

The proofs of these propositions are trivial extensions of the proof for the binary-valued case as in Propositions 3.3.1, 3.3.2 and 3.3.3 of [BMH84].

A simple test for whether a cover is *weakly unate* in a variable X_i is to form the supercube of all cubes of F which do not have a full literal in variable X_i. This supercube has a 0 in any parts of X_i that are weakly unate.

The following result is useful for determining whether a weakly-unate function is a tautology:

Proposition 2.6: Let F be a weakly-unate cover in variable X_i. Let $G = \{ c \in F \mid c$ does not depend on $X_i \}$. Then $G \equiv 1 \leftrightarrow F \equiv 1$.

Proof: Clearly, if $G \equiv 1$, then $F \equiv 1$. Assume that j is the part required by Definition 2.2 for F to be weakly-unate in variable X_i, and assume $G \not\equiv 1$. Then there exists a minterm $m \in \overline{G}$ with a 1 in value j of variable X_i. However, F is unate in X_i, and hence no terms of F have a 1 in value j of variable X_i. Therefore, if follows that $m \notin F$, and hence $F \not\equiv 1$.

∎

There is a special case when all variables are weakly unate:

Proposition 2.7: A weakly-unate cover is a tautology if and only if one of the cubes in the cover is the universal cube.

Proof: By repeated application of Proposition 2.6, the function is a tautology if and only if $G = \{ c \in F \mid c$ does not depend on X_i for all $i \}$. Only the universal cube can be in G, and hence $G \equiv 1$ if and only if the original function contains the universal cube.

∎

Hence, the weakly unate condition on a function is sufficient to allow a simplification of the function for the purpose of answering the tautology question. Also, as is shown in Section 4.6, weak-unateness is sufficient to determine the smallest cube containing the complement of a function. Two other useful properties of binary-valued unate functions are: (1) all prime implicants of a binary-valued unate function are essential, and (2) the complement of a binary-valued unate function is also unate. However, these two properties do not hold for weakly-unate functions. Hence, there is motivation to find a stronger condition than weakly-unate which preserves these properties.

To understand the limitation of weakly unate, consider that, in the binary-valued case, if a cover F is unate, then the cover contains a cube c if and only if the cube is contained by some cube of the cover. This is true because F_c is unate if F is unate, and hence, $F_c \equiv 1$ if and only if F_c contains a universal cube. However, F_c contains a universal cube if and only if it contains a single cube which contains c. However, it is not true that F_c is weakly unate whenever F is weakly unate as the following example shows:

10-11-11-111
11-10-10-100
11-11-10-010

cofactoring against c=10-10-10-110 produces

11-11-11-111
11-11-11-101
11-11-11-011

which is not weakly unate in variable 4. Also note that the function F contains c, but that no single row of F contains c.

Also, in the binary-valued case, all primes of a unate function are essential, and the complement of a unate function is unate. However, the function presented earlier violates both of these properties:

11111-00001-11110	(essential)
01100-00011-01010	(nonessential)
01010-00100-11111	(essential)
00110-01001-11010	(nonessential)
00001-11111-10110	(essential)

The complement of this function is:

00110-01000-00101
11111-00001-00001
00001-11110-01001
01100-00010-10101
11000-11000-11111
10100-10100-11111
10010-10010-11111

which is weakly unate in variable 3, but not in variables 1 or 2.

Hence, we seek a condition stronger than weakly unate that preserves these properties.

2.4.2. Strongly-Unate Functions

Definition 2.3: A function is said to be *strongly unate* in variable X_i if the values of X_i can be totally ordered via $\lesssim$ such that changing the value of variable X_i from value j to value k (where $j \lesssim k$) causes the function value, if it changes, to change from 0 to 1. If all variables of a function are strongly unate, then the function is called strongly unate.

Clearly any function which is strongly unate is also weakly unate in the part of variable X_i which is less than (via $\lesssim$) all the remaining parts. A strongly-unate function provides a total order for all of the parts, and a weakly-unate function merely provides a single part which is less than all remaining parts.

Proposition 2.8: A strongly-unate cover contains a cube if and only if the cube is contained in some cube in the cover.

Proof: If H is strongly unate, then H_c consists of those cubes of H which intersect with c, with the addition of full columns in the positions where c_i^j is 1. Hence, H_c is also strongly unate and is a tautology if, and only if, it contains a universal cube. But, H_c can contain a universal cube if, and only if, it contains a single cube which contains c.

∎

Proposition 2.9: All primes of a strongly-unate function are essential.

Proof: Exactly as proposition 3.3.6 in [BMH84], where Proposition 2.8 replaces proposition 3.3.5 of [BMH84].

∎

Proposition 2.10: The complement of a strongly-unate function is strongly-unate.

The algorithms developed for Espresso-MV make use of weakly-unate functions, but do not make use of strongly-unate functions. The description here of strongly-unate functions is presented for the sake of completeness. I wish to thank Dr. Agnes Hui Chan of Mitre Corporation for suggestions leading to the definition of strongly unate.

2.5. Choice of Partition

Once a cofactor F_{c^i} becomes weakly unate, it is trivial to determine if the function is a tautology, or it is trivial to compute the smallest cube containing the complement of the function. Hence, we wish to choose a partition $c^i, i = 1 \cdots m$ so that each cofactor F_{c^i} becomes a weakly-unate function as quickly as possible.

The choice of partition is simplified by first choosing a *splitting* variable, followed by a choice of a partition of the splitting variable into a number of cubes which depend on only the splitting variable. Any cube in the cover which is independent of the splitting variable is duplicated in all branches of the recursion, hence this consideration enters into our choice of the splitting variable.

There is an important difference between the binary-valued case and the multiple-valued case. When the variable has only two values, the function is split with the cubes $c^1 = X_i^{\{0\}}$ and $c^2 = X_i^{\{1\}}$; the only choice is which variable X_i to use for splitting. But this choice is easy to make. The most *binate* variable [BMH84], defined as the variable which has the most cubes in the cover which depend on it, leads to the minimum duplication of cubes after applying the Shannon Cofactor. As a secondary consideration, it is desirable to keep the recursion balanced. Therefore, as a tie-breaker, Espresso chooses the variable which has the closest to an equal number of cubes with $X^{\{0\}}$ and $X^{\{1\}}$. These rules guarantee a minimum of duplication between F_{c1} and F_{c2} at the next level of the recursion.

When a variable has more than two values, however, we must also choose how to partition the parts of the variable into a number of different cubes. There are two possibilities:

(1) Partition the values of the splitting variable into two disjoint sets $l \subset P_i$ and $r \subset P_i$ (with $l \cap r = \emptyset$, and $l \cup r = P_i$). The function F is then split into two parts:

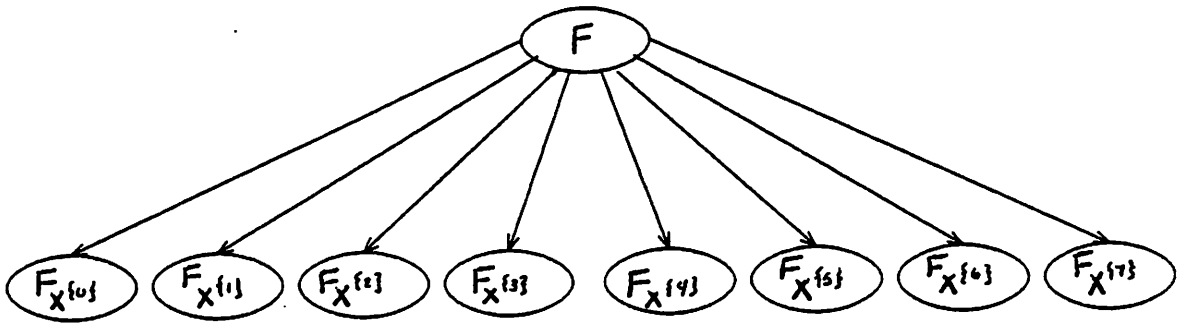
$$F = (X^l \cap F_{X^l}) \cup (X^r \cap F_{X^r})$$

This enables us to maintain a binary recursive strategy. However, unlike the binary-valued case, this does not necessarily make each of the cofactors independent of the splitting variable.

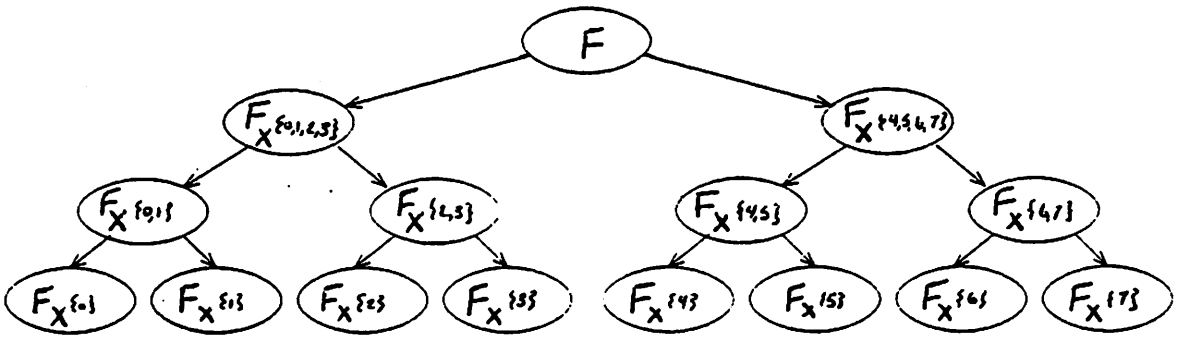
(2) Partition the values of the splitting variable X_i into the p_i cubes $X^{\{0\}}, X^{\{1\}}, \cdots X^{\{p_i-1\}}$. This effectively eliminates variable X_i at this level of the recursion, and forms a p_i-way splitting of the function:

$$F = \left(X^{\{0\}} \cap F_{X^{\{0\}}}\right) \cup \left(X^{\{1\}} \cap F_{X^{\{1\}}}\right) \cup \cdots \left(X^{\{p_i-1\}} \cap F_{X^{\{p_i-1\}}}\right)$$

I chose strategy 1 because it leaves more degrees of freedom at the next level of the recursion. For example, if a variable has 8 values, splitting on all 8 values (as suggested by (2)) gives us the 8-way tree shown below:



Using the binary partition (as suggested by (1)) and choosing the same variable for splitting at the next two levels, we get the binary tree shown below:



However, at either the second or third level there is more freedom in that a different variable may be chosen for splitting. Hence, strategy 1 reduces to strategy 2 in the case that the same variable s chosen at each level. Note too, that strategy 1 also gives us a natural way to use a tree structure to perform the n-way merge which would be required by strategy 2.

2.5.1. Choice of Splitting Variable

The simple test of which variable has the most number of "active" values, i.e., a value which does not have a column of all 1's in the cover is used to select the variable for

splitting. Ties are broken by selecting the variable with the most total number of 0's, and then by selecting the variable which has the fewest number of parts which contain a 0. Note that this heuristic is equivalent to the **binate** heuristic of Espresso-II in the case of binary-valued variables. And, when the variable are binary valued, it achieves the goal of making the cover weakly unate.

2.5.2. Choice of Partition for the Splitting Variable

It was mentioned earlier that in the multiple-valued case it is more difficult to choose a partition of the values which yields a minimum of duplication of cubes during the recursion. This can be formulated as follows:

Problem: Find a set of values c^1 and c^2 such that the total number of cubes in $F_{c^1} \cup F_{c^2}$ is minimized.

Consider the submatrix of the cover F restricting our attention to only the columns associated with variable X_i. Consider finding a row and column permutation of the matrix into the form:

A	0
B	
0	C

which minimizes the number of rows of B.

The columns of A are identified with the first half of the partition c^1, and the columns of C are identified with the second half of the partition c^2. The cubes of B are duplicated in both halves of the recursion.

This problem is a standard partitioning problem. Form a graph from the columns of the matrix by placing an edge between two columns that have 1's in the same row. The weight of this edge is equal to the number of different rows in which these columns share 1's. The problem is then to partition the nodes into two disjoint sets such that a minimum total edge weight connects the two sets.

Solving the preceding problem is potentially expensive so I choose instead to partition the active parts to place the first $\frac{n}{2}$ into the set l, and the remaining $\frac{n}{2}$ active parts into the set r. This heuristic is very fast to compute (if a little crude), but it remains no worse than an initial n-way split on the function. My experience with Espresso-MV is that even with this simple heuristic, up to twenty-five percent of the time for the recursive routines is spent determining the partition for the next step of the recursion.

CHAPTER 3

Logic Optimization of PLA's

In this chapter, I consider several important logic optimization steps in the design of a PLA (i.e., optimizations that change the structure of the Boolean equations implemented in the PLA). The optimizations I consider are: (1) Logic minimization, (2) Output phase assignment, (3) Input variable assignment and the use of two-bit decoders, (4) Optimal encoding of the input values to the PLA, and (5) Optimal encoding of the output values of the PLA. The intent is to show how multiple-valued minimization can be applied to each of these problems.

3.1. Logic Minimization

The logic minimization problem is to find a minimum cost cover for a given Boolean function. The cost of a cover is defined as the sum of the costs of the cubes in the cover. One typical cost function for a cube is:

$$cost(c) = 1 \tag{3.1}$$

This reflects the primary goal of minimizing the number of product terms in the PLA (and hence, minimizing both the area and delay associated with the PLA). It can be shown that restricting attention to prime implicants is sufficient to find a minimum solution for this cost function.

A secondary concern is to also minimize the total number of transistors in the PLA (hence reducing parasitic capacitance in the array, and improving the prospects for folding the array). Hence, another potential cost function is:

$$cost(c) = 1 + \frac{\#\ transistors\ to\ implement\ c}{maximum\ \#\ transistors\ in\ any\ implicant} \tag{3.2}$$

Also, the cost function

$$cost(c) = \#transistors\ needed\ to\ implement\ c \tag{3.3}$$

appears useful in applications of Boolean minimization to multiple-level networks.

When the cost function for an implicant obeys the property that

$$c \subseteq d \implies cost(c) \leqslant cost(d) \tag{3.4}$$

then the solution to the minimization algorithm consists of prime implicants [Rot80]. However, cost functions 3.2 and 3.3 violate this property because of the asymmetric nature of the output-part of the implicant. Consider the two implicants:

10-11-10-110
10-11-10-100

The first three variables each have two parts, and the fourth variable, representing the output-part of the multiple-output minimization, has three parts. Each 1 in the fourth variable corresponds to a transistor in the output-plane of the PLA. The first implicant contains the second implicant, but the second implicant costs less (using 3.2 or 3.3) because fewer transistors are needed to implement it.

Hence, with these last two cost functions, it is possible that the minimum solution will not consist of prime implicants. Most minimization algorithms (whether heuristic or exact) sidestep this problem by limiting themselves to solutions consisting of prime implicants, and then, as a second step, they attempt to minimize the number of transistors needed to implement the PLA. The primary cost function used is 3.1, with consideration also given to reducing the number of transistors.

Finally, it is important to remember that the goals of minimizing the number of product terms and the total number of transistors sometime conflict. It is possible that a cover with fewer product terms may require more transistors than a different cover with more product terms. As is shown in Chapter 6, for the problems which are solvable by an exact minimizer, Espresso generates solutions with more product terms, but fewer transistors.

The techniques for solving this optimization problem exactly are well known [McC56, Qui55]. Using cost function 3.1 the two steps are:

(1) Generate the set P of all prime implicants of the function;

(2) Extract from P a minimum subset that is sufficient to represent the function.

Many algorithms have been presented for generating all of the prime implicants of a multiple-output function [DAR86, Rot80, Tis67], and most of these can be easily extended to the case of multiple-valued binary functions. In Section 4.3, two techniques are presented for generating the complete set of prime implicants for multiple-valued functions.

Solving the second step usually proceeds by forming the *prime implicant table*, A, which is a binary matrix with the prime implicants listed across the columns of the table, and the minterms listed down the rows of the table. A 1 is placed in position A_{ij} if the minterm i is contained in the prime implicant $p_j \in P$. The problem is then reformulated as a special case of an integer-programming problem known as the *minimum cover problem*. This problem is to find a binary vector x satisfying

$$A \bullet x^T \geqslant (1,1,\ldots,1)$$

(i.e., each element of $A \bullet x^T$ is greater than or equal to 1) such that

$$\sum_{i=1}^{|P|} cost(p_i)\, x_i$$

is minimized.

The procedures *row dominance* and *column dominance* (described in Chapter 5) exploit relationships among the rows and columns of A to reduce the size of the matrix. Thus, one of the goals of an exact minimization algorithm is to generate directly the reduced form of the table. For example, generating only the prime implicants and using them for the columns of the matrix is a heuristic (which applies when the cost function follows property 3.4) for generating a reduced form of the table. Likewise, the IRREDUNDANT algorithm presented in Section 4.4 is able to avoid listing minterms along the rows of A in

favor of higher dimensional cubes, and hence it directly generates a reduced form of the prime implicant table.

3.1.1. Difficulty of Logic Minimization

There are good reasons why an exact minimization algorithm cannot always produce a result within a reasonable expenditure of computer resources. I review here two well known failures for minimization algorithms that rely on the two steps outlined above.

First, the generation of the set of all prime implicants may fail because there are too many prime implicants to be enumerated. There are functions with an exponential number of prime implicants (as a function of the number of input variables). These "bad" examples are often cited when referring to the complexity of generating all of the prime implicants of a function. However, it is often the case that these bad functions also require a very large number of implicants just to represent the function in two-level form (i.e., the minimum subset of the set of all prime implicants is also of exponential complexity in the number of inputs).

Example 1: The parity function of n variables has 2^{n-1} prime implicants, but also requires 2^{n-1} implicants for a minimum solution. Thus, if one can afford to present the exact minimization program with a two-level form of this function, then the exact minimization program can always afford to generate the set of all prime implicants (because the ratio between the size of the minimum cover and the set of all primes is $O(1)$).

Example 2: The symmetric function of $3k$ variables described by "between k and $2k$ of the input variables are 1" has $\left\lfloor \frac{3k}{k} \right\rfloor \left\lfloor \frac{2k}{k} \right\rfloor$ prime implicants (which, asymptotically, equals $\frac{3^n}{n}$ for $n = 3k$). But this function requires $\left\lfloor \frac{3k}{k} \right\rfloor$ implicants to describe in two-level form, and hence again, if one can afford to represent the function at all in two levels, then one can afford to generate all of the prime implicants for the function (because the set of all primes is smaller than the square of the number of terms in the minimum cover).

Hence, these two examples are not sufficient to prove the case that the generation of the set of all prime implicants is difficult. One metric for measuring the complexity of generating the set of all prime implicants is the relationship between the size of the minimum cover ($|M|$) for the function and the number of prime implicants ($|P|$) for the function. Generating the set of all prime implicants is most difficult when the complexity of $|P|$ as a function of $|M|$ (and not n) is greatest.

A new result [McS84] shows that the worst case complexity for generating the set of all prime implicants is $|P| = 2^{|M|}-1$, and that this bound is precise in that there exists a function with this complexity between its minimum cover and the set of all primes for the function. Thus, there is the negative result that the set of all primes can become too large to enumerate even though it is possible (and efficient) to represent the function in two-level form.

The second failure of exact minimization algorithms is that they rely on solving the minimum covering problem which is known to be NP-hard [GaJ79]. Further, the parameter that controls the complexity of the covering problem is $|P|$ the size of the set of all prime implicants. Hence, if a branch and bound strategy is used to solve this covering problem, then the complexity can be as bad as $2^{|P|}$, or, with respect to the users initial input, the complexity can be as bad as $2^{2^{|M|}}$.

This is not to say that exact minimization is not possible for many functions. The problem is that, in general, it is not possible to tell *a priori* which problems cannot be solved within a reasonable expenditure of resources. Further, there will always be problems which will be beyond the reach of any exact minimization procedure. Hence, there is strong motivation for good approximate algorithms for solving the logic minimization problem.

3.2. Output Phase Assignment

The output of a PLA is typically buffered with either a noninverting or an inverting buffer (depending on the actual implementation of the PLA). However, consider selectively changing each of the output buffers to be either inverting or noninverting, and choosing, for each function, whether to implement the logic equations for the function or its complement. As the phases of the outputs are changed, the Boolean equations which are implemented by the PLA are changed, and hence the size of a minimum set of terms to implement these equations also changes. Quite often this can reduce the number of rows needed for the PLA as well as the number of transistors, and leads to a more area and

time-efficient implementation of the function. The problem of choosing for each output whether to implement its positive phase or its negative phase is called the *output-phase assignment* problem.

It is still an open problem as to how to make an optimal choice from the 2^m possible assignments of phases. Note that merely minimizing each function once for the function, and again for the complement of the function, is not a good strategy for choosing the phase assignments. In particular, the greatest advantage from the choice of phase assignments for a PLA implementation comes when a single product term can be used in several outputs, and this simple algorithm ignores this effect. Sasao [Sas84b] suggests the following algorithm for determining the phase assignment for a PLA with the outputs $f_0, f_1, \cdots, f_m$:

(1) Form the *double-phase characteristic function* which is a PLA implementing the $2m$ functions $f_0, f_1, \cdots, f_m, \overline{f}_0, \overline{f}_1, \cdots, \overline{f}_m$.

(2) Use a heuristic logic minimization algorithm to find a minimal cover for the double phase characteristic function.

(3) Select from among the cubes in this minimal cover a minimum subset which is sufficient to realize either f_i or $\overline{f}_i$ for each $i = 1 \cdots m$. This is done by either expanding the *covering expression*, or by using a branch and bound method. Each of these techniques is described in more detail in [Sas84b].

(4) Form the PLA which implements the output phases as chosen in Step 3, and find a minimal cover for this function.

Most of the time taken for this algorithm is in the heuristic minimizer, although step 3 is potentially difficult for a problem with many outputs.

As an example of the usefulness of output phase assignment, I consider two of the PLA's from the SOAR microprocessor [SKF85] which used PLA-based control logic. Information on the two largest PLA's are given in Table 3.1.

name	inputs	outputs	rows in minimized PLA	rows after output phase optimization	rows using complement for all functions
cpla1	9	16	38	28	26
xcpla1	9	23	41	32	30
both	10	39	79	45	43

Table 3.1. SOAR Control PLA's Before and After Output Phase Assignment.

It was intended that the control logic would be implemented as a single PLA (*both*). However, the delay through the single PLA was determined to be too long, and hence the PLA was manually partitioned into *cpla1* and *xcpla1* to reduce the delay. This partitioning effectively duplicated the area of the AND-plane in each of the PLA's, and involved a substantial amount of external area to route the inputs to each PLA.

The size of each PLA as implemented in the SOAR design is given in the column "rows". (Each of these has been minimized using Espresso-MV.) The result of Sasao's algorithm (using Espresso-MV as the heuristic minimizer) is given in the column "rows after output phase optimization", and the result of minimizing the complement of every function is shown in the column "rows using complement". The number of rows is seen to be less (in all cases) when the complement of every function is used, as opposed to an "optimal" choice of output phases for the outputs. Note that using the complement of all functions doesn't necessary provide the optimum phase assignment for these examples. Hence, I feel that the potential exists for better algorithms for the optimal phase assignment problem.

3.3. Input Variable Assignment and the Use of Two-Bit Decoders

Typically in a PLA, an input buffer provides the complement of each input, and buffers the normal form of the input for driving the column of the PLA. Consider the input buffers for two adjacent inputs, say a and b. These buffers generate the four logical signals $a, \bar{a}, b, \bar{b}$. In a product term the AND of those columns with a transistor is formed. There are sixteen possible ways to place the transistors in the four columns and ten different functions of two variables can be generated as shown in Table 3.2. (A 0 in

the table corresponds to a transistor, and a 1 corresponds to no transistor to be consistent with the cube representation for a product term.) The function is then AND'ed with the function formed from the rest of the variables to form a single product term. Note that seven of the arrangements of transistors result in the Boolean AND of a signal with its complement and hence is 0. This forces the entire product term to 0, and the product term contains no useful information. Therefore, only nine useful functions of a and b can be generated.

a	$\bar{a}$	b	$\bar{b}$	function	comment
1	1	1	1	1	
1	1	1	0	$\bar{b}$	
1	1	0	1	b	
1	1	0	0	0	trivial
1	0	1	1	$\bar{a}$	
1	0	1	0	$\bar{a}\,\bar{b}$	
1	0	0	1	$\bar{a}\,b$	
1	0	0	0	0	trivial
0	1	1	1	a	
0	1	1	0	$a\,\bar{b}$	
0	1	0	1	$a\,b$	
0	1	0	0	0	trivial
0	0	1	1	0	trivial
0	0	1	0	0	trivial
0	0	0	1	0	trivial
0	0	0	0	0	trivial

Table 3.2. Functions generated by normal PLA buffer.

Another possibility, however, is to generate the four possible decodes of the two variables a and b, namely, $a+b$, $a+\bar{b}$, $\bar{a}+b$, and $\bar{a}+\bar{b}$. Using these rather than using the signals and their complements it is possible to form all sixteen functions of two variables as shown in Table 3.3. For example, $ab = (\bar{a}+b)(a+\bar{b})(a+b)$.

$\bar{a}+\bar{b}$	$\bar{a}+b$	$a+\bar{b}$	$a+b$	function	comment
1	1	1	1	1	
1	1	1	0	$a+\bar{b}$	new
1	1	0	1	$a+\bar{b}$	new
1	1	0	0	a	
1	0	1	1	$\bar{a}+b$	new
1	0	1	0	b	
1	0	0	1	$a\ b + \bar{a}\ \bar{b}$	new
1	0	0	0	$a\ b$	
0	1	1	1	$\bar{a}+\bar{b}$	new
0	1	1	0	$a\ \bar{b} + \bar{a}\ b$	new
0	1	0	1	$\bar{a}\ \bar{b}$	
0	1	0	0	$a\ \bar{b}$	
0	0	1	1	$\bar{a}$	
0	0	1	0	$\bar{a}\ b$	
0	0	0	1	$\bar{a}\ \bar{b}$	
0	0	0	0	0	trivial

Table 3.3. Functions generated by two-bit decoder PLA buffer.

Note that using two-bit decoders generates the nine useful functions obtained with the normal PLA buffer as well as six new functions. This leads to the following conclusion:

Proposition 3.1: Given a PLA with one-bit decoders (normal PLA buffers) it is possible to group the inputs into pairs (in any order) and replace the input buffers with two-bit decoders to yield a bit-paired PLA with the same number of columns and no more rows (product terms) than the original PLA.

A straightforward mapping from the original PLA to a bit-paired PLA results in a PLA with the same number of rows, but there will be more transistors in each row (for n inputs, each row will contain $\frac{n}{2}$ more transistors as it takes one more transistor for every pair of variables to implement the same function after pairing). However, the benefit of bit-pairing comes from minimizing the logic function after the variables have been paired. Theorem 2.1 of [Sas84b] shows that the logic minimization problem for a given choice of pairing of variables is equivalent to a multiple-valued input logic minimization.

This can be generalized to allow forming the 2^n decodes of n variables (i.e., to pair the three variables a, b, c forming the eight decodes $a+b+c$, $a+b+\bar{c}$, $a+\bar{b}+c$,

$a+\bar{b}+\bar{c}$, $\bar{a}+b+c$, $\bar{a}+b+\bar{c}$, $\bar{a}+\bar{b}+c$, and $\bar{a}+\bar{b}+\bar{c}$), or to allow redundant pairings of variables (i.e., to pair four variables a, b, c, d as $(ab)(ac)(ad)$). In both cases, the resulting minimization problem is still equivalent to a multiple-valued Boolean minimization. However, in each of these cases, the number of columns in the PLA will increase. The resulting optimization problem of finding the optimum pairing to minimize the total area of the PLA is a very difficult one. Hereafter I consider only nonredundant pairings of two variables.

However, there is still the problem of choosing which variables should be paired together to achieve the greatest reduction in the number of terms in the PLA. There are a large number of possible pairings, as the next proposition shows:

Proposition 3.2: For a function of n variables (n even), there are

$$\prod_{i=1,odd}^{n-1} i = \frac{n!}{2^{\frac{n}{2}}\left|\frac{n}{2}\right|!}$$

ways to choose the assignment of variables to two-bit decoders. (If n is odd, add a dummy variable and consider a variable to be unpaired if it is paired with the dummy variable; hence, the number of pairs for n odd is the same as the number of pairs for $n+1$.)

Proof: To count the number of possible pairings, consider the problem of pairing n variables as one of picking the first variable and pairing it with each of the remaining $n-1$ variables, and recursively counting the number of pairings for $n-2$ variables leading to the recurrence:

$$f(n) = \begin{cases} n-1 \times f(n-2) & \text{if } n>2 \\ 1 & \text{if } n=2 \\ 0 & \text{if } n<2 \end{cases}$$

It is easy to verify that $\prod_{i=1,odd}^{n-1} i$ satisfies this recurrence.

■

One important observation is that the problems of output phase assignment and input-variable assignment are not independent. For example, consider the simple function $f = ab + cd$. There are three possible input-variable assignments: *(a b) (c d)*; *(a c) (b d)*; and *(a d) (b c)*. Each yields two product terms for f. There are two possible phase assignments: either implement the function as is with two product terms, or implement the complement of the function ($\bar{f} = \bar{a}\,\bar{c} + \bar{a}\,\bar{d} + \bar{b}\,\bar{c} + \bar{b}\,\bar{d}$) which requires four terms. Thus, performing input-variable assignment first, there is no reason to pair any variables at all. Or, by performing output-phase assignment first, it is best to implement the function rather than its complement. However, by implementing the complement of the function with the input-variable assignment of *(a b) (c d)*, the function requires only a single term.

Sasao [Sas84b] presents the following algorithm for choosing an optimal assignment of variables to the input decoders for a PLA function:

(1) Use a heuristic minimizer to obtain a minimal cover for the function without considering two-bit decoders.

(2) Determine the number of cubes that can be removed from the cover if variables i and j are paired for each pair of input variables i and j. This is done by forming the multiple-valued cover corresponding to the pairing of i and j and then either performing a distance-1 merge in the paired variable (a quick upper bound), or by actually minimizing the function after the pairing (more precise, but much more expensive).

(3) Create the *assignment graph* (a complete graph where the nodes represent input variables, and an edge between nodes i and j has weight w if w cubes can be removed from the cover if variables i and j were to be paired).

(4) Cover the assignment graph with disjoint edges so as to maximize the sum of the weights of the edges. If there are 16 or fewer inputs, it is reasonable to enumerate all possible coverings (for 16 variables, there are 2,027,025

different coverings), and choose the pairing of maximum sum; otherwise a heuristic technique (as described in [Sas84b]) can be used.

(5) Form the multiple-valued function corresponding to this pairing of the input variables, and find a minimal cover for the function.

One problem with this algorithm is that it ignores interaction between pairs. For example, if the pair (1 2) removes 5 terms, and the pair (3 4) removes 5 terms, there is no easy way to predict how many terms can be removed with the pairing (1 2) (3 4). Also, it does not consider the effect of output phase assignment on the input-variable assignment problem. Presumably, one can perform phase assignment followed by input bit-pairing, and then bit-pairing followed by phase assignment to see which yields better results for each problem.

As an example of using two-bit decoders, I consider the combined version of the control PLA's from the SOAR microprocessor mentioned earlier (*both*). Choosing the negative phase of each function gives the PLA shown in Table 3.4 with 43 rows and 686 transistors. The function is represented in standard PLA format (described in Appendix A) which uses {0, 1, -} for the binary-valued input variables, and {0, 1} for the output variables. The first step is to apply Sasao's algorithm to choose a pairing of the input variables, and then to minimize the function with this pairing. The optimal pairing chosen was (1 2) (3 7) (4 5) (6 8) (9 10), and Table 3.5 shows the direct translation of the PLA (term by term) from Table 3.4 resulting in the bit-paired multiple-valued function. Finally, Table 3.6 shows the result of the multiple-valued minimization which resulted in 36 rows and 822 transistors. In this case, the bit-pairing has increased by 136 the number of transistors in the PLA while reducing by 7 the number of rows.

input plane	output plane
-0101-001-	00000000000000000000000000000000010000
-0-0111-1-	00000000000000000000000000000000010000
-01---111-	00000000000000000000000000000000010000
--0-01101-	00010010000000000000000000000000000001
-0-1--111-	00000000100000000000000000000000010000
-0---1001-	00000000000000000000000000000000011000
--01011-1-	00010000000000000000000000100000000110
-00--1101-	00000000110000100000000000000000010000
--1--1111-	00010000000000000000000000100000000000
--01--1---	00001000000000000100000001000000000001
--1110011-	10001001010000101001000000000000000000
--100-111-	00000010100010010110000000000001100000
--1010001-	10101011011000000000100000000100000000
-------10	00000000000000000000000100001000000000
----11111-	00010010000000000110000000100001100000
---1--111-	00000010000000010110000000000001100000
--000-111-	00000001000011010010001000000001101010
--0--0-01-	00001001010000100000000000000000000000
-0--1-111-	00000000110000110000000000000000010101
--1-000-1-	00000011100001000010001000000000000010
--11-0001-	10001011110000100001100000000000000000
--0-11011-	11001001010000101001110000000010000000
--101-011-	11001011110000100000010000000100000000
--0--0011-	11001001010001101000101000000100000010
-1-------	00000000110000110000000000000000001111
---0-1101-	10001100010111100000001000100000001010
--11-1101-	10001101010111100000001000100000001011
--1-01101-	01000011100000000011110100111010000101
--0----01-	11000100000101000011110000001010000000
1--------	10110111111111010000000000000000000000
----11-0--	01000010100010000011110100011010000110
---01-0-1-	00000100000111000011101100111000000111
---11-0-1-	01000100000111000010011100111010000111
----10111-	11001110000110000111110100111010110100
--0101011-	11001101010111100111110100111010000111
-----1111-	11001100010101100001110111100110000111
----00--1-	11001100010110100001110100111010000110
----0-001-	01001110100111000011111100111010000111
---001011-	11001101010111100011111100111010000111
--11-1011-	11001111110111100011111100111010000111
-----0101-	11001101010111100111110100111010000111
-------00-	11001010101011100111111100111110000111
-------10-	11001110100111100111111100111110000111

Table 3.4. SOAR Control PLA (Using Complement of all Functions)

(1 2)	(3 7)	(4 5)	(6 8)	(9 10)	Outputs
0101	1010	0010	1100	1100	00000000000000000000000000000010000
0101	1000	1111	1010	1100	00000000000000000000000000000010000
1111	0010	0101	0100	1100	00010010000000000000000000000000001
0101	1010	1100	1010	1100	00000000100000000000000000000010000
0101	0101	1111	0100	1100	00000000000000000000000000000011000
1111	0010	0100	1100	1100	00010000000000000000000000100000110
0101	0010	1111	0100	1100	00000000110000100000000000000010000
1111	1000	1111	1000	1100	00010000000000000000000000100000000
1111	0010	1100	1111	1111	00001000000000000100000001000000001
1111	0100	1000	0010	1100	10001001010000101001000000000000000
1111	1000	0001	1010	1100	00000010100010010110000000000001100000
1111	0100	0010	0001	1100	10101011011000000000100000000100000000
1111	1111	1111	1111	0100	00000000000000000000000100001000000000
1111	1010	1010	1000	1100	00010010000000000110000000100001100000
1111	1010	1100	1010	1100	00000010000000010110000000000001100000
1111	0010	0001	1010	1100	00000001000011010010001000000001101010
1111	0011	1111	0001	1100	00001001010000100000000000000000000000
0101	1010	1010	1010	1100	00000000110000110000000000000000010101
1111	0100	0101	0011	1100	00000011100001000010001000000000000010
1111	0100	1100	0001	1100	10001011110000100001100000000000000000
1111	0001	1010	1000	1100	11001001010000101001110000000010000000
1111	0100	0010	1010	1100	11001011110000100000010000000100000000
1111	0001	1111	0010	1100	11001001010001101000101000000100000010
1010	1111	1111	1111	1111	00000000110000110000000000000000001111
1111	1010	0011	0100	1100	10001100010111100000001000010000001010
1111	1000	1100	0100	1100	10001101010111100000001000100000001011
1111	1000	0101	0100	1100	01000011100000000011110100111010000101
1111	0011	1111	0101	1100	11000100000101000011110000001010000000
1100	1111	1111	1111	1111	10110111111111010000000000000000000000
1111	1111	1010	0100	1111	01000010100010000011110100011010000110
1111	0101	0010	1111	1100	00000100000111000011101100111000000111
1111	0101	1000	1111	1100	01000100000111000010011100111010000111
1111	1010	1010	0010	1100	11001110000110000111110100111010110100
1111	0001	0100	1000	1100	11001101010111100111110100111010000111
1111	1010	1111	1000	1100	11001100010101100001110111100110000111
1111	1111	0101	0011	1100	11001100010110100001110100111010000110
1111	0101	0101	0101	1100	01001110100111000011111100111010000111
1111	0101	0001	1000	1100	11001101010111100011111100111010000111
1111	0100	1100	1000	1100	11001111110111100011111100111010000111
1111	1010	1111	0001	1100	11001101010111100111110100111010000111
1111	1111	1111	0101	0011	11001010101011100111111100111110000111
1111	1111	1111	1010	0011	11001110100111100111111100111110000111

Table 3.5. SOAR Control PLA After Setup for Two-Bit Decoders.

(1 2)	(3 7)	(4 5)	(6 8)	(9 10)	Outputs
0101	1111	0010	0100	1100	00000000000000000000000000000000010010
0101	0111	1111	0100	1100	00000000100000000000000000000000010000
0101	0010	1110	1010	1100	00000000110000100000000000000000010100
1111	0001	0101	0010	1100	00000000000000000001010000100000000001
0101	1000	1111	1010	1100	00000000110000110000000000000000010101
1111	0100	0010	0001	1100	10101011011000000000010000000100000000
1111	1111	1111	1111	0100	00000000000000000000000010000100000000
1111	0100	1000	0010	1100	11001001010000101001000000000000000000
1111	0010	1100	0100	1100	10001000010000100100000000100000000101
1111	1000	0001	1010	1100	00001010000010000111010000100000110000
1111	1010	0011	0100	1100	10001000110010100000001000000000000001000
1111	0001	1111	0001	1100	00001001010000100001010000000000000000
1111	0001	1111	0010	1100	00001001010001101000001000000000000010
1111	0100	0010	1010	1100	11001011110000100000010000000100000000
1111	0001	1010	1000	1100	11001001010000101001110000000010000000
1010	1111	1111	1111	1111	00000000110000110000000000000000000001111
1100	1111	1111	1111	1111	10110111111111010000000000000000000000000
1111	0010	0101	0100	1100	01010110000101000011110000011010000011
1111	1000	1101	0100	1100	00001011110001100011011000100000000011
1111	1001	1111	0111	1100	11000100000110000000100100011010000110 0
1111	0101	0010	1111	1100	00000100000111000011101100111000000001111
1111	0101	1000	1111	1100	00000100000111000010011100111010000111 1
1111	1111	1010	0100	1100	01000110000111000011110100011010000110 0
1111	1000	1111	1000	1100	11011110000101000111110111111011011000 10
1111	0010	1110	1000	1100	11011110000101010111101111101101100011
1111	1010	1110	0010	1100	11001110000110010111110100111010110100 1
1111	0001	0100	1000	1100	11001101010111100111110100111010000111 1
1111	0010	0001	1000	1100	11001101010111110011111111100110110111 1
1111	0101	0101	0101	1100	01001110100111000011111100111010000111 1
1111	0010	0001	0010	1100	11001101010111110011111100111010110111 1
1111	0101	0001	1001	1100	11001101010111100011111100111010000111 1
1111	1111	1111	0101	0011	11001010101011100111111100111110000111 1
1111	0100	0101	0010	1100	11001111110111100011111100111010000111 1
1111	0100	1100	1001	1100	11001111110111100011111100111010000111 1
1111	1010	1111	0001	1100	11001101010111100111110100111010000111 1
1111	1111	1111	1010	0011	11001110100111100111111100111110000111 1

Table 3.6. SOAR Control PLA After Minimization With Decoders

3.4. Optimal Encoding of the Inputs of a PLA

The *input encoding problem* can be stated as follows:

Input Encoding Problem: Given a set of symbols $S^i \equiv \{ s_1, s_2, \cdots, s_p \}$ and a Boolean function:

$$f : \{0,1\}^n \times S^i \rightarrow \{0,1,2\}^m$$

find an encoding of the symbols into binary vectors that minimizes the number of product terms needed to represent the function in two-level, sum-of-products form.

Remark: The problem with n binary inputs, 1 symbolic input and m outputs can easily be extended to consider any number of symbolic inputs.

It has been shown [De83] that this problem can be solved by performing a multiple-valued minimization of the function f (where S^i is represented by a single multiple-valued variable with p values), and then solving an encoding problem which maps the result of the multiple-valued minimization into binary vectors for each symbol. The input encoding problem has been used as an approximation to the state-assignment problem [DBS85] where the set S^i is the set of states, and the function f defines the output functions as a function of the binary inputs and the present state. Note that in this approximation, the effect of the encoding on the next-state function is ignored.

As an example of an input encoding problem, I consider the problem of optimal assignment of opcodes for a simple microprocessor. The Table 3.7 shows the decode logic for the microprocessor with the opcodes in symbolic form. There are two inputs (besides the opcode), and 5 outputs.

in_1	in_2	opcode	outputs
0	0	ADD	10101
0	1	ADD	01100
1	0	ADD	01010
1	1	ADD	10100
0	0	SUB	10111
0	1	SUB	01010
1	0	SUB	01100
1	1	SUB	10100
0	-	LOAD	11010
-	0	LOAD	01000
0	0	STORE	11100
0	1	STORE	01110
1	0	STORE	01100
1	1	STORE	01110

Table 3.7. Microprocessor Decode Logic.

Translating this into a multiple-valued minimization, the problem has four variables — the first two variables are binary-valued, the third variable has 4 values, and the fourth variable has 5 values. Translating each product term results in Table 3.8. Note that, for the binary valued variables, a 0 in Table 3.7 corresponds to 10 in Table 3.8, and a 1 in Table 3.7 corresponds to 01 in Table 3.8. Also, the four values of the third variable represent ADD, SUB, LOAD and STORE respectively.

10-10-1000-10101
10-01-1000-01100
01-10-1000-01010
01-01-1000-10100
10-10-0100-10111
10-01-0100-01010
01-10-0100-01100
01-01-0100-10100
10-11-0010-11010
11-10-0010-01000
10-10-0001-11100
10-01-0001-01110
01-10-0001-01100
01-01-0001-01110

Table 3.8. Multiple-valued version of decode logic.

The results of the multiple-valued minimization are shown in Table 3.9.

01-01-1100-10100	* (ADD, SUB)
10-10-1100-10101	* (ADD, SUB)
01-10-0101-01100	* (SUB, STORE)
11-10-0011-01000	* (LOAD, STORE)
01-10-1000-01010	
10-01-1000-01100	
10-01-0100-01000	
10-11-0100-00010	
10-11-0010-11010	
10-10-0001-10100	
11-01-0001-01110	

Table 3.9. Microprocessor Decode after Minimization.

A constraint is generated for each term with 2 or more values in the symbolic variable. The constraints (ADD, SUB), (SUB, STORE), (LOAD, STORE) can be satisfied with the embedding ADD = 01, SUB = 11, LOAD = 00, and STORE = 10. This embedding satisfies the requirement that each constraint can be represented by a single cube:

ADD, SUB	-1
SUB, STORE	1-
LOAD, STORE	-0

Minimizing with this assignment gives the PLA shown in Table 3.10. Note that the product terms are not identical in form to the multiple-valued minimization because the procedure MAKE_SPARSE has selected implicants which minimize the number of transistors in the PLA (as described in Section 4.8). If the embedding is exact in the sense that all of the constraints are satisfied, then the number of product terms should not change after the embedding is performed. To assist in analyzing this example, the differences have been noted in the table.

input	output	comments
11-1	10100	
00-1	10101	
101-	01100	
-0-0	01000	
1001	01010	
-101	00100	variable 1 raised, variable 5 lowered
01--	01000	variable 3 raised, variable 4 raised
0-11	00010	
0-00	10010	variable 5 lowered
001-	10100	variable 4 raised
-110	01110	

Table 3.10. Microprocessor Decode PLA

3.5. Optimal Encoding of the Outputs of a PLA

The last optimization problem I consider is the *output encoding problem*:

Output Encoding Problem: Given a set of symbols $S^o \equiv \{ * \} \cup \{ s_1, s_2, \cdots, s_p \}$ and a Boolean function f

$$f : \{ 0,1 \}^n \rightarrow S^o$$

find an encoding of the symbols of S^o (as binary vectors) that minimizes the number of product terms needed to represent the function in two-level, sum-of-products form. The value "*" designates input conditions for which the value of the output is a *don't-care*.

Remark 1: The extension of this problem to consider any number of symbolic outputs is straightforward.

Remark 2: The output encoding problem, while very similar in form to the input encoding problem, is a much more difficult problem. This problem has been addressed with *Symbolic Minimization* [De85] which seeks to minimize a multiple-valued input, multiple-valued output function in a code-independent manner. This is still an active area of research.

As a practical example of an output encoding problem, I consider the design of a subcircuit of a high speed division circuit [Tay81]. An n-bit divider accepts an n-bit dividend r and an n-bit divisor d and produces an n-bit quotient q. (Assume that the radix point for both dividend and divisor are to the immediate left of the numbers.) A divider is

typically built as a sequential circuit which requires n clock transitions to produce the quotient. During each clock transition, either d or $-d$ is added to r producing one bit of the quotient, d is shifted 1 position to the right, and the process is repeated n times. This technique is called *radix-2 division*.

However, a faster division circuit can be built if the dividend is shifted 2 positions each clock transition. This is referred to as radix-4 division [Tay81]. In each clock transition of radix-4 division, one of d, $2d$, $-d$, $-2d$, or 0 is added to the dividend. (Computing $\pm 2d$ is easily done with a shift of the divisor.) The divisor is then shifted 2 positions to the right, and the process is repeated $\frac{n}{2}$ times.

An important subcircuit in the design of a radix-4 divisor is the *shift-size* circuit which determines whether to add d, $2d$, $-d$, $-2d$, or 0 to the dividend. This circuit examines a fixed number of leading bits of the dividend and divisor and determines the proper value to be added to the dividend. Whether to add or subtract can be determined from the sign of the dividend, but the decision to use 0, d, or $2d$ requires a nontrivial amount of hardware. An important consideration in the design of the shift-size circuit is that many combinations of leading bits for the divisor and dividend cannot appear in any step of the division algorithm.

I concentrate now on the optimization of the shift-size circuit. The shift-size circuit is a function:

$$f : \{ 0,1 \}^n \rightarrow \{ *, Q_0, Q_1, Q_2 \}$$

where the values Q_0, Q_1, and Q_2 represent the decision to add 0, d, or $2d$ respectively based on the leading bits of the dividend and divisor. The value "*" is specified for input combinations which are known not to occur.

For the shift-size circuit considered here, there are 11 binary inputs and 3 symbolic outputs. Using a minimum bit encoding for the output, there are four values 0, 1, 2, 3 (or 00, 01, 10, 11 in binary) to assign to the three symbols. There are ${}_4P_3 = 24$ different assignments of the values to the symbols. However, not all of these result in different

minimization problems. In Table 3.11, the twelve unique encodings are enumerated. Also shown in the table are the results of an exact minimization of the function under each assignment.

Q_0	Q_1	Q_2	terms	comment
0	1	2	43	(same as 0 2 1)
0	1	3	26	(same as 0 2 3)
0	3	1	26	(same as 0 3 2)
1	0	2	26	(same as 2 0 1)
1	0	3	26	(same as 2 0 3)
1	2	0	29	(same as 2 1 0)
1	2	3	30	(same as 2 1 3)
1	3	0	27	(same as 2 3 0)
1	3	2	27	(same as 2 3 1)
3	0	1	26	(same as 3 0 2)
3	1	0	* 25	(same as 3 2 0)
3	1	2	44	(same as 3 2 1)

Table 3.11. Shift-size Circuit With Different Output Encodings.

The two output bits from this circuit are referred to as O_1 and O_2. The assignments Q_0= 00, Q_1= 01, Q_2= 10 and Q_0= 00, Q_1= 10, Q_2= 01 are equivalent because they result in a swap of the functions O_1 and O_2.

The results show that the assignment of Q_0= 11, Q_1= 01, Q_2= 00 is optimal for this circuit. Also, there is almost a 2:1 ratio in the number of terms needed to implement the function based on the encoding chosen.

There is a close relationship between output encoding and the output phase assignment problem. A simple analysis in this example shows that by enumerating all possible output encodings, we have also considered all possible phase assignments for each of the encodings. For example, the assignment Q_0= 00, Q_1= 01, Q_2= 10 with the second output complemented is equivalent to the assignment Q_0= 01, Q_1= 00, Q_2= 11.

CHAPTER 4

The Espresso-MV Minimization Algorithms

The Espresso-MV strategy for minimizing multiple-valued functions is identical to the strategy employed by Espresso-II (and Espresso-IIC) for multiple-output functions. Figure 4.1 shows an overview of the strategy. I briefly explain here the purpose of each step in the algorithm. Later in the chapter each procedure will be explained in more detail, including the extensions of the procedures for multiple-valued functions.

The first step performed by Espresso-MV is to read the function provided by the user and split the function into a cover of the ON-set, a cover of the OFF-set, and a cover of the DC-set. Espresso-MV requires all three covers. The user is allowed to specify a multiple-valued function by providing any two of these three covers, and Espresso-MV will use the COMPLEMENT procedure to compute the missing cover.

The inner loop of the Espresso-MV strategy consists of reducing the implicants to nonprime cubes, expanding the cubes to prime implicants, and extracting a minimal subset of the prime implicants. This scheme is iterated (using REDUCE) until there is no further reduction in the number of cubes in the function.

When the solution stabilizes, the LAST_GASP strategy performs the reduction and expansion in slightly different manner in an attempt to get past a local minimum.

One interesting variant added in Espresso-MV is the routine SUPER_GASP. This procedure is used optionally instead of LAST_GASP to expend more effort in finding a better solution.

Here are the main procedures employed by Espresso-MV:

COMPLEMENT Returns a representation of the complement of a multiple-valued function. This procedure is used by the setup routine to compute a cover for the ON-set, the OFF-set and the DC-set (when one of these is not

provided by the user). EXPAND is the only routine which requires the OFF-set; the remaining routines use only the ON-set and DC-set.

EXPAND — Replaces each cube in the cover F with a prime cube which covers the cube. Heuristics guide the selection of a single prime from all of the primes which cover the cube.

IRREDUNDANT — Extracts from the cover F a minimal subcover which is still sufficient to represent the function. A key component of this procedure is the routine TAUTOLOGY which tests whether a function is 1 for all possible inputs, and the routine FIND_TAUTOLOGY which returns a list of the ways that cubes can be removed from a function in order to prevent the function from being a tautology.

ESSENTIAL — Identifies which prime cubes in the cover are essential primes. An essential prime must be in any cover of the function, and hence the essential primes can be set aside before entering the iterative part of the algorithm.

REDUCE — Replaces each cube in the cover F with the smallest cube contained in the cube which is necessary to still represent the same function. The cubes are processed one at a time, and so the algorithm is sensitive to the order in which the cubes are processed.

LAST_GASP — An alternate REDUCE, EXPAND, IRREDUNDANT iteration performed in a different manner in an attempt to achieve a better solution. The step replaces each prime cube in the cover F with the maximal reduction of the cube (independent of the order in which the cubes are processed), and then these cubes are expanded in an attempt to cover other maximally reduced cubes. If any maximally reduced cubes cover other maximally reduced cubes, the resulting primes are added to the cover, followed by IRREDUNDANT to select those that are useful for reducing the size of the function.

SUPER_GASP Similar to LAST_GASP, but, instead of using EXPAND to expand the maximally reduced cubes, all prime implicants which contain each maximally reduced cube are used. IRREDUNDANT then selects a minimal subcover of this large cover of prime implicants.

MAKE_SPARSE Iterates over the cover attempting to reduce the total count of transistors needed in a PLA form of the function. The main components are LOWER_SPARSE which reduces the cubes in variables which are desired sparse, and RAISE_DENSE which expands the cubes in variables which are desired dense.

VERIFY This is used as a verification of the Espresso-MV program. When the minimization is finished, VERIFY performs a logical equivalence between the original cover and the minimized cover to verify that the function has not been corrupted. If F_{old} is the original function, F is the minimized function, and D is the don't-care set, then check that $F_{old} \subseteq F \cup D$ and $F \subseteq F_{old} \cup D$.

a

```
/* Espresso-MV — minimize a multiple-valued Boolean function

        F refers to the ON-set of the function
        D refers to the DC-set of the function
        R refers to the OFF-set of the function

        cost ( F ) first considers the number of cubes in F,
        and then the number of literals to implement F.
*/

espresso(F, D)
{
        F_old ← F;                          /* Save original cover for verification */
        R ← COMPLEMENT (F + D);             /* Compute the complement */


        F ← EXPAND (F, R);                  /* Initial expansion */
        F ← IRREDUNDANT (F, D);             /* Initial irredundant */

        E ← ESSENTIAL (F, D);               /* Detect essential primes */
        F ← F - E;                          /* Remove essentials from F */
        D ← D + E;                          /* Add essentials to D */

        do {
                φ2← cost(F);

                /* Repeat inner loop until solution becomes stable */
                do {
                        φ1← |F|;
                        F ← REDUCE (F, D);
                        F ← EXPAND (F, R);
                        F ← IRREDUNDANT (F, D);
                } while ( |F| < φ1);

                /* Perturb solution to see if we can continue to iterate */
                G ← LAST_GASP (F, D, R);

        } while (cost(F) < φ2);

        F ← F + E;                          /* Return essential to F */
        D ← D - E;

        F ← MAKE_SPARSE (F, D, R);          /* Make the solution sparse */

        if (! VERIFY (F, D, F_old))
                exit("verify error");

        return F;
}
```

Figure 4.1. The Espresso-MV main algorithm.

4.1. TAUTOLOGY

Multiple-valued tautology is an important step in many heuristic minimization algorithms [Sas84a]. In this section, we will describe the algorithm used by Espresso-MV for determining if a function is a tautology.

A well known result [BMH84, Sas84a] is the following:

Proposition 4.1.1: A cover F contains a cube c if and only if F_c is a tautology.

Hence, multiple-valued tautology can be used to determine if a cover contains a cube (i.e., the cover contains all of the minterms of the cube). This can be used to expand a cube into a prime implicant [Sas84a, Theorem 5.1], to detect redundant cubes in a cover [Sas84a, Theorem 5.2], and to detect essential primes in a cover of prime implicants [Sas84a, Theorem 5.3]. Although we choose to use the complement of the function to expand a cube into a prime implicant, multiple-valued tautology is used in Espresso-MV to extract an irredundant subcover from a cover, and to detect essential primes in a cover.

The tautology question for a multiple-valued function is NP-complete implying that there is little hope of finding a polynomial-time algorithm to solve the problem. However, in practice, we find that the run-time of the tautology algorithm accounts for only a small fraction of the time for Espresso-MV. We will use the Generalized Shannon Cofactor described in Chapter 2 to recursively divide the function into simpler functions which are examined for tautology.

Proposition 4.1.2: [Sas84a, Lemma 3.3]. If a set of cubes $c^i, i = 1 \cdots m$ satisfies $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \emptyset$ for $i \neq j$ then F is a tautology if, and only if, each of F_{c^i} is a tautology for $i = 1 \cdots m$

To reduce the complexity of answering the tautology question, we will use the properties of *weakly-unate* functions proven in Chapter 2. Using Proposition 2.6 we can always reduce the size of the problem if there are any weakly-unate variables.

4.1.1. Special Cases

Before we split the function, we first check a set of special cases:

(1) If the cover has a row of all 1's (i.e., contains a universal cube), then the function is a tautology.

(2) If the cover has a column of all 0's, then the function is not a tautology.

(3) If the function is weakly unate, then the function is not a tautology because we did not identify a row of 1's in case (1);

(4) If there are any weakly-unate variables, then cubes of F which are not full in the unate variable are discarded according to Proposition 2.6. At this point, we return to case (1) to continue checking the reduced function.

(5) If the cover H can be written as $A \cup B$ where A and B are defined over disjoint variable sets, then F is a tautology if and only if either A or B is a tautology. This case can be detected by finding a row and column permutation of F resulting in a matrix of the form:

A	1
1	B

where 1 represents an appropriately sized block of all 1's (and the division does not split a variable between the two halves). Such a partition can be easily detected with a simple greedy algorithm. However, in practice, such a decomposition may not occur often, and hence should only be checked for in the case that the matrix contains many 1's.

If none of these special cases apply, then two cubes c^1 and c^2 are chosen (as described in Section 2.3) as a partition of a heuristically selected splitting variable, and then each of F_{c1} and F_{c2} are checked recursively for tautology. The function is a tautology only if each of the two cofactors is a tautology.

4.2. COMPLEMENT

COMPLEMENT computes the complement of a multiple-valued function. In the Espresso-MV algorithms, the complement of a function is used by the EXPAND procedure. Also, COMPLEMENT is used to determine the DC-set of a function if Espresso-MV is given only the ON-set and OFF-set for the function.

The complement of a multiple-valued function is computed using the Generalized Shannon Expansion via the following proposition [Sas83, Lemma 3.2]:

Proposition 4.2.1: Let $c^i, i = 1 \cdots m$ be a set of cubes satisfying $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \emptyset$ for $i \neq j$. Then,

$$\bar{F} = \bigcup_{i=1}^{m} c_i \cap \bar{F}_{c_i}$$

In Espresso-MV, a splitting variable X_i, and a partition the values of the variable into two halves c^1 and c^2 is selected. Half of the values of X_i are placed in c^1 and the remaining half are placed in c^2. The complement of the function is computed recursively for each of F_{c1} and F_{c2}, and the complement of F is $\left(c^1 \cap \bar{F}_{c1}\right) \cup \left(c^2 \cap \bar{F}_{c2}\right)$. The procedure *complement_merge* is used to reduce the number of terms in $\bar{F}$.

4.2.1. Merging the Complement

Merging is the process of forming the union of $\bar{F}_{c1}$ and $\bar{F}_{c2}$ in such a way as to minimize the number of terms in the union. The merge step can be viewed as a heuristic minimization algorithm that attempts to minimize the number of terms in the complement of the function while the complement is being computed.

If the same cube d appears in both $\bar{F}_{c1}$ and $\bar{F}_{c2}$ then the relation

$$(c^1 \cap d) \cup (c^2 \cap d) = (c^1 \cup c^2) \cap d = d$$

replaces the two cubes with the single cube d.

An expansion of the splitting variable is also attempted using one of two algorithms:

Algorithm 1:

Check, for each cube $d \in \bar{F}_{c1}$, whether it is contained by $\bar{F}_{c2}$. If so, use the relation

$$\left[c^1 \cap d\right] \cup \left[c^2 \cap \bar{F}_{c2}\right] = \left[(c^1 \cup c^2) \cap d\right] \cup \left[c^2 \cap \bar{F}_{c2}\right]$$

to raise the values of c^2 in d (i.e., replace $c^1 \cap d$ with *supercube*$(c^1 \cap d\ ,\ c^2)$.)

The condition $d \subset \bar{F}_{c2}$ can be checked in three ways:

(a) Check if any single cube of $\bar{F}_{c2}$ contains d; if so, $d \subset \bar{F}_{c2}$. Hence, a single-cube containment check can be used although it may miss some possible lifting of parts.

(b) Determine if $\bar{F}_{c2_d}$ is a tautology. In general, the complexity of this alternative rules it out.

(c) Check if $(c^1 \cup c^2) \cap d$ does not intersect F; if this intersection is empty, then $d \subset \bar{F}_{c2}$.

The condition of Algorithm 1b and 1c is stronger than the single-cube containment of Algorithm 1a because it detects multiple-cube containment.

Algorithm 2:

Check, for each cube $d \in \bar{F}_{c1}$, whether d is distance-1 from a cube $f \in F$. If so, the parts of f which are a 1 may not be raised in d (i.e., they must remain 0). Any parts of d which are not forced to be 0 by some cube $f \in F$ may be raised.

Both of these algorithms are symmetric in that the procedure is repeated for the cubes $d \in \bar{F}_{c2}$.

Remark 1: Because the cubes have been sorted in order to remove the duplicates between the two lists, the complexity of Algorithm 1a can be reduced by roughly a factor of 2 by checking only the cubes of $\bar{F}_{c2}$ which are larger than d to see if they contain d.

Remark 2: Algorithms 1a and 1b either raise all of the parts in the splitting variable or none of the parts. (This is the same technique as used by Espresso-II for merging the results of the complement.) However, Algorithm 2 allows individual parts of a cube to be raised, and is able to determine precisely which parts can be raised, and which cannot be raised. In fact, if the cubes of $c^1 \cap \bar{F}_{c1}$ and $c^2 \cap \bar{F}_{c2}$ are prime implicants, then the cover resulting from applying Algorithm 2 will consist of prime implicants. Each leaf of the recursion in COMPLEMENT produces only prime implicants. Hence, by induction, the final cover returned by COMPLEMENT will consist of only prime implicants.

Remark 3: Algorithm 2 is using a technique similar to that used by EXPAND to determine essentially raised and essentially lowered parts (as described in Section 4.3.2), except that a cube of the OFF-set is being expanded against the cubes of the ON-set.

Algorithm 2 is a more powerful merging algorithm, and will, in general, yield a smaller representation of the complement than either Algorithm 1a or Algorithm 1b. Assuming that the complexity of Algorithm 1a is approximately $0.5|\bar{F}_{c1}||\bar{F}_{c2}|$, and that of Algorithm 2 is approximately $(|\bar{F}_{c1}|+|\bar{F}_{c2}|)|F|$, the following heuristic is used. If:

$$(|\bar{F}_{c1}|+|\bar{F}_{c2}|)|F| \leqslant (|\bar{F}_{c1}||\bar{F}_{c2}|)$$

use Algorithm 2 to raise the parts in the splitting variable; otherwise, use Algorithm 1a. Algorithm 2 is favored (by a factor of two) because it has the possibility of generating a smaller representation of the complement (which improves the performance of the EXPAND procedure).

Note that, as mentioned in Section 2.5, if the same variable is selected for splitting until all cubes in the cover are independent of that variable, then the leaves will be the functions $F_{X\{0\}}, F_{X\{1\}} \cdots F_{X\{p_{i-1}\}}$. Hence, in this case, the technique of splitting the parts in half provides a natural binary tree for performing the merge operation.

4.2.2. Special Cases

As usual, a set of special cases are checked before the function is split by the Generalized Shannon Cofactor. In the case of COMPLEMENT the special cases are:

(1) If there are no cubes in the cover (i.e., the cover is empty), then the complement is the universe; if there is row of all 1's in the cover (i.e., the cover contains a universal cube) then the complement is empty;

(2) If there is only a single cube in the cover, compute the complement using De Morgan's law as described in Chapter 2.

(3) If the matrix of F contains a column of all 0's, form the cube c which has a 0 in a column which is all 0's, and a 1 in all other positions. Then, $F = c \cap F_c$, and $\bar{F} = \bar{c} \cup \bar{F}_c$. Hence, recursively compute the complement of F_c and return the union of $\bar{F}_c$ and the complement of the single cube c.

(4) If all cubes of F depend on only a single variable, then the function is a tautology (because there were no columns of 0's detected in the previous step, the function must be a tautology if it depends on only a single variable) and hence the complement is empty.

If none of these special cases apply, the function is split into two pieces, and the complement is computed recursively.

4.3. EXPAND

The EXPAND procedure examines each cube $c \in F$ (where F is a cover of the ON-set of the binary function f) and replaces c with a prime implicant d with $c \subseteq d$. If c is not prime, then d covers more minterms of F than c does and hence is it said that c has *expanded* into a larger cube. If c is known to be prime from a previous expansion, then there is no reason to attempt to expand c. Note that each c is replaced with a single prime implicant d (out of all of the possible prime implicants which cover c) so that the number of cubes in the cover can never increase during the EXPAND step.

The goal for the minimization program is to minimize the number of cubes in F. There are several criteria that can be used in the EXPAND procedure to achieve this goal. For example, Espresso-II defines an *optimally expanded prime* as a prime d for which:

(a) d covers the largest number of cubes of F, and

(b) among all cubes d which cover the same number of cubes of F, d covers the largest number of minterms of F.

Condition (a) is a local statement of the minimization objective, and condition (b) expresses the condition that ties be broken by covering as many minterms of F as possible.

By enumerating all primes $d \supseteq c$, it is trivial to choose an optimally expanded prime to replace c. Although a technique for enumerating all of these primes is presented here, this can be prohibitively expensive. (It is possible that this would generate all of the prime implicants of the function — something clearly to be avoided.) For these reasons, Espresso-MV does not rely on generating all of these primes.

One strategy employed by some heuristic minimization programs for expanding an implicant into a prime implicant is to scan the cube from left to right and attempt to change each part of the cube which is 0 into a 1. To test whether this expansion is legal, one can test either

(1) that the ON-set of the function still covers the cube after the expansion [BaM85, Sim83], or

(2) that the expanded cube does not intersect any cube in the OFF-set of the function [Rot80].

If the expansion is legal, then the cube is expanded in the particular part. In either case, the algorithm then proceeds to the next part in the cube. The problem experienced with this simple expansion strategy is that the resulting prime implicant depends strongly on the order in which the parts are raised. We have seen examples where the final solution returned by a minimization algorithm (using this simpler heuristic for expansion) can be several times larger than the optimum solution. Also, these simpler algorithms fail to take

the most important condition (a) into account (which is to reduce the size of the cover).

MINI [HCO74] recognized the importance of choosing the order in which to expand the parts. MINI orders the variables, and then maximally expands each variable according to this ordering. The order is chosen in an attempt to expand the parts to cover other cubes of F, but this was not guaranteed.

Espresso-II and Espresso-MV expend more effort in choosing a good set of parts to raise so as to achieve the minimization objective (which is to reduce the number of cubes in the cover). In particular, Espresso-MV first guarantees that if it is possible for the cardinality of F to decrease in a single EXPAND operation, that it will. In addition, the EXPAND operation is able to consider all of the prime implicants which cover a cube.

4.3.1. EXPAND Cube Ordering

The expansion process is loosely cube-order dependent; the order in which the cubes are expanded influences the final result. The same strategy as used in MINI is used [HCO74, ORDF1-ORDF3] for ordering the cubes prior to expansion (namely, to compute a weight for each cube as the inner product of the cube with the column sums of F, and then sort the cubes into ascending order based on the weights). This heuristic attempts to expand cubes first which are unlikely to be covered by other cubes.

The cube-order dependency comes about in the heuristics which are used to expand a cube into a prime. These heuristics look to expand a cube so as to cover cubes which follow the cube in the cover (any cube which has been expanded before the current cube is already prime, and hence the current cube cannot expand so as to cover the cube). Also, if a cube becomes covered by the expansion of some earlier cube in the cover, then the cube is not expanded (because all of its minterms are already covered). Experiments have shown that the order in which the cubes are processed can affect the outcome of a single EXPAND operation, but, for the Espresso-MV running on a large set of test examples, the order in which the cubes are processed appears to matter very little. In fact, the use of a random cube-order (rather than the MINI heuristic) produced results nearly identical in both time

and optimality of solution. For this reason, the EXPAND operation is said to be loosely cube-order dependent.

4.3.2. Blocking Matrix and Covering Matrix

Espresso-II [BMH84] introduced the concepts of the *blocking matrix* and the *covering matrix*, and then used these matrices to guide the expansion of a cube into a prime. The blocking matrix is derived from the OFF-set by ensuring that each cube of the OFF-set has only a single 1 in the output part. (This operation is referred to as *unraveling* the output part.) The covering matrix is derived from the ON-set.

Espresso-MV views the problem a little differently, and uses the ON-set and OFF-set directly to guide the expansion of a cube into a prime. The actual operations performed are very similar in the case of multiple-output functions. Thus, the technique used by Espresso-MV merely provides a different way of explaining the techniques used by Espresso-II.

The blocking matrix is less convenient for the case of multiple-valued functions because the size of the blocking matrix can become very large. A direct extension of the blocking matrix to multiple-valued functions requires unraveling each multiple-valued variable (i.e., each cube in the OFF-set which depends on variable X_i to have only a single 1 in the literal of X_i). The number of rows in the blocking matrix can become very large — a single cube r of the OFF-set of an n-variable function expands into

$$\prod_{\substack{i=1,\\ r_i \neq \text{full}}}^{n} |r_i|$$

rows in the blocking matrix (where $|r_i|$ equals the number of 1's in variable i of the cube r). This is clearly unacceptable, so we seek to avoid forming the blocking matrix if possible. I present here a new explanation of why it was necessary for Espresso-II to unravel the OFF-set to form the blocking matrix, and show how Espresso-MV can avoid doing so until the very last step of the expansion process (and, in many cases, completely avoid the unraveling of the multiple-valued variables).

4.3.3. Expansion of a Single Cube

I now describe the expansion process in detail. Recall that the Boolean function being minimized is f, and a cover of the ON-set of the function is given by F. We assume we have access to a cover of the OFF-set of the function (which we call R), and that we are given a single cube $c \in F$ which we wish to expand. Initially, each part of the cube c which is not already a 1 belongs to the set of free parts which is denoted $free$. As the algorithm progresses, parts are removed from $free$, and some of these parts are added to c. The algorithm terminates when $free$ is empty, and at that point c is a prime cube. As a matter of terminology, when a part of c is changed from a 0 into a 1, the part is said to be *raised* or *expanded*.

Before proceeding, we first define two terms:

Definition: At each step of the algorithm, the *overexpanded* cube of c is the cube which results from simultaneously raising all parts of $free$. Initially, the overexpanded cube is the universe.

Definition: For any $f \in F$, the expansion of c which covers f is the smallest cube containing both f and c (i.e., $supercube(c, f)$). f is said to be *feasibly covered* if $supercube(c, f)$ is an implicant of F.

Of course, all feasibly covered cubes of F are covered by the overexpanded cube of c, but it is possible that some cube which is covered by the overexpanded cube of c may not be feasibly covered (precisely because to cover the cube would force c to intersect R). Also, initially, all parts are free so that the overexpanded cube of c is the universe. However, as parts are removed from $free$, the overexpanded cube changes reflecting that only the parts of $free$ can be raised.

Expansion Algorithm Overview:

(1) (Determination of essential parts): Determine which parts can never be raised and remove these from $free$, and determine which parts can always be raised and raise these parts of c. Exactly how this is done will be explained later.

(2) (Detection of feasibly covered cubes): If there are feasibly covered cubes in F, expand c to cover one of the feasibly covered cubes by adding parts to c and removing these parts from $free$. After each such expansion, check again for parts which can never be raised, and parts which can always be raised. Repeat Step 2 as long as there are feasibly covered cubes in F.

(3) (Expansion guided by the overexpanded cube): While there are cubes which are still covered by the overexpanded cube of c, expand c in a single part so as to overlap a maximum number of the cubes which are covered by the overexpanded cube. After expanding this part, again remove parts which can never be raised, and parts which can always be raised. Repeat Step 3 as long as there are cubes of F covered by the overexpanded cube of c.

(4) (Finding the largest prime implicant covering the cube): When there are no cubes covered by the overexpanded cube of c, map the problem of maximal expansion of c into a covering problem whereby each minimal cover of the covering problem corresponds to a prime implicant which covers c. Choose, using some heuristic technique, a small (not necessarily minimum) cover for the covering problem. This minimal cover corresponds to a large (not necessarily maximally large) prime implicant.

4.3.4. Determination of Essential Parts

This step helps us identify parts which can always be raised, parts which can never be raised, and helps us reduce F and R to just those cubes which will influence the expansion of c. The goal is to reduce the complexity of the following steps.

Proposition 4.3.1: If any cube $e \in R$ is distance 1 from c, then all of the parts of the conflicting variable which are 1 in e may never be raised in c, and any part which does not appear in any cube $r \in R$ may always be raised in c.

Proposition 4.3.2: If any cube $r \in R$ is distance 1 or more from the overexpanded cube of c, then the cube r can be removed from R while still guaranteeing that the expansion of c is an implicant of f. If any cube $f \in F$ is not covered by the overexpanded cube of c, then f is not covered by any prime containing c; hence, F can be reduced.

Therefore, Proposition 4.3.1 is used to identify parts which can never be raised and Proposition 4.3.2 is used to reduce the number of cubes of F and R which have to be considered in subsequent steps. Note that any cube which is used by Proposition 4.3.1 to force parts out of the set $free$ always satisfies the condition of Proposition 4.3.2 (after the parts are removed from the free set), and hence is immediately removed from further consideration.

After applying these two propositions, every cube of R is distance 2 or more from c, and every cube of R intersects the overexpanded cube of c. This is the equivalent to the statement that any single part of $free$ can be raised in isolation without c intersecting R, and that it is not possible to simultaneously raise all the parts of $free$.

4.3.5. Detection of Feasibly Covered Cubes

A cube is feasibly covered if c can be expanded so as to cover the cube. A test to determine whether a cube can be feasibly covered is given by the next proposition:

Proposition 4.3.3: A cube $f \in F$ is feasibly covered if, and only if, $supercube(f, c)$ is distance 1 or more from each cube of R.

Thus, each cube remaining in the cover F is tested for being feasibly covered (i.e., only the cubes of F covered by the overexpanded cube of c are checked for being feasibly covered.) To choose among the feasibly covered cubes, the feasibly covered cube which also covers the most other feasibly covered cubes is chosen. Hence, c is expanded so as to cover as many other feasibly covered cubes as possible.

After selecting a feasibly covered cube f to be covered, c is replaced with *supercube*(c, f), and the parts of f are removed from the $free$ set. Step 1 is repeated to find more essential parts, and then Step 2 (this step) is repeated to detect any more feasibly covered cubes. The algorithm proceeds to Step 3 when there are no more feasibly covered cubes.

This step allows us to guarantee that if it is possible for some expansion of a cube c to cover some other cube in F, then that expansion will be chosen and hence reduce the size of the cover.

4.3.6. Expansion Guided by the Overexpanded Cube

When there are no more feasibly covered cubes and while there are still cubes covered by the overexpanded cube of c, then we select the single part of $free$ which occurs in the most cubes which are covered by the overexpanded cube of c. We are allowed to expand c in this part because the distance between c and each cube of R is 2 or more. This has the goal of forcing c to overlap in as many parts as possible other cubes of F. After adding the part to c and removing it from $free$, Step 1 is repeated to detect essential parts and continue with Step 3 if there are cubes still covered by the overexpanded cube of c.

This is similar to the static ordering used by MINI as the main heuristic for expanding a cube into a prime implicant. The difference is that after selecting a single part to add to c, Espresso-MV follows all consequences of that selection (by finding parts which can never be raised, and parts which can always be raised after raising the single part). Then the new set of cubes which are covered by the overexpanded cube are found and another single part is selected. Thus, in some sense, Espresso-MV defines a dynamic ordering which is recomputed after each selection of a part to raise. Further, this heuristic is performed only while there are no cubes which can be completely covered, but while there are still cubes covered by the overexpanded cube of c.

One other important difference is that, with the strategy of MINI, it is not possible to reach all prime implicants containing c, even if all possible permutations of variables were to be considered. This is because MINI chooses to pick a single variable, and then expand maximally all of the parts in that variable before continuing to the next variable. Espresso-MV instead, chooses a single part of a single variable to expand and is then free to choose another part of a different variable. Therefore, Espresso-MV is able to reach all possible primes which cover the original cube.

4.3.7. Expansion Via the Minimum Covering Problem

In order for c to expand into an implicant of F, we must have that, after expanding, c be distance 1 or more from each $r^i \in R$. We can express this condition by writing a Boolean expression. We let c_j^k be a Boolean variable representing the condition that part k of variable j of an expansion of c be set to 1. Also, we let $(r^i)_j^k$ have the value of 1 if part k of variable j of the cube r^i is a 1. For any variable X_j, we can express the condition that r^i and an expansion of c be disjoint in X_j as:

$$G_{ij} = (r^i)_j^0 c_j^0 \cup (r^i)_j^1 c_j^1 \cup \cdots \cup (r^i)_j^{p_j-1} c_j^{p_j-1} = 0$$

or equivalently:

$$G_{ij} = \bigcup_{k=0}^{p_j-1} (r^i)_j^k c_j^k = 0$$

or, using De Morgan's law, as:

$$G_{ij} = \bigcap_{k=0}^{p_j-1} ((\bar{r}^i)_j^k + \bar{c}_j^k) = 1$$

We stress that the values of r^i written as $(r^i)_j^k$ are known values of either 0 or 1, and that the variables in the above equation are c_j^k.

To continue with the discussion, note that r^i and c are disjoint if they are disjoint for some variable j. This condition is written as:

$$H_i = \bigcup_{j=1}^{n} G_{ij} = 1$$

Finally, the expansion of c is disjoint from R only if it is disjoint from all cubes $r^i \in R$, and we express this as:

$$I = \bigcap_{i=1}^{|R|} H_i = 1$$

We have a Boolean expression which expresses the condition that an assignment of $\{0, 1\}$ to the variables c_j^k results in an implicant of f. We write this in full as:

$$I = \bigcap_{i=1}^{|R|} \bigcup_{j=1}^{n} \bigcap_{k=0}^{p_j-1} ((\bar{r}^i)_j^k + \bar{c}_j^k)$$

An implicant of the function I corresponds to an assignment of $\{0, 1\}$ to the variables c_j^k which results in an implicant of f. Further, a prime implicant of I corresponds to an assignment of $(0, 1)$ to the variables c_j^k which is maximal in the sense that no other variable which is 0 can be made a 1; therefore, a prime implicant of I corresponds to a prime implicant of f.

Proposition 4.3.4: I is a binary-valued unate function in the variables c_j^k.

Proof: By construction, we see that I contains only the complements of the variables c_j^k, and is therefore unate.

Proposition 4.3.5: The prime implicants of I may be obtained by expanding the product-of-sum-of-product form into a sum-of-products form, and then performing single-cube containment on the resulting cover.

Proof: By proposition 3.3.7 of [BMH84], we know that a unate, single-cube contained minimal cover is in fact the set of all primes of the unate function defined by the cover.

Thus, if all c_j^k are considered variables, Proposition 4.3.5 outlines a procedure for generating all of the prime implicants of a function f given a cover for its complement. If, instead, we set the values of c_j^k to be 1 in those places where a cube c already has a 1 (and leave the variables for c_j^k where c has a 0), Proposition 4.3.5 outlines a procedure for generating all of the prime implicants which cover a cube c.

We can also modify the expression for I using De Morgan's theorem to get the equivalent form:

$$\bar{I} = \bigcup_{i=1}^{|R|} \bigcap_{j=1}^{n} \bigcup_{k=0}^{p_j - 1} ((r^i)_j^k c_j^k)$$

Hence, we can directly write a sum-of-products expression for $\bar{I}$ and use COMPLEMENT to generate the sum-of-products form for I. We can identify the blocking matrix as proposed by Espresso-II as a representation of the Boolean function $\bar{I}$. The concept of unraveling the output part of each cube of the OFF-set in order to create the blocking matrix is equivalent to the expansion of the inner product-of-sums in the expression for $\bar{I}$ to yield a sum-of-product form for $\bar{I}$.

Thus, we have two techniques for generating all of the prime implicants of a function: one which involves repeated intersection of sum-of-products forms and one which involves the complementation of a sum-of-products form. We note here that the first formulation is equivalent to the technique outlined by Roth [Rot80] for generating all of the prime implicants of a function. As far as we know, the second technique listed here is a new formulation.

We use the form of $\bar{I}$ to discuss now how to generate the largest prime implicant which covers a cube c. Take the cover R and unravel each variable for which there is more than 1 part in the variable. (As mentioned earlier, this is equivalent to multiplying out the product-of-sums subexpression in $\bar{I}$ to get a single sum-of-products representation of $\bar{I}$.) Let us call the resulting binary matrix R'. A binary row vector x is called a cover for R' if $R' \bullet x^T \geq (1,1,\ldots,1)^T$.

Proposition 4.3.6: Each minimal cover of R' corresponds to a prime cube in the complement of $\bar{I}$, and a minimum cover of R' corresponds to a maximum prime implicant in the complement of $\bar{I}$.

Hence, we can apply a heuristic technique (to be explained in more detail in Chapter 5) to compute from R' the largest possible prime implicant which contains c.

One can reasonably ask whether it would make sense to go directly to Step 4 in the expansion of a cube to a prime implicant. In practice this approach fails because there are often several largest prime implicants, and the covering problem outlines no way to select from among the largest prime implicants. Further, quite often a smaller prime implicant may be more successful in covering other cubes of the function. It is for this reason that Espresso-MV utilizes Steps 1-3 in an attempt to cover other cubes of the ON-set before finally expanding the cube into a large prime implicant.

4.4. IRREDUNDANT

The IRREDUNDANT procedure extracts from a cover a minimal subset which is still sufficient to cover the same function. Many minimization algorithms skip this step, preferring instead to have REDUCE detect redundant cubes. However, that approach has the problem of depending on the order in which the cubes are processed. One might remove a prime implicant which is redundant, but fail to realize that, if that prime had been left in the function, several other redundant primes could have been removed instead.

As usual, we assume we have a set of cubes F which cover the ON-set of the function f, and a set of cubes D which cover the DC-set of the function f.

The cover F is first split into the relatively essential set E_r, and the relatively redundant set R_r. A cube $c \in F$ belongs to E_r if $F \cup D - c$ fails to cover c, or c belongs to R_r if $F \cup D - c$ covers c. The set E_r is relatively essential in the sense that all of the cubes of E_r must be retained in the cover in order to still cover the same function (for if any cube of E_r were removed from the cover, there would be some minterm which wouldn't be covered by the remaining cubes).

Note that any essential prime of the function must belong to the set E_r, but that the primes in E_r need not be essential primes. An essential prime of f must appear in any cover for f, whereas a relatively essential prime of F must appear in any subcover of F. (However, by starting with F as the set of all primes for f, then the set E_r consists of the set of all essential primes of f.)

The prime implicants of R_r are further divided into the totally redundant subset R_t and the partially redundant subset R_p. A cube $c \in R_r$ belongs to R_t if $E_r \cup D$ covers c, or c belongs to R_p if $E_r \cup D$ fails to cover c. The cubes of R_t are totally redundant in the sense that, because they are completely covered by the set of relatively essential primes, they can never be in a minimum subcover of F. The cubes of R_p are relatively redundant because, although any single cube of R_p can be removed, it is not possible to simultaneously remove all of the cubes of R_p while still maintaining a cover of f. Note that if F is the set of all prime implicants, then R_p can be identified as the set of primes which are dominated by the set of essential prime implicants.

What remains in R_p causes the most difficulty in trying to extract a minimum subcover of F. Imagine the following simple irredundant algorithm used by many heuristic minimizers: for each cube $c \in F$ test whether $F \cup D - c$ contains c. If so, c is redundant and is removed from F. Any time a cube of E_r is tested, the cube cannot be removed. Any time a cube of R_t is tested, the cube can always be removed (regardless of the order in which we process the cubes). However, when a cube of R_p is tested with this simple algorithm, we may or may not remove the cube depending on the order in which the cubes are tested. With this simple algorithm, at least one member of R_p will be removed, but we cannot guarantee that we will remove a maximum subset of the set R_p.

The multiple-valued tautology algorithm described earlier is used to split F into E_r, R_r, and R_p.

The Espresso-II (and Espresso-MV) techniques for extracting a maximal subset of primes from R_p is now described. Note that this algorithm becomes important only when there are three or more primes in R_p. It is not possible for there to be only one redundant cube in R_p (because the cube would be totally redundant). Also, the case where there are only two redundant cubes in R_p is uninteresting because we can always remove one cube or the other (but never both — otherwise the cubes would be totally redundant).

The key in the algorithm is a simple modification of the multiple-valued tautology algorithm. Rather than testing whether the function is a tautology, we determine which subsets of cubes in a function would have to be removed to prevent a function from becoming a tautology.

Consider forming $H = E_r \cup R_p - c$, and using the multiple-valued tautology algorithm to determine if H_c is a tautology. H_c is a tautology because every cube of R_p is covered by the union of E_r and the remaining cubes of R_p. When we get to a leaf in the tautology algorithm (i.e., when we are able to determine that the function is a tautology), we examine the cubes which are in the cover at this leaf. If there is a cube from E_r (or D) which is the universe (in this leaf), then it is not possible to avoid the function being a tautology in this leaf. Otherwise, all of the cubes of R_p which are the universe (in this leaf) must be removed in order to avoid this leaf becoming a tautology. In terms of determining how a cover covers the cube, this is equivalent to saying the cover will fail to cover the cube if and only if all of the cubes of R_p which are universal in this leaf are discarded.

In this way, a binary matrix is formed with a cube of R_p associated with each column. At each leaf which is a tautology (and for which no cube from E_r is the universal cube), we add a row to our Boolean matrix with a 1 for each column where $(R_p)^i$ is universal. A minimal cover of this Boolean matrix corresponds to a minimal subset of the primes of R_p which must be retained in the cover for f. The heuristic covering algorithm outlined in Chapter 5 will be used to select a good minimum cover of the covering matrix.

The algorithm proceeds by forming H_c for each $c \in R_p$, and calling a modified version of the TAUTOLOGY procedure called FIND_TAUTOLOGY. FIND_TAUTOLOGY returns a Boolean matrix. Note that after determining how c can be covered, c can be moved to the set E_r thus improving the performance of the algorithm (because we now know how all of the minterms of c can be covered by selecting primes from R_p).

We can relate the binary matrix formed in this way to the prime implicant table of the Quine-McCluskey algorithm for Boolean minimization. By starting with the set of all

prime implicants, the binary matrix created is a reduced form of the prime implicant table: rather than each row of the matrix corresponding to a minterm of the function, each row corresponds to a collection of minterms all of which are covered by the same set of prime implicants.

In practice, the set R_p has been observed to be small. Because the relatively essential and totally redundant sets are first identified, there is little overhead in this algorithm (compared to the simple IRREDUNDANT mentioned earlier). However, when there are partially redundant cubes, there is a much better chance of selecting a smaller subset of the partially redundant primes.

This formulation of the IRREDUNDANT algorithm, including the formation of the prime implicant table and the algorithm for finding a minimum cover for the prime implicant table, will be the basis for the exact minimization algorithm described in Chapter 5.

4.5. ESSENTIAL

Essential primes were defined in Chapter 2 as prime implicants that cover a minterm not covered by any other prime implicant. Because an essential prime implicant provides the only way of covering some minterm, all of the essential prime implicants of a function must be present in any prime cover for the function. There are efficient methods to detect those prime implicants in a cover which are essential. These essential prime implicants can be removed from the function before Espresso-MV iterates over the cover, thus providing fewer cubes which need to be processed in the inner loop. Of course, not all functions have essential primes, but experience has shown that, for most functions, it is a useful heuristic to detect and set aside the essential prime implicants.

The main theorem used for detecting which primes in a cover are essential is due to Sasao [Sas84b, Theorem A.1, Sas]:

Theorem 4.5.1: Suppose that F can be written as $G \cup p$ where p is a prime implicant of the function f, and G and p are disjoint. Then, p is an essential prime implicant of f if, and only if, p is not covered by *consensus* (G, p).

The theorem can be understood by considering the following explanation: Given a $c \in G$, the distance between c and p is at least 1. If the distance is exactly 1, then the consensus of c and p is a cube with minterms in both c and p. Hence, every minterm of p covered by $consensus(c, p)$ is covered by another prime implicant different from p. (That is, a prime implicant which covers $consensus(c, p)$ covers all of the minterms of $p \cap consensus(c, p)$ and is different from p because it contains minterms of c.) Continuing in this manner for all cubes of G, every minterm of p is covered by two or more prime implicants if and only if every minterm is covered by some cube in $consensus(G, p)$.

This theorem provides a simple test for detecting essential prime implicants in any cover:

Proposition 4.5.1: Given a cover F for the ON-set, a cover D for the DC-set of a multiple-valued function, and a prime implicant $p \in F$, form:

$$H = consensus(((F \cup D)\#p), p).$$

p is an essential prime implicant if and only if $p \not\subseteq H \cup D$.

Proof: p is to be tested as an essential prime of the function $F \cup D$. Set $G = (F \cup D)\#p$ and then $F \cup D = G \cup p$ with G and p disjoint. Hence, Theorem 4.5.1 applies and p is essential if, and only if, all of the *care* minterms of p are not covered by H.

■

Remark: The condition that all of the *care* minterms of p are not covered by H is tested by checking if $(H \cup D)_p$ is a tautology. Hence, p is an essential prime implicant if, and only if, $(H \cup D)_p$ is not a tautology.

A potential problem with this procedure is that H may contain a large number of cubes (but no more than $n\,|F \cup D|$). In practice, the performance of the tautology algorithm depends strongly on the number of cubes in the function being tested for tautology.

For each cube of $c \in F \cup D$, I review here the procedure for generating the cubes of $consensus(c\#p, p)$:

(1) If $distance(c, p) \geq 2$ or $c \subseteq p$, then $c\#p$ is empty and $consensus(\emptyset, p) \equiv \emptyset$. Hence, no cubes are generated for H.

(2) If $distance(c, p) = 1$ then $c\#p$ equals c, and a single cube results from $consensus(c, p)$. Hence, a single cube is generated for H.

(3) If $distance(c, p) = 0$, the sharp-product $c\#p$ generates one cube for every variable X_i satisfying $c_i \not\subseteq p_i$. The cube associated with such an X_i is:

$$(c\#p)_j = \begin{cases} c_j \cap \bar{p}_j & \text{if } i = j \\ c_j & \text{if } i \neq j \end{cases} \qquad (4.5.1)$$

Each of these cubes is distance 1 from c, and hence generates a cube after the consensus operation according to:

$$consensus((c\#p), p)_j = \begin{cases} (c_j \cap \bar{p}_j) \cup p_j = c_j \cup p_j & \text{if } i = j \\ c_j \cap p_j & \text{if } i \neq j \end{cases} \qquad (4.5.2)$$

Thus, when p and c intersect, as many as n cubes may be generated for H (where n is the number of variables of c).

The number of cubes generated in the case that p and c intersect can be reduced by not generating extraneous cubes which result from the binary-valued variables (i.e., variables with two parts). Assume that $c \not\subseteq p$, and consider a cube $d \in H$ which results from a binary-valued variable X_i. This cube will necessarily have $d_i \equiv 11$, and $d_j = c_j \cap p_j$ for $j \neq i$. However, p_j cannot be 11 (it must either 10 or 01 to satisfy $c_i \not\subseteq p_i$). Hence $p \cap d \subseteq c \cap p$. Thus, with respect to Proposition 4.5.1, the single cube $c \cap p$ is sufficient to replace all of the cubes which result from considering each binary-valued variables.

This result can be improved by noticing that any cube which results from a multiple-valued variable (according to equation 4.5.2) contains $c \cap p$, and hence it is not necessary to consider the binary-valued variables if any multiple-valued variable generates a cube for H.

Hence, to summarize, if c and p intersect (but $c \not\subseteq p$), a single cube is generated for each multiple-valued variable for which $c_i \not\subseteq p_i$. Then, if no cubes have been generated, the single cube $c \cap p$ is generated.

The TAUTOLOGY procedure outlined in the previous section is used to determine whether the resulting cover does indeed cover the cube c. If it does, then the prime c is nonessential. If it fails to cover the cube c, then the prime c is essential.

There are two methods for determining that a cube cannot be essential, and these are used to reduce the number of cubes which have to be checked for essentiality:

Method 1:

As outlined by [BMH84], if a cube doesn't expand to its overexpanded cube (and if it fails to cover any other cubes), then the resulting prime is nonessential. Hence, this condition is detected in EXPAND, and primes which cannot be essential primes are marked. These primes are not tested in ESSENTIAL for being essential primes.

Method 2:

By performing the IRREDUNDANT procedure before ESSENTIAL, more primes which cannot be essential primes are also detected. If a cube of F belongs to R_r, then it is completely covered by some collection of primes in F. Hence, it cannot be an essential prime. Only the primes in E_r can be essential primes. (This is equivalent to the statement that E_r contains all of the essential primes of the function.) For this reason, the ESSENTIAL operation is performed after IRREDUNDANT.

Note that the first EXPAND procedure is guaranteed to generate all essential primes of F. Hence, ESSENTIAL will detect and remove all essential primes of the function.

Finally, a comment is in order on an error in *Logic Minimization Algorithms for VLSI Synthesis*. Given their definition of consensus, Theorem 4.4.3 on page 92 does not hold for multiple-output functions, but rather, holds only for single-output functions. As shown

here, it is possible to determine if a prime of a single-output function is essential by generating at most one cube from each cube of $F \cup D$ (in the case that c intersects the prime p being tested, we need to use only $c \cap p$ rather than $consensus(c \# p, c)$). However, in the multiple-output case, when c intersects the prime p being tested, we must be careful to generate the single cube resulting from the multiple-valued consensus in the output-variable (if there is such a cube). This statement was mistakenly left out of the definition of consensus.

4.6. REDUCE

REDUCE is the step of the Espresso-II algorithm which transforms an irredundant cover of prime implicants into a new cover by replacing each prime implicant, where possible, with a smaller, nonprime implicant contained in the prime implicant. An irredundant, prime cover is a local minimum for the cost function, and REDUCE moves us away from the local minimum. The hope is that the subsequent EXPAND will determine a better set of prime implicants.

The main component of REDUCE (and both LAST_GASP and SUPER_GASP) involves the computation of the maximal reduction of a cube with respect to a cover:

Definition 4.6.1: The **maximal reduction** of a cube c with respect to a cover F is the smallest cube contained in c that can replace c in F without changing the function realized. The maximal reduction of a cube c is denoted as $\underline{c}$.

As described in MINI, the maximal reduction of a cube c with respect to a cover F and a don't-care cover D equals the supercube of $c \# (F \cup D - c)$. However, computing the reduction in this way is very inefficient.

Espresso-II uses the identity $\underline{c} = c \cap supercube(\overline{(F \cup D - c)}_c)$ to compute the maximal reduction of a cube. Hence, the operation of finding the maximal reduction of a cube can be reduced to finding the smallest cube which contains the complement of a cover. This operation is readily computed recursively using the Generalized Shannon Cofactor.

4.6.1. REDUCE Cube Ordering

Note that the reduction of a single cube depends on the form of the cover for the function. In particular, the order in which the cubes are processed for reduction affects the results of the REDUCE operation. The cubes which are reduced first will tend to reduce to smaller cubes, thus possibly preventing cubes which follow from reducing as much as they might have.

Espresso-II uses the static ordering defined by the *pseudo-distance* between each cube and the largest cube in the cover. Pseudo-distance is defined (for multiple-output cubes) as the number of variables in which the two cubes which are different (e.g., 10-01-11-01-011 and 10-11-01-01-111 have a pseudo distance of 3). MINI uses the reverse order of the EXPAND ordering. (Recall from Section 4.3 that the MINI ordering for EXPAND weights each cube according to how many other cubes have a 1 in the same parts as the cube.) All of these heuristic ordering strategies attempt to place cubes which are the most likely to reduce (i.e., either "large" cubes, or cubes which have parts covered by many other cubes) near the top of the list.

Experiments were performed for these REDUCE ordering strategies and also using a random permutation of the cubes. It was discovered that the solution returned for a particular execution of REDUCE varied, but did not favor any particular ordering over the random permutation. More importantly, the final solution returned from the Espresso-MV algorithm was not sensitive to the ordering in REDUCE. I feel this is due to both the iterative nature of the Espresso-II algorithm (if a cube is ordered such that it fails to reduce, it may reduce on a subsequent iteration), and the LAST_GASP strategy successfully removing the cube-order dependency of REDUCE. Hence, the actual choice of cube ordering is not believed to be critical.

In Espresso-MV we choose to alternate between the MINI strategy and a strategy which places the largest cube on the top of the list, and orders the remaining cubes by increasing distance from the largest cube. Alternating these two strategies produced con-

sistently the same or better results for Espresso-MV than any single heuristic. We speculate that this is because, if the same ordering is used for every iteration, that the same cubes will tend to be reduced first. By mixing the strategies, very different orderings result allowing for exploring a wider range of expansions.

4.6.2. Computing the Supercube of the Complement

The Generalized Shannon Cofactor is used to recursively compute the supercube of the complement (i.e., the smallest cube containing the complement) of a function according to the next two propositions:

Proposition 4.6.1: If a set of cubes $c^i, i = 1 \cdots m$ satisfies $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \emptyset$ for $i \neq j$, then

$$supercube(\bar{F}) = supercube\left(\bigcup_{i=1}^{m} c^i \cap supercube(\bar{F}_{c^i})\right)$$

Proof: Using Proposition 2.3:

$$\bar{F} = \bigcup_{i=1}^{m} c^i \cap \bar{F}_{c^i}$$

to show

$$supercube(\bar{F}) = supercube\left(\bigcup_{i=1}^{m} c^i \cap \bar{F}_{c^i}\right)$$

Given that $supercube(c^i \cap \bar{F}_{c^i}) = c^i \cap supercube(\bar{F}_{c^i})$, we see the proposition holds.

■

This recursion naturally terminates when F_{c^i} becomes a single cube where the following test is applied:

Proposition 4.6.2: Given a cube c:

$$supercube(\bar{c}) = \begin{cases} \emptyset & \text{if } c \textit{ depends on no variables} \\ \bar{c} & \text{if } c \textit{ depends on one variable} \\ \textbf{universe} & \text{if } c \textit{ depends on two or more variables} \end{cases}$$

Remark: If c depends on only the variable X_i, then $\bar{c}$ is a single cube resulting from the bit-wise complement of c_i.

Proof: Trivial if one considers computing the complement of a cube using De Morgan's law. If the cube depends on more than two variables, then the complement contains more than two cubes. Each of these cubes depends on only a single variable (with the remaining literals all full), and hence the supercube of these cubes is the universe. If the cube depends on only a single variable, there is only one cube in the complement. Finally, if the cube is the universe, the complement is empty.

However, there is also the following more powerful result:

Proposition 4.6.3: If F is a weakly unate cover and F^i represents the i^{th} cube in the cover, then:

$$supercube(\bar{F}) = \bigcap_{i=1}^{|F|} supercube(\bar{F}^i)$$

Thus, if the cover is weakly unate, this result is applied to quickly determine the supercube of the complement of a cover. Further, only the cubes of the weakly unate cover which depend on a single variable need be considered (assuming the cover does not contain a universal cube), because the supercube of the complement of any cube which depends on two or more variables is the universe and hence does not affect the intersection.

There are two other results (easily derived from De Morgan's law) which can be useful in reducing the amount of work necessary to compute the supercube of the complement of a function.

Proposition 4.6.4: If the cover F contains a column of 0's, form the cube c which has a 0 in each position where F has a column of all 0's, and 1 elsewhere. Then, from the identity $F = c \cap F_c$, is is seen that

$$supercube(\bar{F}) = supercube(supercube(F_{\bar{c}}), supercube(\bar{c})).$$

Hence, if there is a column of 0's in the matrix for F, this proposition is applied to compute $supercube(\bar{F})$. In particular, if F has a column of 0's in two separate variables,

then it is immediately determined that $supercube(\bar{F}) \equiv$ **universe.**

Proposition 4.6.5: If F can be factored into the form $F = A \cup B$ where A and B are over disjoint variable sets, then

$$supercube(\bar{F}) = supercube(\bar{A}) \cap supercube(\bar{B})$$

Detecting such a partition of H corresponds to finding a row and column permutation resulting in the form:

A	1
1	B

where 1 represents an appropriately sized block of all 1's (and the division does not split a variable between the two halves). As in the case of tautology, such a partition is easily determined with a simple greedy strategy. In practice, such a decomposition may not be common, and should only be checked for when the matrix contains many 1's.

4.6.3. Choice of Splitting Variable

It would be desirable to choose the cubes $c_1, c_2, \cdots, c_m$ so that the resulting cofactors quickly become *weakly unate*. However, it is not clear how to efficiently choose a splitting variable and a partition of that variable so as to achieve this goal. In Espresso-MV the simple strategy outlined in Section 2.5 is used when choosing the cubes for partitioning.

4.7. LAST_GASP and SUPER_GASP

The basic iteration of Espresso-II (REDUCE, EXPAND, IRREDUNDANT) faces the following obstacles: (1) The EXPAND step uses heuristics to choose one prime implicant (from all of the prime implicants which cover a cube) to replace each cube in the cover; and (2) the REDUCE algorithm is cube-order dependent so that cubes which are reduced first tend to reduce more than cubes which are reduced later. Different minimization algorithms have managed these problems in different ways. For example, MINI uses the *reshape* operation in order to sidestep these problems, and Prestol-II uses the *change_shape* opera-

tion (twice in succession) in order to escape these problems. I describe here the Espresso-II strategy LAST_GASP and the Espresso-MV strategy SUPER_GASP for improving the basic minimization algorithm.

4.7.1. LAST_GASP

This algorithm first computes the maximal reduction of every cube of the ON-set cover F and creates a new cover G. If a cube cannot be reduced it is ignored. A modified version of the EXPAND algorithm expands each of the cubes of G. The EXPAND procedure is modified so that: (1) the expansion of a cube is stopped as soon as it is determined that it cannot cover any other cubes; the cube is removed from G in the case that it cannot expand to cover any other cubes; and (2) all of the cubes are expanded even if they are covered by the expansion of a different cube. As shown in [BMH84], those cubes that succeed in covering some other reduced cube are potentially useful primes for reducing the cardinality of the cover. These new primes are simply added to the cover F, and the IRREDUNDANT procedure then extracts a minimal subcover. Because the number of reduced cubes which can expand to cover other reduced cubes tends to be very small, this technique is applicable to a wide range of problems. In particular, I have not found any examples for which the running time of the algorithm is dominated by the LAST_GASP operation.

4.7.2. SUPER_GASP

Espresso-MV also has an optional routine SUPER_GASP. This algorithm computes the maximal reduction of each cube of the cover F and then generates *all* of the prime implicants which cover the cube (rather than only a single prime implicant which covers the cube). In order to generate all of the prime implicants which cover a cube, the algorithm given in Section 4.3 (EXPAND) is used. By sorting this set of prime implicants, duplicate prime implicants are easily detected. IRREDUNDANT then extracts a minimal subcover from the remaining set of prime implicants. Note that if IRREDUNDANT returns the minimum number of cubes necessary to implement the function, then no single

iteration of REDUCE, EXPAND, and IRREDUNDANT can do any better from the same starting point.

Of course, the process of generating all of the primes which cover the maximally reduced cubes may greatly expand the size of the cover. (In particular, if the original cover were all minterms, the generation of all of the primes covering each minterm would be an inefficient way to generate all of the primes for the function.) The program Espresso-MV is careful to terminate the generation of all of the primes in the case there are too many primes, in which case the LAST_GASP strategy is used instead. In practice, the SUPER_GASP can be selected optionally when the program Espresso-MV is run. In Chapter 6, I report experimental results with this option.

4.8. MAKE_SPARSE

When the outer loop of the Espresso-MV algorithm terminates, the solution consists of an irredundant cover of prime implicants which represents the original function. However, depending on the final implementation of the multiple-valued function, we may desire a final cover which does not necessarily consist of prime implicants. One goal is to reduce the number of transistors needs to implement each literal of a cube. This depends on the number of 0's and 1's in the literal, but it also depends on the type of variable as shown in Table 4.8.1:

Variable Type	Number of transistors	Comment
binary-valued variable	count number number of zeros	sparse
multiple-valued variable (for a two-bit decoder)	count number of zeros	sparse
multiple-valued variable (for the output part)	count number of ones	dense
multiple-valued variable (for the input encoding problem)	count number of ones (unless literal is full)	dense

Table 4.8.1. Transistors per Literal in a PLA

For example, if the function being minimized represents a two-level multiple-output PLA function, then each 0 in the cube for a binary-valued variable corresponds to a

transistor in the AND-plane of the PLA, but each 1 in the multiple-valued output variable corresponds to a transistor in the OR-plane of the PLA.

Another example is minimizing a multiple-valued function for the state-assignment program **KISS**. For these functions, it is preferred that the multiple-valued variables have as few 1's as necessary (which will lead to fewer constraints for the embedding problem).

Hence, the binary-valued variables and multiple-valued variables resulting from a bit-paired PLA are desired to be **dense** (i.e., have many 1's), and the multiple-valued variable resulting from the output-part of a PLA are desired to be **sparse** (i.e., have few 1's). Finally, the multiple-valued variables resulting from a symbolic variable (as in **KISS**) should be sparse unless the cube does not depend on this particular variable. With these observations we define, for each variable, whether the variable is to be a **sparse variable** or a **dense variable**. The MAKE_SPARSE procedure then attempts to satisfy these goals.

MAKE_SPARSE consists of two steps: LOWER_SPARSE removes redundant parts from the sparse variables and RAISE_DENSE attempts to add parts to the dense variables (which may be possible following LOWER_SPARSE because the cubes are no longer prime implicants). These two algorithms are iterated until there is no more reduction of any sparse variable, or until there is no more expansion of any dense variable. This algorithm is iterated in Espresso-MV (as opposed to Espresso-II which only executed each step once) because the total literal reduction is worth the extra expense.

During the first iteration of LOWER_SPARSE and RAISE_DENSE the cardinality of the cover cannot decrease (because the cover is an irredundant, and consists of prime implicants). However, in extreme cases, it is possible for the cardinality to decrease in subsequent iterations. In fact, the procedure MAKE_SPARSE can be viewed as a complete minimization algorithm. (The **pop** program from Berkeley [Sim83] uses essentially this simple algorithm, but without the powerful techniques for each of the basic steps as in MAKE_SPARSE. However, this minimization algorithm is restricted in the size of the set of prime implicants which it can explore.)

In the discussion that follows, we assume, as usual, that F is a cover for the ON-set, D is a cover for the DC-set and R is a cover for the OFF-set.

4.8.1. LOWER_SPARSE — Reduce the Sparse Variables

The goal of LOWER_SPARSE is to remove parts from the sparse variables so as to reduce (if possible) the number of 1's in these variables for each cube. This procedure can be viewed as cube reduction applied to each cube with the reduction retained only for the the multiple-valued variables. However, this technique suffers from the same problem as REDUCE, namely that the order in which the cubes are processed can greatly affect the total amount of reduction possible.

Instead, the IRREDUNDANT routine is used to select, for a particular part, which cubes are redundant; this part is set to 0 for the redundant cubes. This way the cube ordering problem is avoided, and the more powerful heuristics of IRREDUNDANT are used to find a good reduction of the sparse variables.

For each value j of a sparse variable X_i, define e_j^i to be the cube of $X_i^{|j|}$. By finding an irredundant cover for $(F \cup D)_{e_j^i}$ we can determine which cubes of F can have part j removed. If a cube does not belong to the irredundant subcover of $(F \cup D)_{e_j^i}$, then the part in the cube is redundant and can be removed. These parts are removed, and, after all parts for a variable have been processed, the next variable is processed.

Note that by using the IRREDUNDANT algorithm rather than REDUCE, the order in which the cubes are examined in part j of variable X_i is immaterial. (Further, the order in which the parts of any variable is processed is also immaterial.) But, the order in which the sparse variables are processed does influence the reduction of variables which are not processed first. In Espresso-MV, LOWER_SPARSE is applied to sparse variables corresponding to multiple-valued variables resulting from the input-encoding problem. This is done to simplify the constraints which arise from the multiple-valued parts. The last variable processed is the multiple-output variable. Admittedly, this heuristic is a little crude.

4.8.2. RAISE_BV — Expand the Dense Variables

As mentioned earlier, we desire that the binary-valued variables, and the variables resulting from bit-pairing be dense. After reducing the multiple-valued variables with LOWER_SPARSE, the resulting set of cube is no longer prime. Hence, we can try to expand this set of cubes by expanding only the dense parts of each cube. This is done with a modified version of EXPAND which removes all of the sparse parts from the *free* set (cf. sec 4.3) before finding the expansion of a cube. Hence, none of the sparse parts will be expanded.

Interestingly, EXPAND will still check for cubes which, when limited to only the dense variables, can expand to cover another cube. As mentioned earlier, on subsequent iterations of MAKE_SPARSE it is possible for the cardinality of the cover to decrease. If it is possible for a cube to be covered, EXPAND will expand the dense variables so as to cover the cube.

CHAPTER 5

Exact Boolean Minimization

Two methods for generating all of the prime implicants of a Boolean function were presented in Section 4.3 (EXPAND), and in Section 4.4 (IRREDUNDANT) an algorithm for efficiently generating the prime implicant table of Quine and McCluskey was presented. Generating the set of all prime implicants, using IRREDUNDANT to generate the prime implicant table, and then solving the covering problem for this table provides an algorithm for determining the minimum solution for a given minimization problem.

In this chapter, a new set of heuristics for guiding a branch and bound solution to the covering problem is presented. These heuristics have been used to solve many large covering problems resulting from Boolean minimization problems. A new approximate algorithm of polynomial complexity (based on these heuristics without any backtracking) which is more practical for heuristic minimization programs is also presented. This approximate algorithm also has the advantage of providing a lower-bound on the cardinality of the exact solution, and hence can sometimes determine that the solution provided is in fact optimum.

5.1. Minimum Cover Problem in Espresso-MV

Recall that the minimum covering problem appears in Espresso-MV in two ways:

(1) During IRREDUNDANT when there are partially redundant cubes in the cover, the problem is translated (via the Generalized Shannon Cofactor) into an equivalent covering problem. A minimal solution to this covering problem corresponds to discarding a maximal subset of the partially redundant set. (Also, LOWER_SPARSE uses IRREDUNDANT to remove redundant parts from the sparse variables).

(2) During EXPAND, when there is no longer any way to obviously guide the expansion of a cube into a prime implicant, the problem of expanding the cube into the largest possible prime implicant (the prime implicant covering a maximal number of minterms) is translated into a covering problem. The solution to this covering problem determines how the cube should expand.

5.2. Minimum Cover Problem

Minimum Covering Problem: Given a binary matrix A, and a cost $cost(\cdot)$ for each column of the matrix, find a vector x such that $A \bullet x^T \geq (1,1,....,1)^T$ and $\sum_{i=1}^{m} x_i cost(i)$ is minimum.

The constraint $A \bullet x^T \geq (1,1,....,1)^T$ can be understood as saying that each row of the matrix must have at least one 1 in some column where x has a 1. (In this case, the row is said to be "covered" by the particular "column" of x, and the goal is to cover all rows with a vector of minimum weight.) This problem is NP-hard [GaJ79] so that any algorithm which solves the problem can be expected to have a bad worst-case complexity.

In this chapter, a cost function of 1 for each column of the matrix is used to simplify the explanation. In Section 5.8, the extensions of the algorithm presented here to a more general cost function are considered.

5.3. Reducing the Size of the Covering Problem

First, I review some results which are of interest in reducing the size of a covering problem:

(1) **Partitioning**: If the rows and columns of matrix A can be permuted to yield a block structure of the form:

A	0
0	B

where 0 represents an appropriately sized block of all zeros, then a minimum cover for A can be written as the union of a minimum cover for A, and a minimum cover for B.

(2) **Essential Elements**: Any row of the matrix A which has only a single 1 identifies an essential column. The solution vector x must have a 1 in the essential column in

order to cover the row singleton. After placing a 1 in the essential column, any other rows which become covered can be removed from consideration.

(3) **Row Dominance**: If row i of A contains another row j of A (i.e., row i contains a 1 for all columns in which row j has a 1), then row i can be removed from the matrix A without changing the minimum solution. Clearly, once row j has been covered, then row i will automatically also be covered, and hence row i is providing redundant information in the covering problem.

(4) **Column dominance**: If column i of A contains another column j of A (i.e., column i contains a 1 for all rows in which column j contains a 1), then column j can be removed from the matrix A without changing the minimum solution. Clearly, there could be no advantage to choosing column j because choosing column i instead would cover the same set of rows, and perhaps more. Hence, column j is not needed for a minimum solution.

Therefore, the strategy to reduce the size of the matrix is:

(1) Look for a block partitioning.

(2) Use row dominance and column dominance to reduce the number of rows and columns in the matrix. Note that it is only necessary to apply either transformation once, and the order in which they are applied is irrelevant.

(3) Identify essential elements and add them to the covering set. The rows which are now covered and the essential columns are removed from the matrix.

(4) Repeat Steps (2)-(4) until no essential elements are detected in Step (3).

After using Steps (1)-(4) to reduce the size of the matrix, if a solution has not been reached, an element is selected for branching. The problem is then solved recursively assuming the branching element is in the solution, and then assuming the branching element is not in the solution.

The branch and bound algorithm for solving this problem is shown on the next page. The routine is entered at the top level with: the matrix (A) to be covered, a current solution (x) which is initially the empty set, a record (*best*) of the best solution known to be a cover (which is initially a full set), a lower bound (*best_possible*) on the size of the best solution (which is initially ∞), and an indication *level* of the current level in the recursion (which is initially 0). The routine returns a set of the columns of A which is a minimum cover for A.

```
bit_vector minimum_cover(A, x, best, level)
bit_matrix A;                          /* the matrix to be covered */
bit_vector x;                          /* the current solution */
bit_vector best;                       /* the best solution seen so far */
int best_possible;                /* the best solution possible */
int level;                             /* recursion level */
{
        if (partition(A , H1, H2)) {          /* check for block partition */
                x1← minimum_cover(H1, Ø, Ø, best_possible, 0);
                x1← minimum_cover(H2, Ø, Ø, best_possible, 0);
                return x1∪x2;
        }

        do {
                /* reduce the number of rows and columns */
                A ← remove_row_dominance(A);
                A ← remove_column_dominance(A);

                /* Select essentials, and remove rows covered by an essential */
                p ← detect_essential(A);
                x ← x ∪p ;
                A ←reduce(A, p);
        } while (p ≠Ø);

        independent_set ← maximal_independent_set(A);
        if (level == 0)
                best_possible ← |independent_set| ;

        /* if current solution exceeds the best possible from here on, bound the search */
        if ( |x ∪independent_set| ≥ |best| )
                return best ;

        /* if no rows left in A, then new best solution */
        else if (numrows(A) == 0)
                return x ;

        /* Else branch on some column */
        else {
                q ← select_column(A, independent_set);
                /* recur assuming q belongs to the minimum cover */
                left ←minimum_cover(reduce(A , q ), x ∪q , best , best_possible , level +1);
                if ( |left| < |best| )
                        best = left;
                if ( |best_possible| = |best| )
                        return best;

                /* recur assuming q does not belong to the minimum cover */
                right ←minimum_cover(remove(A , q ), x , best , best_possible , level +1);
                if ( |right| < |best| )
                        best = right;

                return best;
        }
}
```

The routines **remove_row_dominance** and **remove_column_dominance** apply row and column dominance to A to reduce its size. The routine **detect_essential** detects rows with only a single 1, and these are added to the selected set. The function **select_column** applies heuristics to select a column of A for branching. The function **reduce** removes those rows of A which are covered by q and removes the column q, and the function **remove**(A, q) deletes the column q from A.

First a check is made for a simple partition of the covering problem. If this fails, row and column dominance are applied iteratively to reduce the size of the covering problem, and then the essential elements are detected and added to the selected set. Then, using a technique described in the next section, a lower bound is placed on the size of the cover for A, and the search is terminated (or bounded) if the size of the selected set exceeds the best solution possible for A. If there are no more rows in A, then we have reached a new best solution, and the solution is returned. Otherwise, a column is selected heuristically to branch on and recursively compute the solution assuming that the element is in the covering set, and then assuming that the element is not in the covering set.

5.4. Use of the Maximum Independent Set

The most important feature of the above algorithm is in the routine **maximal_independent_set**. This routine finds a maximal set of rows of A all of which are pairwise disjoint (i.e., they do not have 1's in the same column). It should be clear that the number of rows in this independent set is a lower bound on the solution to the covering problem, because a different element must be selected from each of the independent rows in order to cover these rows. Hence, this lower bound can be used to terminate the search if the size of the current solution plus the size of the independent set is greater or equal to the best solution seen so far. Also, the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover. Hence, by recording this value, the search can be terminated if a solution is found which meets this lower bound.

The major drawback of this technique, of course, is that the problem of finding a maximum independent set of rows is itself an NP-hard problem. But this is of no concern. The problem of finding a maximal independent set of rows can be solved heuristically while still providing a correct lower bound on the size of the final solution. (In general, finding the maximum independent set provides the best bound; other minimal solutions provide less precise, but, nonetheless, accurate lower bounds.) Hence, even though this problem is itself difficult, a good, heuristic algorithm is sufficient for finding a maximal independent set of rows.

To find a large independent set of rows, a graph is constructed where the nodes correspond to rows in the matrix, and an edge is placed between two nodes if the two rows are disjoint. The problem is now equivalent to finding a maximal clique (a maximal, completely connected subgraph) of this graph. To solve this problem, a greedy algorithm is used:

(1) Initialize the clique to be empty (contains no nodes);

(2) Pick the node of largest degree (and not already in the current clique), and add this node to the clique. Break ties by choosing the node which is connected to the most other nodes of maximum degree;

(3) Remove all nodes and their edges from the graph which are not connected to the current clique;

(4) Repeat Steps 1 and 2 while there are still nodes in the graph not in the current clique.

The node of largest degree in Step 2 corresponds to the row which is disjoint with the maximum number of other rows of the matrix. The tie-breaker attempts to preserve as many of the remaining nodes of maximum degree as possible.

Thus, the bounding in the branch and bound algorithm is modified by bounding the search if $|$**maximal_independent_set**$(A) \cup x|$ equals or exceeds the best known solution (rather than waiting until $|x|$ equals or exceeds the best known solution.) The goal is to terminate unprofitable searches as early as possible.

Besides the fact that the problem of finding a maximum independent set of rows is NP-hard, there is the further difficulty that the bound provided by the maximum independent set may not be sharp. For example, consider the matrix:

$$\begin{matrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{matrix}$$

A maximum independent set of rows for this matrix contains only a single row, but a minimum cover requires at least two columns. The size of the maximum independent set remains a lower bound on the size of a minimum cover; the search may just not be terminated as early as possible.

5.5. Choice of Branching Column

A unique element from each set of the independent set of rows must be in the minimum solution. Once a maximal independent set of rows has been computed, the selection of a branching element is limited to some element which belongs to one of these rows. Each element of each row is given a weight as the reciprocal of the row sum. Then the weights are summed for each column, and the column of maximum weight which is also in the independent set of rows is chosen for the branching variable. This weighting strategy gives the elements of the smaller sets a higher weight. For example, in a set with 2 elements, each element receives a weight of 0.5, whereas in a set with 10 elements, each element receives a weight of 0.1. The larger sets are thought of as "easier" to cover, and the smaller sets are "harder" to cover. The heuristic is to try to force a selection from one of the smaller sets. Another reason for favoring choosing an element from a smaller set (for example, a set with two elements) is to create more essential elements at the next step of the recursion.

5.6. Heuristic Covering Algorithm

The heuristic covering algorithm used in Espresso-MV is based on the above algorithm for the minimum covering problem. In order to make the running time more

predictable, the algorithm is converted into a greedy algorithm in which the first leaf visited is taken as the solution and no backtracking is performed. Note that this greedy algorithm has the nice property that it can compute a lower bound on the size of a minimum cover (even though it is not guaranteed to generate a minimum cover). (Recall that the size of the maximal independent set of rows at the first level of the recursion is a lower bound for the minimum solution to the covering problem.) Hence, sometimes this greedy algorithm is able to demonstrate that it has achieved a minimum solution.

5.7. Implementation

The matrix A is stored as a fully packed bit-matrix. Each row occupies a number of consecutive words, and each bit in the word is set to either 0 or 1.

The algorithm, as described above, is recursive. At the top level, the maximal independent set determines a lower bound on the final solution. This is recorded, and if the lower bound is ever achieved, the branch and bound is terminated.

The first step is to determine if the matrix has a block partition. If so, the matrix is split into two parts, and the algorithm is recursively entered at the top level.

Row dominance is detected by first sorting the rows of the matrix using an $O(n \log n)$ sorting algorithm. The rows are sorted into ascending order based on the number of 1's in the row; two rows with the same number of 1's are sorted into lexicographical order. Equal rows (a special case of row dominance) are then easily detected and removed. Because duplicate rows have been removed, a row can only dominate another row if it has strictly fewer 1's; hence, to determine if a row is dominated, it is only necessary to compare it against rows which precede it in the sorted matrix.

Column dominance is slightly more difficult because of the row-oriented structure of the bit-matrix. The matrix is first transposed so that all column operations become row operations, and then the matrix is sorted as described above. Then containment is performed on the columns in a similar manner to the row containment described above. Finally, the matrix is transposed a second time to restore it to its proper shape.

The maximal independent set graph (G) is symmetric, and is most easily represented by a fully-packed binary adjacency matrix. The matrix is generated by intersecting each pair of rows of the matrix, and inserting a 1 into position G_{ij} if the rows are disjoint.

5.8. Extension to a General Cost Function

The branch and bound algorithm presented here can also be extended to treat the more general case of an arbitrary cost function $c(\cdot)$ defined for each column.

Row dominance remains a valid technique to reduce the number of rows in the matrix, and essential columns must still be in a minimum cover. However, if column i contains column j, then column j can be deleted only if the cost of column j is the same or more than the cost of column i.

The major extensions to the covering algorithm depend mostly on how to interpret the maximal independent set for the purpose of bounding the search. The bound on a minimum cost solution is given by the cost of the current solution plus the cost of the element of least cost in each row of the set of independent rows.

CHAPTER 6

Experimental Results

In this chapter I report results from an implementation of the Espresso-MV algorithms. The Berkeley PLA test set includes a large collection of PLA's and a smaller collection of multiple-valued logic functions. I present results from the program Espresso-MV (in both its heuristic and exact modes) for all examples in the test set and compare the results to the exact minimizer McBoole [DAR86], and to the heuristic minimizer Presto1-II [BaM85]. For the multiple-valued minimization problems, I present results for Espresso-MV minimizing these problems as a binary-valued minimizer with an appropriate don't care set, and as a multiple-valued minimizer. Unfortunately, I do not have access to other multiple-valued minimization programs for comparison.

6.1. Espresso-MV

The program Espresso-MV implements the heuristic and exact logic minimization algorithms described earlier, as well as heuristic and exhaustive algorithms for the *output phase assignment* and the *input variable assignment* problems. The program can also be used for manipulating multiple-valued logic functions. Espresso-MV will (1) compute the intersection, union, or sharp-product between two logic functions; (2) verify the logical consistency of two logic functions; (3) compute the complement; (4) compute the set of all prime implicants; (5) check the logical consistency of a single logic function. The use of the program (including the input and output file formats) is documented in Appendix A. Espresso-MV is written in the C language and is about 10,000 source lines. The program as written fits into the UNIX environment as a filter (reading a logic function or logic functions from standard input, and writing the logic functions to standard output).

The command line option *-do exact* selects the exact minimization algorithm of Espresso-MV. This is referred to as Espresso-MV in the exact mode. Likewise, the

command line option *-strong* uses the SUPER_GASP heuristic described in Section 4.7, and is referred to as Espresso-MV in the strong mode.

6.2. The PLA Test Set

When research leading to the Espresso-II algorithms began, PLA examples were collected as a vehicle for comparing different minimization algorithms. By the time the book *Logic Minimization Algorithms for VLSI Synthesis* was written, 56 PLA examples had been collected. Further donations to the test set from industry and Universities has expanded the test set to 134 functions. Of these, 111 are designated as industrial examples (implying that their origin is either an industrial or University chip design), and 23 are mathematical functions such as multiply and square root. Included in the test set are 11 randomly generated examples given to us by the authors of Prestol-II. Because the random examples exhibit behavior which is much different from the industrial examples, they are reported in a separate section. Tables 6.1 and 6.2 show the raw data for Espresso-MV in its normal, strong, and exact modes and raw data for McBoole and Prestol-II when such data is available. (This raw data is summarized in the text.)

The complete test set presented here is available from the Industrial Support Office, 461 Cory Hall, University of California, Berkeley, CA 94720.

6.2.1. Grading the Test Set by Problem Difficulty

With a test set so large, it is a challenge to present the results from competing algorithms in a meaningful manner. It can be misleading to merely report the total number of cubes and total number of literals for each algorithm and then attempt to draw conclusions from these totals. Hence, my first goal is to determine the difficulty of the minimization problem for each PLA in the test set.

For each problem in the test set, I first classify the problem as one of the following:

Classification	Description
trivial	minimum solution consists of essential prime implicants
noncyclic	the covering problem contains no cyclic constraints
cyclic and solved	the covering problem contains cyclic constraints and the minimum solution is known
cyclic and unsolved	the covering problem contains cyclic constraints but the minimum solution is unknown
too many primes	there were too many primes to be enumerated

Table 6.1. PLA Classification by Degree of Difficulty.

The classifications were determined by allowing the exact minimization algorithm of Espresso-MV and the exact minimization algorithm of McBoole to run for 5 hours for each example on an Apollo DN660[1]. (If a program had not terminated after 5 hours, it was aborted). By examining the results for each program, a classification is determined for each example. If the problem was solved by either of the two exact minimization algorithms, it is easy to decide whether it belongs to the class **trivial**, **noncyclic**, or **cyclic and solved**. An example is classified as **too many primes** only if neither program was able to enumerate the complete set of prime implicants, and an example is classified as **cyclic and unsolved** only if neither program was able to complete the covering program after having generated the set of all prime implicants.

6.2.2. Comparison of Exact Minimization Algorithms

I first report the results from the exact minimization algorithm of Espresso-MV, and the exact minimization algorithm McBoole. Note that both programs first generate the set of all prime implicants, and then attempt to find a minimum subset of the set of all prime implicants. Further, both programs attempt to solve only the simpler covering problem, namely, to return the cover with the fewest number of cubes without consideration for

[1] Tests show that the Apollo DN660 with Version 3.12 of the C Compiler executes Espresso-MV at the same speed as a DEC VAX 11/785 with the 4.3BSD portable C compiler. All results in this section were timed on an Apollo DN660 with 4 megabytes of memory.

the number of literals. (In fact, both programs use a "cleanup" step where the number of literals is reduced once the minimum number of rows has been achieved, but both programs solve this problem heuristically.) McBoole generates the prime implicants using the consensus algorithm described in [DAR86]. By maintaining the tree structure corresponding to where a cube was generated, McBoole is able to reduce the number of pairwise consensus operations that need to be performed. During the generation of prime implicants, McBoole creates a directed graph which is used to solve the selection of a subset of the set of all prime implicants.

Table 6.2.2 summarizes the comparison between Espresso-MV (exact mode), and McBoole for the 134 PLA's in the test set. *Number primes* is the number of examples for which each program was able to generate all of the primes for, *number solved* is the number of the examples for which each program was able to solve, and *time* gives the total time on an Apollo DN660 (in seconds) taken for those examples which could be solved within the 5 hour time limit. Thus, for example, Espresso-MV took more than 30,000 seconds longer than McBoole for the category cyclic and solved, but this involved solving 20 more problems than McBoole.

type	total	Espresso-MV (exact)			McBoole (exact)		
		number primes	number solved	time (sec)	number primes	number solved	time (sec)
trivial	9	9	9	120	9	9	271
noncyclic	56	55	54	26524	56	56	35956
cyclic and solved	42	42	41	41330	42	21	11241
cyclic and unsolved	10	7	0		10	0	
too many primes	17	0	0		0	0	
Totals	134	113	104	67974	117	86	47468

Table 6.2.2. Comparison of Espresso-MV (exact) and McBoole.

For examples with no cyclic constraints, both Espresso-MV and McBoole are usually able to find the minimum solution. Espresso-MV failed to generate the minimum solution for two examples (*al2* and *prom1*). For *prom1*, it was unable to enumerate all of the primes (which has 9,179 primes). For *al2*, it was able to generate all of the primes (there were 9,326 primes), but was unable to generate the prime implicant table.

However, when there are cyclic constraints, the covering algorithm of Espresso-MV is able to find the minimum solution for many more of the PLA's than McBoole. Only for example *intb* did Espresso-MV fail to solve an example with cyclic constraints that McBoole was able to solve. (Espresso-MV was unable to generate the prime implicant table for *intb* which has 6,522 prime implicants.) Sometimes the results are quite dramatic. The example *sqr6* was allowed to run for 58 hours with McBoole without terminating with the minimum solution; however, Espresso-MV is able to complete this same example in only 100 seconds. Also, Espresso-MV was able to determine the minimum cover for the example *mlp4* (a four bit multiplier) in about 1 hour. Results have been published for both of these examples without presenting the minimum solution [DAR86, Sas82]. As far as I know, no previous program has successfully minimized these two examples.

Comparing the efficiency of the prime generation algorithms, we find that in 113 cases both programs could generate all of the prime implicants, in 4 cases (*b4* with 6,455 primes, *bc0* with 6,596 primes, *prom1* with 9,326 primes, and *t1* with 15,135 primes) McBoole was able to generate all of the prime implicants when Espresso-MV could not, and in 17 cases neither program was able to generate all of the prime implicants. There were no cases where Espresso-MV was able to generate all of the primes, and McBoole was unable to.

Overall, there were 83 examples which both programs could minimize, 3 examples which McBoole could minimize which Espresso-MV could not, 21 examples which Espresso-MV could minimize which McBoole could not, and 27 examples for which neither program was able to complete the exact minimization (20 %). For the 83 examples which both programs could minimize, Espresso-MV used 38,198 seconds, and McBoole used 28,628 seconds. The Espresso-MV result had 51,821 literals, and McBoole had 53,686 literals indicating that MAKE_SPARSE was more efficient at reducing the number of literals (once the minimum number of terms was determined). Of course, for these 83 examples, both returned the same number of prime implicants, essential prime implicants, and solution cubes.

Including the time each program used on those examples for which a solution was not found, Espresso-MV used 6.1 days of computer time and McBoole used 10.3 days of computer time.

Detailed results for all 134 examples are given in Table 6.1. We summarize the lower bounds obtained from Espresso-MV and the best upper bound results for the 10 examples in the category **cyclic and unsolved** in Table 6.2.3.

Example	Primes	Essential Primes	Lower Bound	Upper Bound
9sym	1680	0	84	84
b4	6455	40		54
bc0	6596	37		177
ex5	2532	28	59	67
lin.rom	1087	8	125	129
max1024	1278	14	239	267
prom2	2635	9	274	287
spla	4972	33		251
t1	15135	7		102
tial	7145	220		575

Table 6.2.3 Upper and Lower Bounds for the **Cyclic and Unsolved** Problems.

6.2.3. Espresso-MV Results

I am thus in an excellent position to grade the quality of the results for the heuristic minimization algorithm Espresso-MV. I know the minimum solution for 107 of the 134 examples in the test set, and, as shown in 6.2.3 I have a lower bound for 5 of the remaining 27 examples.

Table 6.2.4 shows the totals for 133 examples, broken down by category, for Espresso-MV and Espresso-MV (strong mode). The examples were run on an Apollo DN660. It is evident that the SUPER_GASP option can be expensive; but, sometimes the extra reduction in the number of terms might be considered worthwhile. Curiously, SUPER_GASP produces more literals in all categories.

type	#	Espresso-MV solution cubes	lits	time (secs)	Espresso-MV (strong) solution cubes	lits	time (secs)
trivial	9	243	1683	23	243	1683	23
noncyclic	56	3909	45712	1674	3899	45956	2372
cyclic-s	42	4092	42030	3202	4056	42577	5403
cyclic-us	10	2023	25347	3444	2010	25438	4637
too-many-primes	16	2759	35718	6751	2755	35881	7924
Totals	133	13026	150490	15094	12963	151535	20359

Table 6.2.4. Espresso-MV Results.

Next I compare the results from Espresso-MV (again, with and without SUPER_GASP), but I only consider those examples for which Espresso-MV running as an exact minimizer was able to generate the minimum solution. This will allow me to compare the relative efficiency of Espresso-MV in its exact and heuristic modes. The results are shown in Table 6.2.4. It is evident that Espresso-MV provides a high quality result for all of the examples for which I can generate a minimum solution — the difference between Espresso-MV and Espresso-MV (exact) is about one percent. Also, Espresso-MV is more than fifteen times faster than the exact minimizer on problems that both algorithms can solve.

type	#	Espresso-MV solution cubes	lits	time (sec)	Espresso-MV (strong) solution cubes	lits	time (sec)	Espresso-MV (exact) solution cubes	lits	time (sec)
trivial	9	243	1683	23	243	1683	23	243	1683	120
noncyclic	54	3371	34060	1366	3361	34223	2030	3360	34204	26523
cyclic-s	41	3463	36163	2532	3427	36658	4279	3395	36564	41329
totals	104	7077	71906	3920	7031	72564	6332	6998	72451	67973

Table 6.2.5. Espresso-MV Exact Mode versus Heuristic Mode.

6.2.4. Comparison of Prestol-II and Espresso

Without access to the program Prestol-II, direct comparisons have been difficult to make. I compare here the results from Espresso-MV (in both normal and strong modes) and the results from Prestol-II reported in [BaM85]. (The raw data comes from the Ph.D. thesis of Marc Bartholomeus of Leuven University.) Table 6.2.6 presents results for 65 examples from the industrial and mathematical class. (Results for some random examples

will be reported in the next section.) The times for Prestol-II, which is a Pascal program, are from a VAX 11/780 running VMS. The times for Espresso-MV are from an Apollo DN660.

type	#	Espresso-MV solution cubes	Espresso-MV solution lits	Espresso-MV time (sec)	Espresso-MV (strong) solution cubes	Espresso-MV (strong) solution lits	Espresso-MV (strong) time (sec)	Prestol-II solution cubes	Prestol-II solution lits	Prestol-II time (sec)
trivial	1	112	736	13	112	736	13.3	112	736	84
noncyclic	29	2362	33217	1231	2354	33391	1340	2359	31858	1396
cyclic-s	22	2110	21751	1266	2098	21935	2542	2108	21944	1321
cyclic-us	6	1181	14392	1649	1178	14416	2457	1189	14329	1993
too many primes	7	928	7544	2758	924	7632	3283	928	7887	2186
total	65	6693	77640	6917	6666	78110	9635	6696	76754	6980

Table 6.2.6. Comparison Between Espresso-MV and Prestol-II.

We see that the results returned by Espresso-MV and Prestol-II are very close in quality of solution and in the execution time required. However, I have results from Prestol-II for only 13 of the 27 difficult problems.

6.2.5. Random Example Results

Included in the test set are 11 random examples provided by H. De Man of the University of Leuven. Results for some of these examples were first reported in [BaM84]. Each example is a truth table where the output value is randomly chosen from {0, 1, 2} with probabilities p_{off}, p_{on}, and p_{dc} respectively. Although I don't know the probabilities used to generate each example, I report below the observed percentages of minterms in the OFF-set, ON-set and DC-set for each example.

name	in	out	% OFF-set	% ON-set	% DC-set
bench	6	8	22.1	9.4	68.6
bench1	9	9	22.5	9.3	68.2
ex1010	10	10	15.3	14.4	70.3
exam	10	10	7.1	6.1	86.8
fout	6	10	28.8	29.8	41.4
p1	8	18	16.6	6.4	77.0
p3	8	14	14.0	6.3	79.6
test1	8	10	35.7	14.3	50.1
test2	11	35	19.4	9.9	70.7
test3	10	35	19.3	9.9	70.8
test4	8	30	8.8	19.7	71.5

Table 6.2.8. Distribution of Minterms for the Random Examples.

Note that the examples *test2*, and *test3*, and *test4* are large examples. Also, all of the examples have extremely large don't-care sets.

I next report the success for Espresso-MV (both normal and strong modes), McBoole, Espresso-MV (exact mode), Prestol-II and MINI for each of these examples. The results for MINI and Prestol-II are quoted from [BaM85]. Results for SPAM, Presto and Phipmin were also reported in [BaM84]; however, each of these three programs did significantly worse than either Espresso-MV or Prestol-II, and hence these results are not repeated here. The results from Espresso-MV in the exact mode also include, in some cases, a lower bound (returned from the minimum cover strategy outlined in Chapter 5), and an upper bound (if the minimum solution was not achieved).

name	MINI	Prestol-II	Espresso-MV	Espresso-MV strong	Espresso-MV exact	McBoole
bench	24	19	17	17	16	16
bench1	177	148	140	128	111-126	-
ex1010	389	246	302	264	-	-
exam	86	59	70	66	52-?	-
fout	48	42	42	42	40	-
p1	57	54	56	54	54	54
p3	41	39	40	39	39	39
test1	138	123	126	115	103-111	-
test2	-	-	1118	995	-	-
test3	922	552	558	491	-	-
test4	-	-	120	104	-	-

Table 6.2.9. Random Example Summary.

The results for Prestol-II have not been published for for examples *test2* and *test4*. There is a much greater variability among the different programs for these examples (especially for the largest random examples).

In particular, the results for *test3* were very surprising; Prestol-II and Espresso-MV were very close to each other, and it was assumed they were both close to the minimum. The addition of the SUPER_GASP strategy to Espresso-MV, however, produced a result with 61 fewer cubes then the best previously known result. Similar surprising results are seen in the data for *test2*, which, with the addition of the SUPER_GASP strategy produced a solution with 123 fewer cubes than without that strategy. However, running Espresso-MV in the strong mode greatly increased the execution time for this example. (Espresso-MV required 7 hours on an Apollo DN660).

These random problems are especially difficult minimization problems because of the large percentage of don't-care minterms, and the fact that the DC-set is scattered. As a result, all of these examples have a very large number of prime implicants, very few essential prime implicants, and most of them had cyclic constraints in the covering problem. Because these examples exhibit behavior much different from either the industrial examples or the mathematical functions, these results have been presented apart from the rest of the test set.

6.3. Multiple-Valued Minimization Results

6.3.1. Multiple-Valued Minimizer versus Binary-Valued with a DC-set

As mentioned in Chapter 1, it is possible to use a binary-valued minimizer to minimize a multiple-valued function. The problem is recast so that each value of a multiple-valued variable uses a single binary-valued variable, and a 1 in a cube for a multiple-valued variable is represented as a 1 in the binary-valued cube. A don't-care set is added which allows any number of 1's to appear simultaneously in the binary-variables which correspond to each of the multiple-valued variables. This technique is described in more detail in [BMH84, Chapter 5].

I present results for a small collection of multiple-valued minimization problems. Table 6.3.1 compares Espresso-MV running as a multiple-valued minimizer versus translating the problem into an equivalent binary-valued minimization problem, and using Espresso-MV as the binary-valued minimizer. The time reported for these examples was measured on an IBM 3081. The examples DK14, DK16, PCC, and BLUE represent problems that are being solved by the state-assignment program KISS [DBS85]. They have 7, 8, 12, and 93 states respectively.

Solving a multiple-valued minimization problem using a binary-valued minimization tool can be inefficient. In the two largest cases, the binary-valued minimizer was unable to complete the solution after 1 hour on an IBM 3081.

Example	States	Binary-Valued		Multiple-Valued	
		Terms	Time[2]	Terms	Time[2]
DK14	7	26	4.3	26	0.5
DK16	8	55	108.6	55	1.6
PCC	12	-	(3600)	48	4.4
BLUE	93	-	(3600)	775	1053.0

Table 6.3.1. Using a Binary-Valued Minimizer for Multiple-Valued Functions.

The computation did not terminate for either PCC or BLUE within the 1 hour time limit.

6.3.2. Multiple-Output Espresso-IIC versus Espresso-MV

Table 6.3.2 compares the performance of Espresso-MV against the binary-valued minimizer Espresso-IIC for the 56 examples published in [BMH84].

Program	Cubes	Literals	Time[2]
Espresso-MV	5993	60322	560
Espresso-IIC	6001	60578	992

Table 6.3.2. Espresso-MV versus Espresso-IIC.

Comparing Espresso-IIC and Espresso-MV, the quality of the results is almost identical, but the run-time has been reduced by almost fifty percent. This is a surprising result.

[2] Time in seconds measured on an IBM 3081 using the Waterloo C Compiler, Version 1.1 under the VM/CMS Operating System.

as one might expect the generalization of the algorithms to multiple-valued variables to penalize the performance for binary-valued minimization problems. However, the algorithms are improved by the more uniform treatment of the output-part during the multiple-valued minimization. For example, as described in Section 4.3, the OFF-set does not need to be represented with only a single-output active in each cube. This leads to a more compact representation of the OFF-set, and to a more efficient EXPAND procedure. Likewise, Espresso-IIC effectively would not split against the output part until reaching a leaf of one of the recursive procedures (e.g., TAUTOLOGY). By allowing the program to split against the output at any step of the procedure, the heuristics of choosing the splitting variable leads to a more efficient choice of splitting variables.

name	type	primes	essen	Espresso-MV cubes	Espresso-MV lits	Espresso-MV time	McBoole cubes	McBoole lits	McBoole time
5xp1	cyclic-s	390	8	63	360	55	* 64	0	1322
9sym	cyclic-us	1680	0	0	0	18000	0	0	18000
accpla	primes	0	0	0	0	18000	0	0	18000
add6	noncyclic	8568	153	355	2551	4546	355	2935	3235
addm4	cyclic-s	1122	24	189	1405	1526	* 191	1508	3477
adr4	noncyclic	397	35	75	415	34	75	467	12
al2	noncyclic	9179	16	0	0	18000	66	427	3017
alcom	noncyclic	4657	16	40	223	4596	40	224	1156
alu1	trivial	780	19	19	60	94	19	60	195
alu2	noncyclic	434	36	68	347	85	68	369	64
alu3	noncyclic	540	27	64	352	94	64	367	100
amd	cyclic-s	457	32	66	658	93	* 66	692	260
apla	noncyclic	201	0	25	232	52	25	228	11
b10	cyclic-s	938	51	100	1009	409	100	1081	55
b11	noncyclic	44	22	27	181	5	27	187	2
b12	cyclic-s	1490	2	41	233	715	0	0	18000
b2	noncyclic	928	54	104	1970	906	104	1977	41
b3	cyclic-s	3056	123	210	2506	6399	0	0	18000
b4	cyclic-us	6455	0	0	0	18000	0	0	18000
b7	noncyclic	44	22	27	181	5	27	187	2
b9	noncyclic	3002	48	119	873	687	119	938	558
bc0	cyclic-us	6596	37	0	0	18000	0	0	18000
bca	noncyclic	305	144	180	3281	1627	180	3454	8
bcb	noncyclic	255	137	155	2763	728	155	2799	7
bcc	cyclic-s	237	119	137	2530	892	137	2570	6
bcd	noncyclic	172	100	117	2026	444	117	2057	5
bcd.div3	trivial	13	9	9	38	1	9	38	1
bench	cyclic-s	391	0	16	102	43	16	125	13
bench1	cyclic-us	5972	0	0	0	18000	0	0	18000
br1	noncyclic	29	17	19	254	5	19	257	1
br2	noncyclic	27	9	13	172	4	13	174	1
chkn	cyclic-s	671	86	140	1742	629	140	1770	893
clpl	trivial	143	20	20	75	6	20	75	11
co14	trivial	14	14	14	210	2	14	210	1
cps	cyclic-s	2487	57	157	2849	2370	* 162	3154	10689
dc1	noncyclic	22	3	9	58	2	9	57	1
dc2	noncyclic	173	18	39	260	11	39	275	3
dekoder	cyclic-s	26	3	9	47	2	9	52	1
dist	cyclic-s	401	23	120	875	68	120	913	18
dk17	noncyclic	111	0	18	177	29	18	137	11
dk27	cyclic-s	82	0	10	61	28	10	46	12
dk48	cyclic-s	157	0	21	224	190	0	0	18000
ex1010	primes	0	0	0	0	18000	0	0	18000
ex4	primes	0	0	0	0	18000	0	0	18000
ex5	cyclic-us	2532	28	0	0	18000	0	0	18000

Table 6.1. Raw Data for Espresso-MV / McBoole Comparison.

name	type	primes	essen	Espresso-MV			McBoole		
				cubes	lits	time	cubes	lits	time
ex7	noncyclic	3002	48	119	873	709	119	938	540
exam	cyclic-us	4955	0	0	0	18000	0	0	18000
exep	noncyclic	558	82	108	1278	3318	108	1281	26
exp	noncyclic	238	30	56	559	75	56	662	5
exps	cyclic-s	852	56	132	1928	346	* 135	2099	4818
f51m	cyclic-s	561	13	76	401	64	76	450	25
fout	cyclic-s	436	2	40	306	399	* 41	392	2762
gary	cyclic-s	706	60	107	1118	180	107	1162	21
ibm	primes	0	0	0	0	18000	0	0	18000
in0	cyclic-s	706	60	107	1118	162	107	1149	17
in1	noncyclic	928	54	104	1970	892	104	1977	38
in2	cyclic-s	666	85	134	1430	189	134	1453	65
in3	noncyclic	1114	44	74	772	616	74	808	259
in4	cyclic-s	3076	118	211	2539	6945	211	2635	2331
in5	noncyclic	1067	53	62	741	258	62	746	36
in6	noncyclic	6174	40	54	547	3745	54	553	11819
in7	noncyclic	2112	31	54	427	950	54	434	1305
inc	cyclic-s	124	12	29	196	10	29	212	3
intb	cyclic-s	6522	186	0	0	18000	629	6342	7595
jbp	primes	0	0	0	0	18000	0	0	18000
l8err	cyclic-s	142	15	50	304	23	* 51	327	64
life	noncyclic	224	56	84	756	15	84	756	2
lin.rom	cyclic-us	1087	8	0	0	18000	0	0	18000
log8mod	cyclic-s	105	13	38	225	9	38	236	2
luc	noncyclic	190	14	26	388	37	26	416	5
m1	noncyclic	59	6	19	217	5	19	223	1
m181	cyclic-s	1636	2	41	233	866	0	0	18000
m2	cyclic-s	243	7	47	670	39	47	686	22
m3	cyclic-s	344	4	62	841	63	* 63	861	1438
m4	cyclic-s	670	11	101	1241	1049	* 103	1360	12919
mainpla	primes	0	0	0	0	18000	0	0	18000
mark1	cyclic-s	208	1	19	265	527	0	0	18000
max1024	cyclic-us	1278	14	0	0	18000	0	0	18000
max128	cyclic-s	469	6	78	1174	157	* 83	1105	13776
max46	trivial	49	46	46	441	7	46	441	1
max512	cyclic-s	535	20	133	1006	519	* 136	1069	9129
misg	primes	0	0	0	0	18000	0	0	18000
mish	primes	0	0	0	0	18000	0	0	18000
misj	primes	0	0	0	0	18000	0	0	18000
mlp4	cyclic-s	606	12	121	865	4722	* 123	955	3326
mp2d	cyclic-s	469	13	30	201	278	0	0	18000
newapla	noncyclic	113	9	17	102	7	17	106	4
newapla1	noncyclic	31	9	10	76	2	10	76	1
newapla2	trivial	7	7	7	49	1	7	49	1
newbyte	trivial	8	8	8	48	1	8	48	1

Table 6.1. Raw Data for Espresso-MV / McBoole Comparison (cont.).

name	type	primes	essen	Espresso-MV cubes	Espresso-MV lits	Espresso-MV time	McBoole cubes	McBoole lits	McBoole time
newcond	noncyclic	72	18	31	239	5	31	239	2
newcpla1	cyclic-s	170	22	38	263	24	38	303	16
newcpla2	noncyclic	38	14	19	129	4	19	129	1
newcwp	noncyclic	23	7	11	50	2	11	53	1
newill	cyclic-s	11	5	8	50	1	8	49	1
newtag	trivial	8	8	8	26	1	8	26	1
newtpla	noncyclic	40	16	23	199	3	23	201	1
newtpla1	noncyclic	6	3	4	37	1	4	37	1
newtpla2	noncyclic	23	4	9	69	2	9	69	1
newxcpla1	noncyclic	191	18	39	309	31	39	336	6
opa	cyclic-s	477	22	77	1121	234	* 78	1369	2951
p1	noncyclic	287	25	54	404	165	54	612	20
p3	noncyclic	185	22	39	280	79	39	324	10
p82	noncyclic	48	16	21	149	4	21	156	1
pdc	primes	0	0	0	0	18000	0	0	18000
pope.rom	cyclic-s	593	12	59	1472	347	* 61	1427	17167
prom1	noncyclic	9326	182	0	0	18000	472	11228	8228
prom2	cyclic-us	2635	9	0	0	18000	0	0	18000
radd	noncyclic	397	35	75	415	24	75	465	10
rckl	noncyclic	302	6	32	657	67	32	657	3043
rd53	noncyclic	51	21	31	173	2	31	175	1
rd73	noncyclic	211	106	127	903	18	127	904	3
risc	noncyclic	46	22	28	187	4	28	191	1
root	cyclic-s	152	9	57	381	26	57	401	5
ryy6	trivial	112	112	112	736	7	112	736	61
sex	noncyclic	99	13	21	105	6	21	105	2
shift	primes	0	0	0	0	18000	0	0	18000
signet	primes	0	0	0	0	18000	0	0	18000
soar.pla	primes	0	0	0	0	18000	0	0	18000
spla	cyclic-us	4972	33	0	0	18000	0	0	18000
sqn	noncyclic	75	23	38	226	6	38	233	1
sqr6	cyclic-s	205	3	47	274	114	* 49	299	1322
sym10	cyclic-s	3150	0	210	1470	9182	0	0	18000
t1	cyclic-us	15135	7	0	0	18000	0	0	18000
t2	noncyclic	233	25	52	363	39	52	386	8
t3	noncyclic	42	30	33	250	4	33	251	1
t4	noncyclic	174	0	16	91	68	16	97	14
test1	cyclic-us	2407	0	0	0	18000	* 116	1160	10727
test2	primes	0	0	0	0	18000	0	0	18000
test3	primes	0	0	0	0	18000	0	0	18000
test4	cyclic-us	6139	0	0	0	18000	0	0	18000
ti	primes	0	0	0	0	18000	0	0	18000
tial	cyclic-us	7145	220	0	0	18000	* 575	5355	11346
tms	cyclic-s	162	13	30	415	25	30	451	4
ts10	primes	0	0	0	0	18000	0	0	18000

Table 6.1. Raw Data for Espresso-MV / McBoole Comparison (cont.).

name	type	primes	essen	Espresso-MV cubes	Espresso-MV lits	Espresso-MV time	McBoole cubes	McBoole lits	McBoole time
vg2	noncyclic	1188	100	110	914	765	110	942	616
vtx1	noncyclic	1220	100	110	1074	259	110	1094	562
wim	cyclic-s	25	3	9	47	2	9	54	1
x1dn	noncyclic	1220	100	110	1074	257	110	1094	568
x2dn	primes	0	0	0	0	18000	0	0	18000
x6dn	cyclic-s	916	60	81	817	1848	81	820	150
x7dn	primes	0	0	0	0	18000	0	0	18000
x9dn	noncyclic	1272	110	120	1258	452	120	1298	611
xparc	primes	0	0	0	0	18000	0	0	18000
z4	noncyclic	167	35	59	311	11	59	333	3

Table 6.1. Raw Data for Espresso-MV / McBoole Comparison (cont.).

* indicates McBoole terminated branching after 10 levels; hence, the solution returned is not guaranteed optimal.

Times for both Espresso-MV and McBoole are for an Apollo DN660 with 4 megabytes of memory using Version 3.12 of the C Compiler.

McBoole detected that it had solved 5xp1 incorrectly; the problem was reported to the author, and the program was subsequently corrected.

McBoole and Espresso-MV disagree on the number of prime implicants for *l8err* (McBoole has 16, and Espresso-MV has 15). The problem is being investigated by the author of McBoole.

name	Espresso-MV			Espresso-MV (strong mode)			Prestol-II		
	cubes	lits	time	cubes	lits	time	cubes	lits	time
5xp1	63	358	27	64	415	30			
9sym	85	595	30	84	588	37	86	602	98
accpla	175	2750	924	175	2741	956			
add6	355	2551	144	355	2581	691			
addm4	200	1500	98	192	1441	219			
adr4	75	415	19	75	417	22	75	415	37
al2	66	427	21	66	427	22			
alcom	40	223	10	40	224	10			
alu1	19	60	1	19	60	2			
alu2	68	347	18	68	347	21			
alu3	66	347	14	64	360	26			
amd	66	660	50	66	658	53			
apla	25	221	9	25	238	10	25	223	18
b10	100	1000	35	100	1009	42	101	1004	42
b11	27	181	7	27	182	7			
b12	42	208	27	41	234	59	42	246	21
b2	106	1940	51	104	1972	49	104	1893	21
b3	211	2511	122	211	2512	149	211	2511	202
b4	54	546	35	54	546	37	54	546	22
b7	27	181	7	27	182	7	27	181	5
b9	119	873	22	119	873	33	119	873	53
bc0	178	2061	197	177	2088	260			
bca	180	3266	300	180	3285	307	181	2618	67
bcb	156	2778	159	155	2762	170	155	2191	595
bcc	137	2530	177	137	2533	179	138	2034	40
bcd	117	2026	96	117	2026	98			
bcd.div3	9	38	1	9	38	1			
bench	18	100	8	17	100	22	19	112	5
bench1	136	1187	161	128	1147	394	148	1245	705
br1	19	254	3	19	254	3	20	268	2
br2	13	172	3	13	172	3	14	188	2
chkn	140	1739	60	140	1764	70	140	1740	215
clpl	20	75	2	20	75	2			
co14	14	210	1	14	210	1			
cps	163	2824	344	159	2857	508			
dc1	9	54	2	9	58	2			
dc2	39	260	5	39	262	6	40	264	5
dekoder	9	47	2	9	48	3	9	53	1
dist	121	875	49	121	882	58	120	872	61
dk17	18	135	7	18	142	8			
dk27	10	46	5	10	61	7			
dk48	22	143	23	22	211	31			
ex1010	283	2743	1270	264	2623	2461	246	2667	2525
ex5	74	1900	115	72	1861	417	76	2014	69
ex7	119	873	22	119	873	33			

Table 6.2. Raw Data for Espresso-MV / Prestol-II Comparison.

name	Espresso-MV			Espresso-MV (strong mode)			Prestol-II		
	cubes	lits	time	cubes	lits	time	cubes	lits	time
exep	108	1274	61	108	1276	64	109	1221	55
exp	59	558	24	56	560	31	56	560	13
exps	134	1959	129	133	1946	145	135	2152	49
f51m	77	400	32	76	399	45	76	405	49
fout	44	315	31	42	318	36	42	335	22
gary	107	1116	30	107	1135	39	107	1119	25
ibm	173	1055	35	173	1055	36	173	2191	115
in0	107	1116	37	107	1133	45			
in1	106	1940	51	104	1972	49	104	1893	20
in2	136	1420	28	134	1437	58	137	1507	36
in3	74	773	30	74	775	32	74	754	23
in4	212	2543	105	212	2561	121			
in5	62	741	13	62	742	14	62	739	14
in6	54	547	11	54	547	12			
in7	54	427	12	54	429	13			
inc	30	198	7	29	195	9			
intb	629	5867	671	629	5919	1124			
jbp	122	1027	134	122	1030	151	123	1036	139
l8err	51	313	29	51	319	30			
life	84	756	18	84	756	20			
lin.rom	128	3202	269	128	3202	278			
log8mod	38	228	6	38	231	7			
luc	26	394	11	26	388	13	26	394	4
m1	19	217	4	19	217	6	19	217	2
m181	42	213	27	41	233	55	42	245	21
m2	47	648	29	47	640	34	47	672	6
m3	65	770	45	63	836	54	64	834	12
m4	107	1194	126	104	1172	153	105	1372	25
mainpla	172	8759	373	172	8761	650			
mark1	19	154	219	19	282	256			
max1024	274	2273	508	267	2266	678			
max128	82	1070	89	79	1108	111			
max46	46	441	2	46	441	2			
max512	143	1072	116	137	1058	141			
misg	69	247	15	69	279	23	69	247	59
mish	82	238	26	82	242	33			
misj	35	102	4	35	102	6			
mlp4	128	893	60	127	899	73	124	878	62
mp2d	31	198	22	31	201	34	34	215	44
newapla	17	102	3	17	102	3			
newapla1	10	76	1	10	79	1			
newapla2	7	49	1	7	49	1			
newbyte	8	48	1	8	48	1			
newcond	31	239	2	31	239	3			
newcpla1	38	263	7	38	264	9			

Table 6.2. Raw Data for Espresso-MV / Prestol-II Comparison (cont.).

name	Espresso-MV			Espresso-MV (strong mode)			Prestol-II		
	cubes	lits	time	cubes	lits	time	cubes	lits	time
newcpla2	19	129	2	19	130	2			
newcwp	11	50	1	11	52	1			
newill	8	50	1	8	52	1			
newtag	8	26	1	8	26	1			
newtpla	23	199	2	23	200	2			
newtpla1	4	37	1	4	37	1			
newtpla2	9	69	1	9	69	1			
newxcpla1	39	282	11	39	283	15			
opa	79	1097	111	79	1095	136	79	1144	50
p82	21	149	3	21	149	3			
pdc	123	1126	2133	119	1172	2581	122	1097	1611
pope.rom	62	1345	90	59	1418	140			
prom1	472	11225	288	472	11306	320	472	11237	200
prom2	287	5610	635	287	5610	662	288	5353	50
radd	75	415	9	75	417	15	75	415	24
rckl	32	657	50	32	657	49	32	657	11
rd53	31	175	2	31	173	3	31	173	2
rd73	127	903	14	127	903	15			
risc	28	187	5	28	187	5			
root	57	383	21	57	387	23	57	384	14
ryy6	112	736	13	112	736	13	112	736	84
sex	21	105	2	21	109	2			
shift	100	493	6	100	493	6	100	493	14
signet	119	636	356	119	638	360			
soar.pla	352	3049	1053	352	3094	1197			
spla	262	3419	821	260	3466	964			
sqn	38	230	5	38	228	6	38	228	8
sqr6	49	266	13	49	280	18	49	268	17
sym10	210	1470	98	210	1470	1093	210	1470	282
t1	102	612	84	102	628	120	102	650	95
t2	53	362	17	53	361	19	52	359	21
t3	33	250	4	33	251	4	33	251	4
t4	16	89	27	16	94	45	17	89	15
ti	213	2572	425	213	2579	478	213	1799	230
tial	579	5129	751	579	5183	1185	583	5164	1659
tms	30	486	9	30	416	18			
ts10	128	1024	8	128	1024	8	128	1024	18
vg2	110	914	17	110	914	19	110	914	60
vtx1	110	1074	14	110	1074	16			
wim	9	43	2	9	43	3			
x1dn	110	1074	14	110	1074	16	110	1074	47
x2dn	104	564	53	104	565	61			
x6dn	81	814	22	81	823	24	81	819	47
x7dn	538	4600	524	538	4603	651			

Table 6.2. Raw Data for Espresso-MV / Prestol-II Comparison (cont.).

name	Espresso-MV			Espresso-MV (strong mode)			Prestol-II		
	cubes	lits	time	cubes	lits	time	cubes	lits	time
x9dn	120	1258	17	120	1258	18	120	1258	52
xparc	254	7476	680	254	7503	728			
z4	59	311	8	59	311	9	59	311	17

Table 6.2. Raw Data for Espresso-MV / Prestol-II Comparison (cont.).

Data for Prestol-II comes from the PhD thesis of Marc Bartholomeus, Leuven University.

Time for Prestol-II is on a VAX 11/780 under VMS in seconds; Time for Espresso-MV is on an Apollo DN660 in seconds.

There is the possibility that the total number of literals for *ibm* is in error for Prestol-II. The number of literals is very large, and happens to equal the number of literals on the line immediately above in Table 4.4 of Bartholomeus' thesis.

In [BaM84], a result was reported for *m2* which was later proven incorrect by Espresso-MV running in the exact mode. The error was subsequently acknowledged and corrected by the author of Prestol-II.

APPENDIX A

Espresso-MV Program Documentation

NAME

espresso — Boolean Minimization

SYNOPSIS

espresso [*type*] [*file*] [*options*]

DESCRIPTION

Espresso takes as input a two-level representation of a two-valued (or a multiple-valued) Boolean function, and produces a minimal equivalent representation. The algorithms used are new and represent an advance in both speed and optimality of solution in heuristic Boolean minimization.

Espresso reads the *file* provided (or standard input if no files are specified), performs the minimization, and writes the minimized result to standard output. *Espresso* automatically verifies that the minimized function is equivalent to the original function. Options allow for using an exact minimization algorithm, for choosing an optimal phase assignment for the output functions, and for choosing an optimal assignment of the inputs to input decoders.

The default input and output file formats are compatible with the Berkeley standard format for the physical description of a PLA. The input format is described in detail in espresso(5). Note that the input file is a *logical* representation of a set of Boolean equations, and hence the input format differs slightly from that described in pla(5) (which provides for the *physical* representation of a PLA). The input and output formats have been expanded to allow for multiple-valued logic functions, and to allow for the specification of the don't care set which will be used in the minimization.

Type specifies the logical format for the function. The allowed types are -f, -r, -fr, -fd, -dr, and -fdr which have the same meanings assigned in espresso(5).

The command line options described below can be specified anywhere on the command line and must be separated by spaces. A complete list of the command line options is given below. Be warned that many of the command line options are for internal use and debugging only.

-d Verbose detail describing the progress of the minimization is written to standard output. Useful only for those familiar with the algorithms used.

-do [s] This option executes subprogram [s]. Some of the more useful ones are listed separately below. The remaining subprograms (contain, d1merge_in, d1merge_out, disjoint, dsharp, intersect, minterms, primes, sharp, union, unravel; essen, expand, irred, make_sparse, mincov, reduce, taut, super_gasp) are intended for those heavily into manipulating Boolean functions.

-do check

Checks that the function is a partition of the entire space (i.e., that the ON-set, OFF-set and DC-set are pairwise disjoint, and that their union is the Universe)

-do d1merge

Performs a quick distance-1 merge on the input file to reduce the number of terms. Useful when the input file is very large (e.g., a truth table with more than 1000 terms) because distance-1 merge is O(n log n) rather than Espresso which is O(n * n). It is expected that the output would then be run through espresso to complete the minimization.

-do echo

Implies "-out fdr" and echoes the function to standard output. This can be used

to compute the complement of a function.

-do exact
Exact minimization algorithm (guarantees minimum number of product terms, and heuristically minimizes number of literals). Potentially expensive.

-do map
Draw the Karnaugh maps for a function.

-do opo Perform output phase optimization (i.e., determine which functions to complement to reduce the number of terms needed to implement the function). After choosing an assignment of phases for the outputs, the function is minimized.

-do opoall
Minimize the function with all possible phase assignments. The option can be followed by three integers which specify the first and last outputs to be used (counting from 0), and the third integer is 0 to use the heuristic minimizer in espresso or 1 to use the exact minimizer in espresso. Be warned that opoall requires an exponential number of minimizations !

-do pair
Choose an assignment of the inputs to two-bit decoders, and minimize the function.

-do pairall
Minimize the function with all possible assignments of inputs to two-bit decoders. The option can be followed by an integer which is 2 to use the heuristic minimizer of espresso, 3 to use the exact minimizer of espresso, and 4 to perform output phase assignment (as in the -do opo option) for each assignment. Be warned that pairall requires an exponential number of minimizations !

-do single_output
Minimize each function one at a time as a single-output function. Terms will not be shared among the functions.

-do single_output_best
Minimize each function one at a time as a single-output function, but choose the function or its complement based on which has fewer terms.

-do stats
Provide simple statistics on the size of the function.

-do verify
Reads two file names from the command line and verifies that the two functions are Boolean equivalent.

-do PLAverify
Reads two filenames from the command line, assumes that each specifies names for the inputs and outputs, permutes columns so that the two PLA's have the same order for the inputs and outputs, and then checks Boolean equivalence between the two functions.

-eat Normally comments are echoed from the input file to the output file. This options discards any comments in the input file.

-fast Stop after the first EXPAND and IRREDUNDANT operations (i.e., do not iterate over the solution).

-kiss Sets up a *kiss*-style minimization problem.

-ness Essential primes will not be detected and removed from the minimization.

-nirr The result will not necessarily be made irredundant in the final step which removes redundant literals.

-nunwrap
The ON-set will not be unwrapped before beginning the minimization.

-help Provides a quick summary of the available command line options.

-onset Recompute the ON-set before the minimization. Useful when the PLA has a large number of product terms (e.g., an exhaustive list of minterms).

-out [s] Selects the output format. By default, only the ON-set (i.e., type f) is output after the minimization. [s] can be one of **f**, **d**, **r**, **fd**, **dr**, **fr**, or **fdr** to select any combination of the ON-set (f), the OFF-set (r) or the DC-set (d). [s] may also be **eqntott** to output algebraic equations acceptable to eqntott(1), or **pleasure** to output an unmerged PLA (with the *.label* and *.group* keywords) acceptable to pleasure(1).

-pos Swaps the ON-set and OFF-set of the function after reading the function. This can be used to minimize the OFF-set of a function. *.phase* in the input file can also specify an arbitrary choice of output phases.

-s Will provide a short summary of the execution of the program including the initial cost of the function, the final cost, and the computer resources used.

-strong Uses an alternate strategy for the LAST_GASP step which is more expensive, but occasionally provides better results.

-t Will produce a trace showing the execution of the program. After each main step of the algorithm, a single line is printed which reports the processor time used, and the current cost of the function.

-x Suppress printing of the solution.

DIAGNOSTICS

espresso will issue a warning message if a product term spans more than one line. Usually this is an indication that the number of inputs or outputs of the function is specified incorrectly.

SEE ALSO

kiss(1), pleasure(1), pla(5), espresso(5)

R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

R. Rudell, A. Sangiovanni-Vincentelli, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization," *Proc. Cust. Int. Circ. Conf.*, May 1985.

R. Rudell, "Multiple-Valued Minimization for PLA Synthesis," Master's Report, University of California, Berkeley, June 1986.

AUTHOR

Richard Rudell

BUGS

Always passes unrecognized options straight from the input file to standard output (sometimes this isn't what you want).

There are a lot of options, but typical use doesn't need them.

NAME

espresso -- input file format for espresso(1)

DESCRIPTION

Espresso accepts as input a two-level description of a Boolean switching function. This is described as a character matrix with keywords imbedded in the input to specify the size of the matrix and the logical format of the input function. Comments are allowed within the input by placing a pound sign (#) as the first character on a line. Comments and unrecognized keywords are passed directly from the input file to standard output. Any white-space (blanks, tabs, etc.), except when used as a delimiter in an imbedded command, is ignored. It is generally assumed that the PLA is specified such that each row of the PLA fits on a single line in the input file.

KEYWORDS

The following keywords are recognized by *espresso*. The list shows the probable order of the keywords in a PLA description. [d] denotes a decimal number and [s] denotes a text string.

.i [d] Specifies the number of input variables.

.o [d] Specifies the number of output functions.

.type [s] Sets the logical interpretation of the character matrix as described below under "Logical Description of a PLA". This keyword must come before any product terms. [s] is one of f, r, fd, fr, dr, or fdr.

.phase [s] [s] is a string of as many 0's or 1's as there are output functions. It specifies which polarity of each output function should be used for the minimization (a 1 specifies that the ON-set of the corresponding output function should be used, and a 0 specifies that the OFF-set of the corresponding output function should be minimized).

.pair [d] Specifies the number of pairs of variables which will be paired together using two-bit decoders. The rest of the line contains pairs of numbers which specify the binary variables of the PLA which will be paired together. The binary variables are numbered starting with 1. The PLA will be reshaped so that any unpaired binary variables occupy the leftmost part of the array, then the paired multiple-valued columns, and finally any multiple-valued variables.

.kiss Sets up for a *kiss*-style minimization.

.p [d] Specifies the number of product terms. The product terms (one per line) follow immediately after this keyword. Actually, this line is ignored, and the ".e", ".end", or the end of the file indicate the end of the input description.

.e (.end) Marks the end of the PLA description.

LOGICAL DESCRIPTION OF A PLA

When we speak of the ON-set of a Boolean function, we mean those minterms which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

The purpose of the *espresso* minimization program is to find a logically equivalent set of product-terms to represent the ON-set and optionally minterms which lie in the DC-set, without containing any minterms of the OFF-set.

A Boolean function can be described in one of the following ways:

1) By providing the ON-set. In this case, *espresso* computes the OFF-set as the complement of the ON-set and the DC-set is empty. This is indicated with the keyword **.type f** in the input file, or **-f** on the command line.

2) By providing the ON-set and DC-set. In this case, *espresso* computes the OFF-set as the complement of the union of the ON-set and the DC-set. If any minterm belongs to both the ON-set and DC-set, then it is considered a don't care and may be removed from the ON-set during the minimization process. This is indicated with the keyword **.type fd** in the input file, or **-fd** on the command line.

3) By providing the ON-set and OFF-set. In this case, *espresso* computes the DC-set as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the subprogram "-do check" which will check the consistency of a function. This is indicated with the keyword **on the command line.**

4) By providing the ON-set, OFF-set and DC-set. This is indicated with the keyword **.type fdr** in the input file, or **-fdr** on the command line.

If at all possible, *espresso* should be given the DC-set (either implicitly or explicitly) in order to improve the results of the minimization.

A term is represented by a "cube" which can be considered either a compact representation of an algebraic product term which implies the function value is a 1, or as a representation of a row in a PLA which implements the term. A cube has an input part which corresponds to the input plane of a PLA, and an output part which corresponds to the output plane of a PLA (for the multiple-valued case, see below).

SYMBOLS IN THE PLA MATRIX AND THEIR INTERPRETATION

Each position in the input plane corresponds to an input variable where a 0 implies the corresponding input literal appears complemented in the product term, a 1 implies the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term.

With logical type *f*, for each output, a **1** means this product term belongs to the ON-set, and a **0** or - means this product term has no meaning for the value of this function. This logical type corresponds to an actual PLA where only the ON-set is actually implemented.

With logical type *fd* (the default), for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term has no meaning for the value of this function, and a - implies this product term belongs to the DC-set.

With logical type *fr*, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, and a - means this product term has no meaning for the value of this function.

With logical type *fdr*, for each output, a **1** means this product term belongs to the ON-set, a **0** means this product term belongs to the OFF-set, a - means this product term belongs to the DC-set, and a ~ implies this product term has no meaning for the value of this function.

Note that regardless of the logical type of PLA, a ~ implies the product term has no meaning for the value of this function. **2** is allowed as a synonym for **-**, **4** is allowed for **1**, and **3** is allowed for ~ . Also, the logical PLA type can also be specified on the command line.

MULTIPLE-VALUED FUNCTIONS

Espresso will also minimize multiple-valued Boolean functions. There can be an arbitrary number of multiple-valued variables, and each can be of a different size. If there are also binary-valued variables, they should be given as the first variables on the line (for ease of description). Of course, it is always possible to place them anywhere on the line as a two-valued multiple-valued variable. The function size is described by the imbedded option

.mv [num_var] [num_binary_var] [s1] . . . [sn]

Specifies the number of variables (num_var), the number of binary variables (num_binary_var), and the size of each of the multiple-valued variables (s1 through sn).

A multiple-output binary function with *ni* inputs and *no* outputs would be specified as ".mv *ni+1 ni no*." ".mv" cannot be used with either ".i" or ".o" — use one or the other to specify the function size.

The binary variables are given as described above. Each of the multiple-valued variables are given as a bit-vector of **0** and **1** which have their usual meaning for multiple-valued functions. The last multiple-valued variable (also called the output) is interpreted as described above for the output (to split the function into an ON-set, OFF-set and DC-set). A vertical bar | may be used to separate the multiple-valued fields in the input file.

If the size of the multiple-valued field is less than zero, than a symbolic field is interpreted from the input file. The absolute value of the size specifies the maximum number of unique symbolic labels which are expected in this column. The symbolic labels are white-space delimited strings of characters.

To perform a *kiss*-style encoding problem, either the keyword **.kiss** must be in the file, or the **-kiss** option must be used on the command line. Further, the third to last variable on the input file must be the symbolic "present state", and the second to last variable must be the "next state". As always, the last variable is the output. The symbolic "next state" will be hacked to be actually part of the output.

EXAMPLE #1.

A two-bit adder which takes in two 2-bit operands and produces a 3-bit result can be described completely in minterms as:

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
.type fr
.pair 2 (1 3) (2 4)
.phase 011
00 00     000
00 01     001
00 10     010
00 11     011
01 00     001
01 01     010
01 10     011
01 11     100
10 00     010
10 01     011
10 10     100
10 11     101
11 00     011
11 01     100
11 10     101
11 11     110
.end
```

The logical format for this input file (i.e., type fr) is given to indicate that the file contains both the ON-set and the OFF-set. Note that in this case, the zeros in the output plane are really specifying "value must be zero" rather than "no information".

The imbedded option *.pair* indicates that the first binary-valued variable should be paired with the third binary-valued variable, and that the second variable should be paired with the fourth variable. The function will then be mapped into an equivalent multiple-valued minimization problem.

The imbedded option *.phase* indicates that the positive-phase should be used for the second and third outputs, and that the negative phase should be used for the first output.

EXAMPLE #2

This example shows a description of a multiple-valued function with 5 binary variables and 3 multiple-valued variables (8 variables total) where the multiple-valued variables have sizes of 4 27 and 10 (note that the last multiple-valued variable is the "output" and also encodes the ON-set, DC-set and OFF-set information).

```
.mv  8  5  4  27  10
0-010|1000|100000000000000000000000000|0010000000
10-10|1000|010000000000000000000000000|1000000000
0-111|1000|001000000000000000000000000|0001000000
0-10-|1000|000100000000000000000000000|0001000000
00000|1000|000010000000000000000000000|1000000000
00010|1000|000001000000000000000000000|0010000000
01001|1000|000000100000000000000000000|0000000010
0101-|1000|000000010000000000000000000|0000000000
0-0-0|1000|000000001000000000000000000|1000000000
10000|1000|000000000100000000000000000|0000000000
11100|1000|000000000010000000000000000|0010000000
10-10|1000|000000000001000000000000000|0000000000
11111|1000|000000000000100000000000000|0010000000
                      .
                      .
                      .
11111|0001|000000000000000000000000001|0000000000
```

EXAMPLE #3

This example shows a description of a multiple-valued function setup for *kiss*-style minimization. There are 5 binary variables, 2 symbolic variables (the present-state and the next-state of the FSM) and the output (8 variables total).

```
.mv 8 5 -10 -10 6
.type fr
.kiss
# This is a translation of IOFSM from OPUS
# inputs are      IO1 IO0 INIT SWR MACK
# outputs are    WAIT MINIT MRD SACK MWR DLI
# reset logic
--1--      -          init0      110000
# wait for INIT to go away
--1--      init0      init0      110000
--0--      init0      init1      110000
# wait for SWR
--00-      init1      init1      110000
--01-      init1      init2      110001
# Latch address
--0--      init2      init4      110100
# wait for SWR to go away
--01-      init4      init4      110100
--00-      init4      iowait     000000
# wait for command from MFSM
0000-      iowait     iowait     000000
1000-      iowait     init1      110000
01000      iowait     read0      101000
11000      iowait     write0     100010
01001      iowait     rmack      100000
11001      iowait     wmack      100000
--01-      iowait     init2      110001
# wait for MACK to fall (read operation)
--0-0      rmack      rmack      100000
--0-1      rmack      read0      101000
# wait for MACK to fall (write operation)
--0-0      wmack      wmack      100000
--0-1      wmack      write0     100010
# perform read operation
--0--      read0      read1      101001
--0--      read1      iowait     000000
# perform write operation
--0--      write0     iowait     000000
.end
```

APPENDIX B

Summary of Optimal Results for the PLA Test Set

This table presents the results of Boolean minimization for the 145 PLA's in the Berkeley PLA test suite. The cost function is assumed to be minimum number of terms with only a secondary concern given to the number of literals. Each example is classified as one of 3 types:

type	description
indust	example donated from actual chip designs
math	mathematical function
random	randomly generated example

Each example also belongs to one of 5 categories, which measures the relative difficulty of the problem:

class	description
trivial	minimum solution consists of essential prime implicants
noncyclic	the covering problem contains no cyclic constraints
cyclic-s	the covering problem contains cyclic constraints, and the covering problem has been solved
cyclic-us	the covering problem contains cyclic constraints, and the covering problem as not been solved
primes	unable to enumerate all prime implicants

These classifications were determined by using the exact minimization algorithms of Espresso-MV as well as the exact minimization algorithm of McBoole. The classifications of *cyclic-us* and *primes* are dependent on the exact minimization algorithms which were used. For example, although we know the minimum solution for *Z9sym* and *ibm* (by methods not involving the use of an exact minimization algorithm) these examples are still classified as *cyclic-us* and *primes* respectively because the exact minimization algorithm was unable to determine the minimum solution.

For each example, we first give the number of inputs, the number of outputs, and the number of terms in the initial representation of the function. If the number of terms is

marked by *, then there is a don't-care set specified for the function (which is not counted in the initial number of terms).

We then present the number of prime implicants (when known), the number of essential primes, and the minimum solution (when known). When the minimum solution is not known for the class *cyclic-us* a lower bound (as determined by the covering algorithm of Espresso-MV) and an upper bound (the best solution we've seen) are given. For the class *primes*, the lower bound is merely the number of essential prime implicants, and the upper bound is the best solution we've seen. For the examples *ex1010* and *exam* the best results have been reported by the authors Prestol-II, and we have not seen or verified the results.

This table also gives the results for Espresso-MV in both its normal mode (Esp.) and its *strong* mode (Esp. (s)).

name	in/out	terms	type	class	# primes	# essen.	minimum solution	Esp.	Esp. (s)
alu1	12/8	19	indust	trivial	780	19	19	19	19
bcd.div3	4/4	* 9	math	trivial	13	9	9	9	9
clpl	11/5	20	indust	trivial	143	20	20	20	20
co14	14/1	14	math	trivial	14	14	14	14	14
max46	9/1	46	indust	trivial	49	46	46	46	46
newapla2	6/7	7	indust	trivial	7	7	7	7	7
newbyte	5/8	8	indust	trivial	8	8	8	8	8
newtag	8/1	8	indust	trivial	8	8	8	8	8
ryy6	16/1	112	indust	trivial	112	112	112	112	112
add6	12/7	1092	math	noncyclic	8568	153	355	355	355
adr4	8/5	255	math	noncyclic	397	35	75	75	75
al2	16/47	103	indust	noncyclic	9179	16	66	66	66
alcom	15/38	47	indust	noncyclic	4657	16	40	40	40
alu2	10/8	* 87	indust	noncyclic	434	36	68	68	68
alu3	10/8	* 68	indust	noncyclic	540	27	64	66	64
apla	10/12	* 112	indust	noncyclic	201	0	25	25	25
b11	8/31	* 74	indust	noncyclic	44	22	27	27	27
b2	16/17	110	indust	noncyclic	928	54	104	106	104
b7	8/31	* 74	indust	noncyclic	44	22	27	27	27
b9	16/5	123	indust	noncyclic	3002	48	119	119	119
bca	26/46	* 301	indust	noncyclic	305	144	180	180	180
bcb	26/39	* 299	indust	noncyclic	255	137	155	156	155
bcd	26/38	* 243	indust	noncyclic	172	100	117	117	117
br1	12/8	34	indust	noncyclic	29	17	19	19	19
br2	12/8	35	indust	noncyclic	27	9	13	13	13
dc1	4/7	15	indust	noncyclic	22	3	9	9	9
dc2	8/7	58	indust	noncyclic	173	18	39	39	39
dk17	10/11	* 57	indust	noncyclic	111	0	18	18	18
ex7	16/5	123	indust	noncyclic	3002	48	119	119	119
exep	30/63	* 149	indust	noncyclic	558	82	108	108	108
exp	8/18	* 89	indust	noncyclic	238	30	56	59	56
in1	16/17	110	indust	noncyclic	928	54	104	106	104
in3	35/29	75	indust	noncyclic	1114	44	74	74	74
in5	24/14	62	indust	noncyclic	1067	53	62	62	62
in6	33/23	54	indust	noncyclic	6174	40	54	54	54
in7	26/10	84	indust	noncyclic	2112	31	54	54	54
life	9/1	140	math	noncyclic	224	56	84	84	84
luc	8/27	27	indust	noncyclic	190	14	26	26	26
m1	6/12	32	indust	noncyclic	59	6	19	19	19
newapla	12/10	17	indust	noncyclic	113	9	17	17	17
newapla1	12/7	10	indust	noncyclic	31	9	10	10	10
newcond	11/2	31	indust	noncyclic	72	18	31	31	31
newcpla2	7/10	19	indust	noncyclic	38	14	19	19	19
newcwp	4/5	11	indust	noncyclic	23	7	11	11	11
newtpla	15/5	23	indust	noncyclic	40	16	23	23	23

Table B.1. Optimum Results for the Berkeley PLA Test Set.

name	in/out	terms	type	class	# primes	# essen.	minimum solution	Esp.	Esp. (s)
newtpla1	10/2	4	indust	noncyclic	6	3	4	4	4
newtpla2	10/4	9	indust	noncyclic	23	4	9	9	9
newxcpla1	9/23	40	indust	noncyclic	191	18	39	39	39
p1	8/18	* 89	random	noncyclic	287	25	54	55	54
p3	8/14	* 66	random	noncyclic	185	22	39	39	39
p82	5/14	24	indust	noncyclic	48	16	21	21	21
prom1	9/40	502	indust	noncyclic	9326	182	472	472	472
radd	8/5	120	math	noncyclic	397	35	75	75	75
rckl	32/7	.96	math	noncyclic	302	6	32	32	32
rd53	5/3	31	math	noncyclic	51	21	31	31	31
rd73	7/3	147	math	noncyclic	211	106	127	127	127
risc	8/31	74	indust	noncyclic	46	22	28	28	28
sex	9/14	23	indust	noncyclic	99	13	21	21	21
sqn	7/3	84	indust	noncyclic	75	23	38	38	38
t2	17/16	* 128	indust	noncyclic	233	25	52	53	53
t3	12/8	148	indust	noncyclic	42	30	33	33	33
t4	12/8	* 38	indust	noncyclic	174	0	16	16	16
vg2	25/8	110	indust	noncyclic	1188	100	110	110	110
vtx1	27/6	110	indust	noncyclic	1220	100	110	110	110
x1dn	27/6	112	indust	noncyclic	1220	100	110	110	110
x9dn	27/7	120	indust	noncyclic	1272	110	120	120	120
z4	7/4	127	math	noncyclic	167	35	59	59	59
Z5xp1	7/10	128	math	cyclic-s	390	8	63	63	64
addm4	9/8	480	math	cyclic-s	1122	24	189	200	192
amd	14/24	171	indust	cyclic-s	457	32	66	66	66
b10	15/11	* 135	indust	cyclic-s	938	51	100	100	100
b12	15/9	431	indust	cyclic-s	1490	2	41	42	41
b3	32/20	* 234	indust	cyclic-s	3056	123	210	211	211
bcc	26/45	* 245	indust	cyclic-s	237	119	137	137	137
bench	6/8	* 31	random	cyclic-s	391	0	16	18	17
chkn	29/7	153	indust	cyclic-s	671	86	140	140	140
cps	24/109	654	indust	cyclic-s	2487	57	157	163	159
dekoder	4/7	* 10	indust	cyclic-s	26	3	9	9	9
dist	8/5	255	math	cyclic-s	401	23	120	121	121
dk27	9/9	* 20	indust	cyclic-s	82	0	10	10	10
dk48	15/17	* 42	indust	cyclic-s	157	0	21	22	22
exps	8/38	* 196	indust	cyclic-s	852	56	132	134	133
f51m	8/8	255	math	cyclic-s	561	13	76	77	76
fout	6/10	* 61	random	cyclic-s	436	2	40	44	42
gary	15/11	214	indust	cyclic-s	706	60	107	107	107
in0	15/11	135	indust	cyclic-s	706	60	107	107	107
in2	19/10	137	indust	cyclic-s	666	85	134	136	134
in4	32/20	234	indust	cyclic-s	3076	118	211	212	212
inc	7/9	* 34	indust	cyclic-s	124	12	29	30	29

Table B.1. Optimum Results for the Berkeley PLA Test Set.

name	in/out	terms	type	class	# primes	# essen.	minimum solution	Esp.	Esp. (s)
intb	15/7	664	indust	cyclic-s	6522	186	629	629	629
l8err	8/8	* 253	math	cyclic-s	142	15	50	51	51
log8mod	8/5	46	math	cyclic-s	105	13	38	38	38
m181	15/9	430	math	cyclic-s	1636	2	41	42	41
m2	8/16	96	indust	cyclic-s	243	7	47	47	47
m3	8/16	128	indust	cyclic-s	344	4	62	65	63
m4	8/16	256	indust	cyclic-s	670	11	101	107	104
mark1	20/31	* 23	indust	cyclic-s	208	1	19	19	19
max128	7/24	128	indust	cyclic-s	469	6	78	82	79
max512	9/6	512	indust	cyclic-s	535	20	133	142	137
mlp4	8/8	225	math	cyclic-s	606	12	121	128	127
mp2d	14/14	123	indust	cyclic-s	469	13	30	31	31
newcpla1	9/16	38	indust	cyclic-s	170	22	38	38	38
newill	8/1	8	indust	cyclic-s	11	5	8	8	8
opa	17/69	342	indust	cyclic-s	477	22	77	79	79
pope.rom	6/48	64	indust	cyclic-s	593	12	59	62	59
root	8/5	255	math	cyclic-s	152	9	57	57	57
sqr6	6/12	63	math	cyclic-s	205	3	47	49	49
sym10	10/1	837	math	cyclic-s	3150	0	210	210	210
tms	8/16	30	indust	cyclic-s	162	13	30	30	30
wim	4/7	* 10	indust	cyclic-s	25	3	9	9	9
x6dn	39/5	121	indust	cyclic-s	916	60	81	81	81
Z9sym	9/1	420	math	cyclic-us	1680	0	84/84	85	84
b4	33/23	* 54	indust	cyclic-us	6455	40	40/54	54	54
bc0	26/11	419	indust	cyclic-us	6596	37	37/177	178	177
bench1	9/9	* 285	random	cyclic-us	5972	0	111/126	136	128
ex5	8/63	256	indust	cyclic-us	2532	28	59/67	74	72
exam	10/10	* 410	random	cyclic-us	4955	0	52/59	67	66
lin.rom	7/36	128	indust	cyclic-us	1087	8	125/128	128	128
max1024	10/6	1024	indust	cyclic-us	1278	14	239/267	274	267
prom2	9/21	287	indust	cyclic-us	2635	9	274/287	287	287
spla	16/46	* 2296	indust	cyclic-us	4972	33	33/251	262	260
t1	21/23	796	indust	cyclic-us	15135	7	7/102	102	102
test1	8/10	* 209	random	cyclic-us	2407	0	103/111	123	115
test4	8/30	* 256	random	cyclic-us	6139	0	0/104	122	104
tial	14/8	640	math	cyclic-us	7145	220	220/575	579	579
accpla	50/69	183	indust	primes	?	97	97/175	175	175
ex1010	10/10	* 810	random	primes	?	0	0/246	283	264
ex4	128/28	620	indust	primes	?	138	138/279	279	279
ibm	48/17	173	indust	primes	?	172	173/173	173	173
jbp	36/57	166	indust	primes	?	0	0/122	122	122
mainpla	27/54	181	indust	primes	?	29	29/172	172	172
misg	56/23	75	indust	primes	?	3	3/69	69	69

Table B.1. Optimum Results for the Berkeley PLA Test Set.

name	in/out	terms	type	class	# primes	# essen.	minimum solution	Esp.	Esp. (s)
mish	94/43	91	indust	primes	?	3	3/82	82	82
misj	35/14	48	indust	primes	?	13	13/35	35	35
pdc	16/40	* 2406	indust	primes	?	2	2/100	125	121
shift	19/16	100	indust	primes	?	100	100/100	100	100
signet	39/8	124	indust	primes	?	104	104/119	119	119
soar.pla	83/94	529	indust	primes	?	2	2/352	352	352
test2	11/35	* 1999	random	primes	?	0	0/995	1105	
test3	10/35	* 1003	random	primes	?	0	0/491	543	491
ti	47/72	241	indust	primes	?	46	46/213	213	213
ts10	22/16	128	indust	primes	?	128	128/128	128	128
x2dn	82/56	112	indust	primes	?	2	2/104	104	104
x7dn	66/15	622	indust	primes	?	378	378/538	538	538
xparc	41/73	551	indust	primes	?	140	140/254	254	254

Table B.1. Optimum Results for the Berkeley PLA Test Set.

References

[BaM84] M. Bartholomeus and H. D. Man, Prestol-II: Yet Another Logic Minimizer for Programmed Logic Arrays, Draft, Nov. 1984.

[BaM85] M. Bartholomeus and H. D. Man, "Prestol-II: Yet Another Logic Minimizer for Programmed Logic Arrays", *Proc. Int. Symp. Circ. Syst.*, June 1985, 58.

[BMH84] R. K. Brayton, C. McMullen, G. D. Hachtel and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[DAR86] M. R. Dagenais, V. K. Agarwal and N. C. Rumin, "McBoole: A New Procedure for Exact Logic Minimization", *IEEE Trans. on CAD*, Jan. 1986, 229-238.

[DeS83] G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications", *IEEE Trans. on CAD*, July 1983, 151-167.

[De83] G. De Micheli, *Computer Aided Synthesis of PLA-based Systems*, PhD Thesis, University of California, Berkeley, 1983.

[DeB84] G. De Micheli and R. Brayton, "KISS: A program for Optimal State Assignment of Finite State Machines", *Proc. IEEE Int. Conf. on Comp. Aid. Des. (ICCAD)*, Nov. 1984, 209-211.

[De84] G. De Micheli, "Optimal Encoding of Control Logic", *Proc. Int. Conf. on Comp. Des. (ICCD)*, 1984, 16-22.

[DBS85] G. De Micheli, R. K. Brayton and A. Sangiovanni-Vincentelli, "Optimal State Assignment For Finite-State Machines", *IEEE Trans. on CAD*, July 1985, 269-285.

[De85] G. De Micheli, "Symbolic Minimization of Logic Functions", *Proc. IEEE Int. Conf. on Comp. Aid. Des. (ICCAD)*, Nov. 1985, 293-295.

[FlM75] H. Fleisher and L. I. Maissel, "An Introduction to array logic", *IBM J. of R&D 19* (Mar. 1975), 98-109.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, 1979.

[HNS82] G. D. Hachtel, A. R. Newton and A. Sangiovanni-Vincentelli, "An Algorithm for Optimal PLA Folding", *IEEE Trans. on CAD*, Jan. 1982, 63-76.

[HCO74] S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization", *IBM J. of R&D*, Sep. 1974, 443-458.

[LBH75] J. C. Logue, N. F. Brickman, F. Howley, J. W. Jones and W. W. Wu, "Hardware Implementation of a small system in programmable logic arrays", *IBM J. of R&D 19* (Mar. 1975), 110-119.

[Mah84] G. H. Mah, "PANDA: A PLA Generator for Multiple Folded PLAs", UCB M84/95, University of California Electronics Research Laboratory, May 1984.

[McC56] E. J. McCluskey, "Minimization of Boolean Functions", *Bell System Tech. J. 35* (Apr. 1956), 1417.

[McS84] C. McMullen and J. Shearer, "Prime Implicants, Minimum Covers, and the Complexity of Logic Simplification", *Draft*, 1984.

[Qui55] W. V. Quine, "A Way to Simplify Truth Functions", *Am. Math. Monthly 62* (Nov. 1955), 627.

[Rot80] J. P. Roth, *Computer Logic, Testing, and Validation*, Computer Science Press, 1980.

[SKF85] A. D. Samples, M. Klein and P. Foley, "Soar Architecture", UCB/Computer Science Dpt. 85/226, University of California, Mar. 1985.

[Sas78] T. Sasao, "An Application of Multiple-Valued Logic to a Design of Programmable Logic Arrays", *Proc. 8th Int. Symp. on Mult. Val. Logic (ISMVL)*, 1978.

[Sas81] T. Sasao, "Multiple-Valued Decomposition of Generalized Boolean Functions and the Complexity of Programmable Logic Arrays", *IEEE Trans. on Computers c-30* (Sep. 1981), 635-643.

[Sas82] T. Sasao, "Comparison of Minimization Algorithms for Multiple-Valued Expressions", *Submitted to Proc. 12th Int. Symp. on Mult. Val. Logic (ISMVL)*, 1982.

[Sas83] T. Sasao, "A Fast Complementation Algorithm for Sum-of-Products Expressions of Multiple-Valued Input Binary Functions", *Proc. 13th Int. Symp. on Mult. Val. Logic (ISMVL)*, 1983.

[Sas84a] T. Sasao, "Tautology Checking Algorithms for Multiple-Valued Input Binary Functions and Their Application", *Proc. 14th Int. Symp. on Mult. Val. Logic (ISMVL)*, 1984.

[Sas84b] T. Sasao, "Input Variable Assignment and Output Phase Optimization of PLA's", *IEEE Trans. on Computers c-33* (Oct. 1984), 879-894.

[Sas] T. Sasao, "Corrections and Addition to Input Variable Assignment and Output Phase Optimization of PLA's", *Personnel Communication*, .

[Sim83] P. Simanyi, "POP Reference Manual", *Berkeley CAD Tools Manual*, Sep. 1983.

[Su72] Y. H. Su and P. T. Cheung, "Computer Minimization of multi-valued Switching functions", *IEEE Trans. on Computers c-21* (1972), 995-1003.

[Tay81] G. Taylor, "Compatible Hardware for Division and Square Root", *Proc. 5th Symposium on Computer Arithmetic*, May 1981, 127-134.

[Tis67] P. Tison, "Generalization of Consensus Theory and Application to the minimization of Boolean Functions", *IEEE Trans. on Computers c-16* (Aug. 1967), 446.