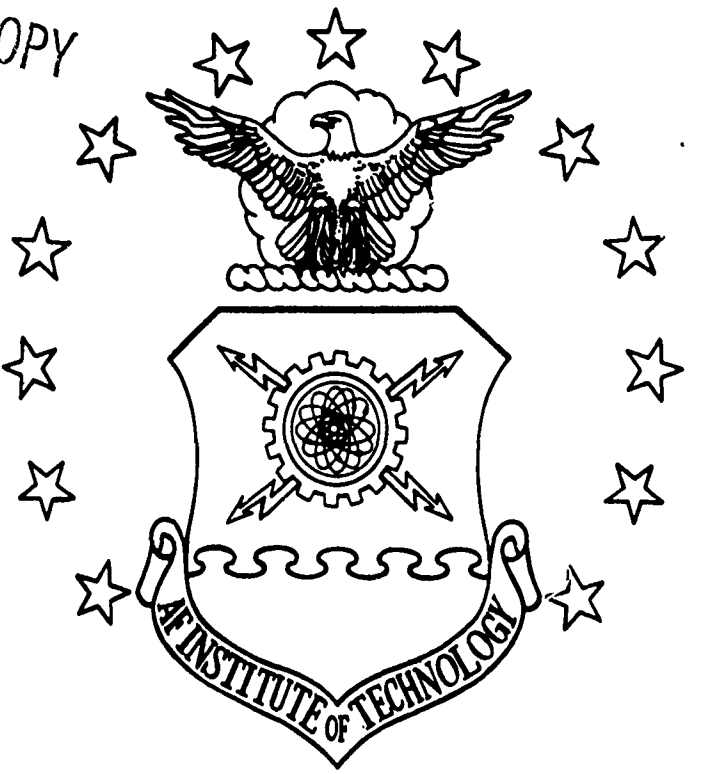


Recursive Optimization
of
Digital Circuits

THESIS

Eric John Knutson
Captain, USAF

AFIT/GCE/ENG/90D-03

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/90D-03

Recursive Optimization of Digital Circuits

THESIS

Eric John Knutson
Captain, USAF

AFIT/GCE/ENG/90D-03

AFIT/GCE/ENG/90D-03

Recursive Optimization of Digital Circuits

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Eric John Knutson, B.S.E.E.

Captain, USAF

December 14, 1990

Acknowledgments

I would like to dedicate this thesis to my wife, Leah, for the countless sacrifices she made on my behalf. For without her continual love and support, none of this would have been possible. I would like to thank Captain James Kainec for always being there to answer all my questions, no matter how moronic they were. I would like to thank the members of my committee, Doctor Matthew Kabrisky and Captain Robert Hammell II for reviewing this thesis and adding their constructive comments. I would especially like to thank Doctor Frank Brown, my thesis advisor, for his guidance and support throughout this research effort. Though the task seemed insurmountable at times, he provided the positive reinforcement to keep me going. Finally, I would like to thank my parents, Bernie and June. They instilled in me the conviction to pursue my goals and to never settle for second best. They have always been there when I needed them and certainly deserve to share in any of my accomplishments.

Eric John Knutson

Table of Contents

List of Figures

List of Tables

Abstract

The goal of this thesis is twofold: first, to identify the advantages and disadvantages of existing optimization systems and second, to develop an optimization system that uses Boolean principles to generate a recursive realization of combinational logic. Current multi-level optimization systems fall into two categories: local optimization which removes redundancy by pattern matching on a local scale and global optimization which works with the equations that specify a circuit rather than with the circuit implementation itself. While global systems are very flexible and can produce near-optimal solutions, they are inherently complex. This research effort demonstrates that an effective global optimization system can be built upon sound Boolean principles. A recursive optimization system built in Scheme was thoroughly evaluated. The system achieved gate-input reductions as high as 52 percent. Subsequent modifications targeted improving the system's speed and effectiveness. As a result of these efforts, the optimization speed for a variety of sample specifications was doubled. Other findings led to a better understanding of this approach and showed that it is a viable technique for the optimization of digital circuits.

Recursive Optimization of Digital Circuits

I. Introduction

1.1 Background

Since the advent of the first electronic computers back in the early 1940s, scientists and engineers have sought to develop increasingly complex circuits that are smaller, faster, and more reliable than ever before. Because of remarkable advancements in integrated circuit (IC) technology, computers that used to weigh several tons and occupy entire rooms can now be found on a single chip. Although progress in semiconductor technology continues today, it is unlikely that we will see improvements on the order of magnitude that we have seen over the last 40 years. As a result, the emphasis is shifting to finding optimal or near-optimal circuit designs which reduce cost (circuit area), propagation delay (speed) or a desired combination of both.

Optimizing digital logic circuits makes even more sense when we consider that typically "twenty to 50 percent of the active area of most semi-custom integrated circuits is devoted to combinational logic (31)." Over the years many different approaches have been taken in the development of efficient algorithms for logic synthesis and optimization. Despite considerable progress, the synthesis of non-trivial, digital circuits (deciding how to partition the logic, in what form to implement pieces of the logic, and what layout style to use) is still largely a manual process (14).

The long-term goal of logical design is to build a system that will accept a functional specification for a logic network and automatically generate an optimal, technology-specific implementation that is comparable in quality to that of an experienced designer. According to Karen Bartlett, "automating the synthesis and optimization of combinational logic reduces the design time, improves the size and speed of the circuitry and guarantees functional correctness (5)."

This realization, coupled with the increasing availability of Computer-Aided Design (CAD) tools and new Artificial Intelligence (AI) techniques, has caused research into auto-

matic synthesis and optimization of digital circuits to blossom over the last decade. With these tools in hand and ideas in mind, one can address the problems that have hindered the development of an ideal logic optimization system.

1.2 Statement of the Problem

Despite remarkable gains in the past few years, there is still a tremendous need for an efficient, general-purpose algorithm to optimize multi-output logic circuits. Current algorithms fall into two categories: *local optimization*, which removes redundancy by pattern matching on a local scale, and *global optimization*, which works with the equations that specify a system rather than the circuit implementation itself. Global optimization is potentially more powerful than local techniques, but it is also inherently complex and currently quite slow in comparison (77). It is imperative that we develop a further understanding of available state-of-the-art logic optimization techniques and the principles of Boolean reasoning if we hope to improve some of the deficiencies associated with a global approach.

1.3 Research Objectives

To overcome some of the problems associated with the global optimization of multi-level logic circuits, each of the following research objectives will be addressed:

- To analyze, compare and contrast the current state-of-the-art techniques in circuit synthesis and optimization.
- To identify current areas of active research including the use of AI principles, simulated annealing, binary decision diagrams, and Boolean reasoning methodologies to solve the problem.
- To investigate the idea of a recursive realization of a combinational logic circuit, which takes advantage of existing signals to produce new ones.
- To develop a simple, recursive optimization system that uses global methodologies and is built on a sound theoretical foundation.

- To explore new search strategies, heuristics, and Boolean reasoning techniques, designed to improve the speed and effectiveness of our optimization system.
- To make recommendations based on the experience accumulated.

1.4 Scope

To keep the scope of this research effort at a manageable level, several limitations were imposed. The effort was focused on the design and development of a global optimization system that generates a recursive realization of combinational logic. It was built utilizing the reasoning-toolset BORIS (Boolean Reasoning In Scheme) developed by F.M. Brown at the Air Force Institute of Technology (AFIT) (22). It was designed to accept a set of Boolean equations defining the behavior of a multiple-output, combinational circuit and return a set of equations that satisfies the specification at a reduced cost. The circuit optimization process will not target a particular implementation technology. In other words, our reduced equations will map directly into circuits consisting of AND, OR and NOT gates. The results of this research effort could be extended to include optimization around a particular technology at some future date.

This research addresses an innovative new approach to optimization that generates a recursive realization of combinational logic (22). It involves performing a dependency analysis to determine minimal subsets of inputs and outputs that can be used to generate a given output; these sets are called *minimal determining subsets* and are described in further detail later. While this technique is quite successful in reducing the cost of numerous circuits, it is currently computationally intensive and doesn't always find an optimal solution. Our goal was to evaluate and improve the speed and accuracy of the recursive optimization system that was developed using the BORIS toolset. At the same time, an effort was made to modify several aspects of the BORIS toolset to make it more versatile in its ability to handle equations of any practical form and size. The completion criterion for this endeavor would be achieved when we finished the development of this optimization system, compare its effectiveness to other current systems and draw conclusions based on the results.

1.5 Assumptions

We assume that the reader has a basic understanding of Boolean algebra and fundamental circuit design techniques. If that is not the case, there are a variety of good sources available including *Digital Logic and Computer Design* by M. Morris Mano (73) and *Boolean Reasoning* by Dr. Frank M. Brown (22). We will build upon some of these fundamental Boolean principles as required.

Other important assumptions were that the optimization algorithm has a sound theoretical basis with results that are verifiable. While the results should be verifiable, no attempt was made to formally prove such is the case. The system was initially developed on an IBM-compatible, personal computer with plans to eventually port it over to a faster, workstation or minicomputer sometime in the future. No attempt was be made to address every aspect of global optimization, but rather the key issues, as they apply to our objectives, were brought to light.

To simplify the optimization task, but by no means to diminish their importance, several other assumptions were made (66):

- The final, optimized circuit consisted of AND, OR and NOT gates with no attempt made to adapt it to an alternative technology.
- All circuit components were considered to be "ideal", consisting of a unit delay.
- Limitations were not placed on the ultimate shape of the circuit.
- Any constraints on the maximum propagation delay through the circuit were ignored.
- "Race conditions" that may exist because of an uneven propagation of the signals through the circuit were ignored.

1.6 Standards

To lend credibility to this research effort, it was imperative that we conduct extensive tests, comparing our results with ones that have been previously established. One measure of our success is obviously the ability of our system to produce circuits with a lower cost than previously attainable using automated techniques. Another measure of success

is how much we can improve the speed of our system while not sacrificing any of its effectiveness. While both of these are important we would ultimately like to compare our system against other commercial or prototype systems. One way to do this is to use a set of previously established benchmarks such as the one developed by Aart J. de Geus for the 1986 Design Automation conference (32). Unfortunately our system did not reach the level of sophistication where a comparison with these benchmarks was possible. For example we do not map our results into a particular target technology. However, by limiting the scope of our effort, we were able to concentrate on critical aspects of the optimization system.

Another important consideration was how to measure the cost of a circuit. Cost can be measured as a function of the number of gate inputs, the number of gates, the maximum time delay through the circuit, or a combination of all of these. How cost is calculated is a function of the intended application of a given circuit. It was important that we establish some guidelines for determining cost, laying out in detail our specific design criteria.

1.7 Approach/Methodology

The first step in developing an effective global optimization system was to explore how the current, state-of-the-art optimization systems operate and to analyze new approaches to this problem. An extensive review of current literature was conducted and culminated with nearly a hundred, useful sources. Pertinent information gathered from these sources, as well as a detailed understanding of Boolean reasoning principles, provided the foundation upon which our global optimization system was eventually developed.

After comparing and contrasting the latest in multi-level optimization systems and techniques, we focused our efforts on one particular global methodology: the recursive realization of combinational logic circuits. We established the necessary foundation of theoretical information that would enable us to intelligently formulate the specific requirements necessary to design and develop such a system. We followed this with a more detailed problem analysis, identifying the specific stages necessary in the development of our system, and the problems that we were likely to encounter. We addressed such aspects as the types of computers, search algorithms, heuristics, and other tools that would be

used. We also justified our selection of a particular programming language to solve the problem.

Once we developed the requirements, and a detailed plan of attack, we were able to proceed with the development of the software. This phase was broken down into three primary stages: design, coding, and testing. All three of these stages proceeded concurrently. The testing stage involved a rigorous validation to determine if the system performed as expected and to establish its overall credibility. Most of our efforts, concerning the software development, focused on improving the speed of the global optimization system while not sacrificing any of its effectiveness.

Finally, it was necessary to evaluate the performance of the modified optimization system. The results of these tests were tabulated, analyzed, and presented for further review. We concluded this research effort by summarizing all of the results and presenting the overall conclusions.

1.8 Maximum Expected Gain

The ultimate goal is to improve our understanding of global optimization techniques and to make qualified recommendations for the direction this research should take in the future. Our success will provide a foundation for continued research into this area. Improved circuit design techniques, will enable us to build inexpensive digital computers and electronic systems that are smaller, faster, and more accurate than ever before.

1.9 Overview

The present chapter provides a general introduction to the problems associated with synthesizing optimal digital circuits and the growing importance of this field. It outlines the overall research objectives along with the scope, assumptions, standards and approach.

Chapter 2 is designed to provide a brief theoretical background on some of the key principles of Boolean algebra. It reviews a variety of general concepts, some which may seem familiar, others which may not. They are all necessary to provide a basis for further discussions and more detailed theoretical development.

Chapter 3 provides a summary of the current knowledge in the field. It is based on an extensive literature review and includes a historical perspective of circuit synthesis and optimization. It highlights the current state-of-the-art optimization systems, pointing out their advantages as well as their drawbacks. It introduces and contrasts multi-level optimization techniques versus the better understood two-level approach. It compares and contrasts local redesign techniques versus a more flexible global approach. It analyzes a variety of new approaches to the problem and new tools, including AI techniques, binary decision diagrams, simulated annealing, minimal determining subsets and others.

Chapter 4 lays the theoretical foundation for the recursive realization of multi-level logic circuits. It expands on a process that involves: transforming a behavioral specification into a system of Boolean equations, reducing the system of equations into a single equation that represents the circuit, performing a dependency analysis to determine the relationship among variables, and to use this knowledge to determine an optimal multi-level representation that expresses outputs recursively in terms of inputs and previously defined outputs.

Chapter 5 discusses how to build a global optimization system. It justifies the selection of Scheme as a programming language. It illustrates how the scope of our project fits into a general circuit optimization scheme. It introduces us to the BORIS toolset and discusses the structure and data flow of the recursive optimization system.

A detailed analysis of the specific problems that will be attacked in this research effort and the approaches aimed at solving them are developed in Chapter 6. It includes investigations into the performance of the BORIS design system and discusses the integration of a tabular design filter. It discusses modifications to the system in an attempt to find even better solutions. It provided a framework upon which the software modifications and updates were developed.

Chapter 7 presents the results of this research effort. A typical optimization session using the recursive design system is illustrated. The performance of the tabular design filter is summarized along with the efficiency improvements and cost reductions that were achieved using this system. This chapter also highlights some interesting results that

warrant further investigation.

The final chapter, Chapter 8, presents an overall summary of this research effort. It assesses the strengths and weaknesses of the optimization system, provides some lessons learned and makes recommendations for the direction of further research.

II. Review of Important Boolean Concepts

2.1 Fundamentals of a Boolean Algebra

> Boolean Algebra forms a cornerstone of computer science and digital circuit design. Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions. (20)

Before we begin a detailed discussion of optimization techniques, it is imperative that we arrive at the problem with an adequate theoretical background. This section attempts to bridge any gaps in knowledge by building a sound, theoretical foundation. To accomplish this we will highlight some of the fundamental concepts of a Boolean algebra. The application and relevance of these concepts will become clearer as we proceed.

2.1.1 Postulates for a Boolean Algebra. All attempts at developing new global optimization techniques are linked by one common thread. They must be based on the sound principles of a Boolean algebra. A Boolean algebra is often denoted by a quintuple

$$< \mathbf{B}, +, \cdot, 0, 1 > \tag{2.1}$$

where $\mathbf{B}$ is a set called the *carrier* , $+$ and $\cdot$ are binary operations on $\mathbf{B}$, and 0 and 1 are distinct members of $\mathbf{B}$ (22). Huntington developed a set of six postulates used to define a Boolean algebra (58). These postulates are by no means unique; other sets have been developed (73). The algebraic system defined by (2.1) already satisfies two of these postulates: there is closure with respect to $+$ and $\cdot$ and there exists at least two elements in $\mathbf{B}$, in this case 0 and 1. The remaining four postulates that also need to be satisfied to define a valid Boolean algebra are:

1. **Commutative Laws.** For all a, b in $\mathbf{B}$,

$$a + b = b + a \tag{2.2}$$

$$a \cdot b = b \cdot a\,. \tag{2.3}$$

2. **Distributive Laws.** For all a, b, c in **B**,

$$a + (b \cdot c) = (a + b) \cdot (a + c) \tag{2.4}$$

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c) . \tag{2.5}$$

3. **Identities.** For all a in B,

$$0 + a = a \tag{2.6}$$

$$1 \cdot a = a . \tag{2.7}$$

4. **Complements.** For every a in **B**, there exists an element a' in **B** such that

$$a + a' = 1 \tag{2.8}$$

$$a \cdot a' = 0 . \tag{2.9}$$

Note that the " ′ " symbol stands for *complementation.*

2.1.2 The Inclusion Relation. A very important relationship, as we will soon discover, is the *inclusion relation.* This relation on a Boolean algebra is denoted by $\leq$ and defined as follows (22):

$$a \leq b \iff a \cdot b' = 0 . \tag{2.10}$$

It is helpful to draw an analogy between the inclusion relation and the algebra of subsets of a set. An isomorphism exists between Equation (2.10) and Equation (2.11) shown below:

$$A \subseteq B \iff A \cap B' = \emptyset . \tag{2.11}$$

We can visualize this relationship by use of a Euler diagram, where A and B are subsets of a universal set S. This is shown below in Figure 2.1.

Boolean relations are often expressed as *intervals* (segments) between an upper and lower bound. Let a and b be members of a Boolean algebra **B**, and assume that $a \leq b$.

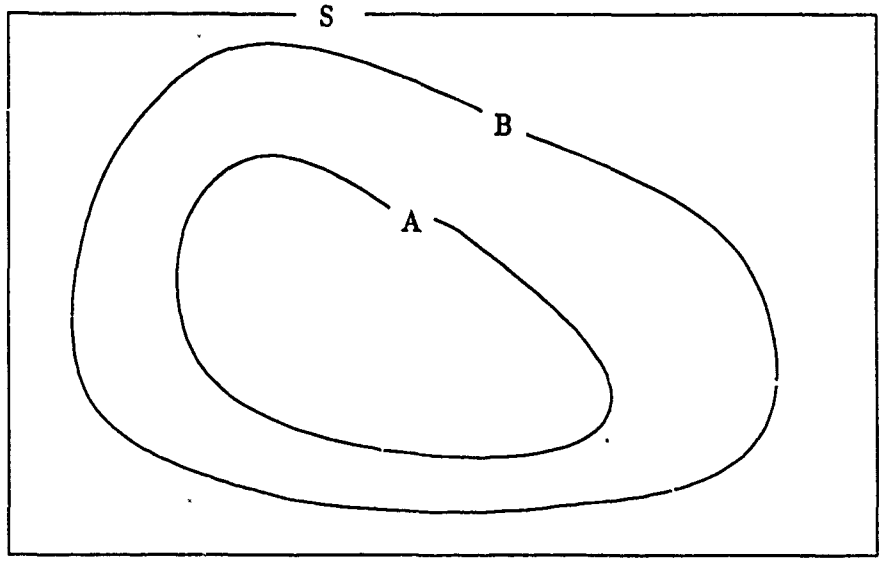


Figure 2.1. Euler Diagram

The interval $[a, b]$ is the set of elements of **B** lying between a and b, i.e.,

$$[a, b] = \{x \mid x \in \mathbf{B} \text{ and } a \leq x \leq b\} . \tag{2.12}$$

Once again it is easy to visualize this by looking at the Euler diagram in Figure 2.1. The interval $[A, B]$ consists of all elements outside of A but within B.

2.1.3 Some Useful Properties. It is often inconvenient to formulate all proofs based on the original Boolean postulates themselves. Consequently, to facilitate the manipulation of Boolean expressions, a number of formal properties have been developed which can be proven from the original postulates and the definition of an inclusion relation. Below is a list of some of the more useful properties defined for all a, b, c in a Boolean algebra (22):

1. **Associativity.**

$$a + (b + c) = (a + b) + c \tag{2.13}$$

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c . \tag{2.14}$$

2. **Idempotence.**

$$a + a = a \tag{2.15}$$

$$a \cdot a = a . \tag{2.16}$$

3. **Boundedness.**

$$a + 1 = 1 \tag{2.17}$$

$$a \cdot 0 = 0 . \tag{2.18}$$

4. **Absorption.**

$$a + (a \cdot b) = a \tag{2.19}$$

$$a \cdot (a + b) = a . \tag{2.20}$$

5. **Involution.**

$$(a')' = a . \tag{2.21}$$

6. **DeMorgan's Laws.**

$$(a + b)' = a' \cdot b' \tag{2.22}$$

$$(a \cdot b)' = a' + b' . \tag{2.23}$$

7.

$$a + a' \cdot b = a + b \tag{2.24}$$

$$a \cdot (a' + b) = a \cdot b . \tag{2.25}$$

8. **Consensus.**

$$a \cdot b + a' \cdot c + b \cdot c = a \cdot b + a' \cdot c \tag{2.26}$$

$$(a+b) \cdot (a'+c) \cdot (b+c) = (a+b) \cdot (a'+c) . \tag{2.27}$$

9.

$$a \leq a+b \tag{2.28}$$

$$a \cdot b \leq a . \tag{2.29}$$

2.1.4 Equivalent Boolean Equations. It is often convenient to express a Boolean equation in the form $f = 0$ or $g = 1$. The following properties can be proven directly from the Boolean postulates.

1. An arbitrary Boolean equation can be transformed into the form $f = 0$ using the following relationship:

$$a = b \Longleftrightarrow a' \cdot b + a \cdot b' = 0 . \tag{2.30}$$

Since $(a' \cdot b + a \cdot b')$ is the Exclusive-OR of a and b, this equation can be rewritten as shown below with $\oplus$ representing an Exclusive-OR:

$$a = b \Longleftrightarrow a \oplus b = 0 . \tag{2.31}$$

2. Similarly, an arbitrary Boolean equation can be transformed into the form $g = 1$ using the following relationship:

$$a = b \Longleftrightarrow a' \cdot b' + a \cdot b = 1 . \tag{2.32}$$

Since $(a' \cdot b' + a \cdot b)$ is the Exclusive-NOR of a and b, this equation can be rewritten as shown below with $\odot$ representing an Exclusive-NOR:

$$a = b \Longleftrightarrow a \odot b = 1 . \tag{2.33}$$

3. If we let a and b be members of a Boolean algebra, then the following properties are valid:

$$a = 0 \text{ and } b = 0 \Longleftrightarrow a + b = 0 \tag{2.34}$$

$$a = 1 \text{ and } b = 1 \Longleftrightarrow a \cdot b = 1\,. \tag{2.35}$$

2.1.5 Boole's Expansion Theorem. Sometimes called "the fundamental theorem of Boolean algebra (22)," Boole's Expansion Theorem forms the foundation for computation with Boolean functions. If f is an n-variable Boolean function, then f has the expansions shown below:

$$f(x_1, x_2, \ldots, x_n) = x_1' f(0, x_2, \ldots, xn) + x_1 f(1, x_2, \ldots, x_n) \tag{2.36}$$

$$f(x_1, x_2, \ldots, x_n) = [x_1' + f(1, x_2, \ldots, xn)][x_1 + f(0, x_2, \ldots, x_n)] \tag{2.37}$$

2.2 Boolean Functions and Formulas

2.2.1 What Is A Boolean Function? A Boolean function is actually a mapping that can be described by a Boolean formula. Given a Boolean algebra **B**, the set of *Boolean formulas* on the n symbols $x_1, x_2, \ldots, x_n$ is defined by the following rules (22):

1. The elements of **B** are Boolean formulas.
2. The symbols $x_1, x_2, \ldots, x_n$ are Boolean formulas.
3. If g and h are Boolean formulas, then so are
   (a) $g + h$
   (b) $g \cdot h$
   (c) g' .
4. A string is a Boolean formula if and only if it can be formed from a finite number of applications of rules 1, 2 and 3.

Examples of Boolean formulas include x, x', $x + y$ and $xy' + (z \cdot (w + u))$.

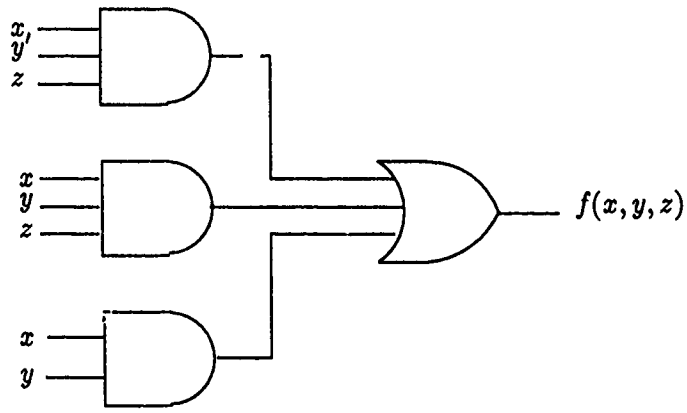


Figure 2.2. Circuit Implementation of $f(x,y,z) = xy'z + xyz + xy'$

With this in mind, an n-variable Boolean function $f : \mathbf{B}^n \rightarrow \mathbf{B}$ is called a *Boolean function* if and only if it can be expressed by a Boolean formula. Any given Boolean function may have a number of corresponding Boolean formulas.

In this research effort the focus will be confined to a two-element Boolean algebra. In this context, a Boolean formula is formed with binary variables, the two binary operators OR and AND, the unary operator NOT, parenthesis, and equal sign (73). It can be shown that any digital logic circuit can be described mathematically by a Boolean function of the two-element Boolean algebra, $f : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$. For a given value of the variables, the values of the function can be either 0 or 1. Consider, for example, a three variable Boolean function, $f : \mathbf{B}_2^3 \rightarrow \mathbf{B}_2$, expressed by

$$f(x,y,z) = xy'z + xyz + xy' \,. \tag{2.38}$$

The two-level AND-to-OR circuit corresponding to this function is given in Figure 2.2. Here we see that each term (conjunction of literals) in (2.38) corresponds to an AND gate. The disjunction of these terms can be represented by passing the outputs of all the AND gates through an OR gate. Since this equation contains no parentheses and consists of a sum of terms, its Boolean representation is in a *sum-of-products* form.

2.2.2 Boolean Function Representation. There are a number of ways to represent a Boolean function, one of the more common being a *truth table.* A truth table

enumerates the output values of a function, given every possible input combination. For a function of n binary variables, there are 2^n rows in the associated truth table. The truth table for (2.38) is shown in Table 2.1.

While a truth table uniquely defines the desired behavior of a circuit, there are a multitude of algebraic expressions that can specify the same function (73). Finding one with the least cost is the central focus of circuit optimization.

2.2.3 Relationships Among Variables. The *Venn diagram* was developed to help visualize the relationship among variables of a Boolean expression. An example of a Venn Diagram for the circuit described by (2.38) is shown in Figure 2.3. M. Mano points out that the diagram consists of a rectangle, inside of which are overlapping circles, one for each variable (73). We designate all points inside a given circle as belonging to the named variable and all points outside the circle as not belonging to that variable. For example if we are inside the circle labeled x then we say $x = 1$; otherwise, if we are outside the circle $x = 0$.

In Figure 2.3 we see that the three overlapping circles create eight distinct areas. Each area represents one of the possible combinations of variables. In general for an n-variable function, there will be 2^n possible unique areas on the Venn diagram. "Venn diagrams may be used to illustrate the postulates of Boolean algebra or to show the validity of theorems (73)." As an example, we can visually observe that the same Venn diagram shown in Figure 2.3 can be produced with the function $f = xy' + xyz$. Notice that the term $xy'z$ is missing from the equation. It turns out that the term is redundant, adding

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.1. Truth Table for $f(x,y,z) = xy'z + xyz + xy'$

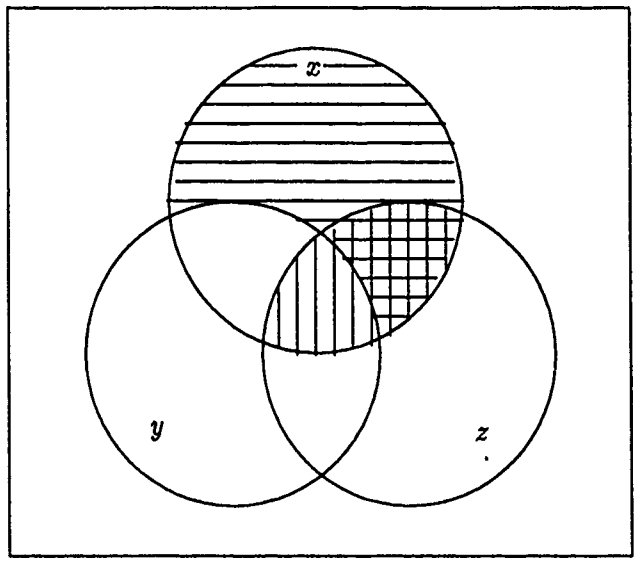


Figure 2.3. Venn Diagram for $f(x,y,z) = xy'z + xyz + xy'$

no new information to the equation. This illustrates one of the Boolean properties known as *absorption.*

2.2.4 Canonical Forms. Any Boolean function can be expressed by an infinite number of Boolean formulas. It would be convenient if there were a restricted class of Boolean formulas where each Boolean function had a single, unique representation. Formulas in such a class do exist and are considered as a *canonical form* (22). A number of canonical forms have been developed. We will focus on two of the most common: the *minterm canonical form* and the *Blake canonical form.*

2.2.4.1 Minterm Canonical Form. A *minterm* is a term in a formula of n variables which contains all variables of the formula either in complemented or uncomplemented form. This concept is illustrated in Table 2.2 for three variable minterms along with their shorthand representation. We should note that for a function $f(x,y,z)$, the values

$$f(0,0,0), f(0,0,1), \ldots, f(1,1,1) \tag{2.39}$$

are called the *discriminants* of the function.

	Minterms	
xyz	Term	Shorthand Notation
0 0 0	$x'y'z'$	m_0
0 0 1	$x'y'z$	m_1
0 1 0	$x'yz'$	m_2
0 1 1	$x'yz$	m_3
1 0 0	$xy'z'$	m_4
1 0 1	$xy'z$	m_5
1 1 0	xyz'	m_6
1 1 1	xyz	m_7

Table 2.2. Minterms for Three Binary Variables

A formula in *minterm canonical form* is a sum-of-products (SOP) formula in which all the terms are minterms (65). Each minterm represents one of the distinct areas shown in a Venn diagram. Assuming a two-element Boolean algebra, a Boolean function may be expressed algebraically from this diagram by taking the OR of all the terms represented by shaded regions. Similarly a Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of the variables which produces a 1 in the function, and then taking the OR of all the terms (73). As an example, (2.38) can be represented in minterm canonical form as follows:

$$f(x,y,z) = xy'z' + xy'z + xyz\,. \tag{2.40}$$

Using a shorthand notation, we can express this Boolean function as a sum (OR) of minterms as shown in (2.41).

$$f(x,y,z) = \sum m(4,5,7)\,. \tag{2.41}$$

It is important to differentiate the terminology applied to the three categories of minterms. The first is the *on-set* which describes all minterms for which the function evaluates to '1'. In the example above, minterms 4, 5 and 7 belong to the on- set. The *off-set* is the set of all minterms for which the function evaluates to '0'. In the example above, minterms 0, 1, 2, 3 and 6 belong to the off-set. The third and final category is the *don't-care set* usually denoted by an 'x' or a 'd'. It represents those minterms for which we

don't care if they evaluate to '0' or '1'. *Incompletely specified functions* are those functions that contain "don't-care" conditions.

2.2.4.2 Blake Canonical Form. One of the key advantages of a Blake canonical form (BCF) is that, not only is it canonical, but it is also significantly reduced. A term p is called an *implicant* of a Boolean function f if $p \leq f$. Clearly if a function is expressed in SOP form, all of the terms are implicants of f. An implicant of f is considered a *prime implicant* if the removal of any of its literals results in its no longer being an implicant (85). The Blake canonical form of a function f consists of the disjunction of all the prime implicants of f.

The efficient transformation of a formula to its Blake canonical form has been the topic of numerous research efforts. It is possible to accomplish this task through the careful application of the fundamental Boolean postulates and their associated properties. Methods for generating BCF(f) include *exhaustion of implicants, iterated consensus*, and *multiplication* (65). One of the more popular methods today uses implication and was developed by W.V. Quine (86).

2.3 Boolean System

2.3.1 What Is A Boolean System? An *n-variable Boolean system* is a collection

$$\begin{aligned} p_1(X) &= q_1(X) \\ p_2(X) &= q_2(X) \\ &\vdots \\ p_k(X) &= q_k(X) \end{aligned} \tag{2.42}$$

of simultaneously asserted equations (22). p_i and q_i are n-variable Boolean functions on B and X represents the vector $(x_1, x_2, \ldots, x_n)$. Even though we have defined a Boolean system as a collection of equations, we know that an inclusion relation can be easily transformed into an equation as shown by Equation (2.10).

Typically a circuit specification will be defined by a Boolean system rather than a

single Boolean equation. Each equation generally specifies one output of a multi-output system. When attempting to optimize a multi-output circuit design, if we treat each equation independently we reduce our chances of arriving at an optimal or near-optimal solution. This is because we fail to take advantage of similiarities between the equations such as identical terms. For this reason, it would be advantageous to transform a Boolean system into a single Boolean equation. Fortunately such a transformation is possible utilizing a technique called *Boolean reduction.*

2.3.2 Boolean Reduction. Any system of Boolean equations can be reduced to a single equation. By applying the property (2.31), the Boolean system represented by (2.42) can be reduced to the equivalent system shown below:

$$\begin{aligned} p_1(X) \oplus q_1(X) &= 0 \\ p_2(X) \oplus q_2(X) &= 0 \\ &\vdots \\ p_k(X) \oplus q_k(X) &= 0\,. \end{aligned} \tag{2.43}$$

This system of equations can in turn be transformed into a single Boolean equation by using the property described by (2.34). This single equation is

$$f(X) = 0\,, \tag{2.44}$$

where f is defined by

$$f = \sum_{i=1}^{k} p_i \oplus q_i\,. \tag{2.45}$$

Similarly we can show that any Boolean system can be converted into the form $f(X) = 1$. The system of equations shown in (2.42) can be transformed into an equivalent

system using the property shown in (2.33):

$$\begin{aligned} p_1(X) \odot q_1(X) &= 1 \\ p_2(X) \odot q_2(X) &= 1 \\ &\vdots \\ p_k(X) \odot q_k(X) &= 1 . \end{aligned} \tag{2.46}$$

This system of equations can be transformed into a single Boolean equation using property described by (2.35). This single equation is

$$f(X) = 1 , \tag{2.47}$$

where f is defined by

$$f = \prod_{i=1}^{k} (p_i \odot q_i) . \tag{2.48}$$

The *normal form* representation described by 2.47 is advantageous for a number of reasons (22). It provides a standardized representation on which to base further synthesis and analysis. The function f, corresponding to a given system of Boolean equations, is unique. Finally, the normal form provides a way to deal with "don't-care" conditions in a uniform and convenient manner.

III. Overview of Digital Circuit Optimization Techniques

3.1 The Motivation for Optimizing Digital Circuits

Over the years, the need for effective circuit optimization tools has not changed, but the reasons for optimizing digital circuits have. The optimization of digital circuits has been an active area of research since the early 1950s. In those days digital systems like the revolutionary ENIAC (Electronic Numerical Integrator and Calculator) weighed several tons and contained thousands of resistors, vacuum tubes and other discrete components (55). Because of the exorbitant expense of logic gates in those days, early optimization efforts concentrated primarily on the reduction of discrete components in a given logic design.

With the advent of the transistor and later, integrated circuits, the cost of logic gates in the late 1960s and early 1970s was reduced dramatically. Because of this, there was a lack of interest in investing a lot of time in sophisticated optimization techniques when hand-simplification seemed to perform adequately for most designs.

It wasn't until the mid to late 1970s that the field of circuit optimization caught a second wind; it has been going strong ever since. A number of factors have contributed to this resurgence. The most prominent of these is the introduction of LSI (Large Scale Integration) and later VLSI (Very Large Scale Integration) technology. This involved placing an increasing number of logic gates on an ever decreasing amount of surface area. As the variety of new applications for digital circuits multiplied exponentially, their complexity increased likewise. Whereas earlier circuits consisted of five to ten variables and a few hundred gates, today it isn't uncommon to find 30 or more variables, multiple outputs and thousands or in some cases millions of gates. As you can imagine, the human designer's ability to synthesize such complex circuits has placed serious limitations on the achievable goals. Not only are human designers prone to making mistakes, but their designs are often quite inefficient.

By the early 1980s Computer-Aided Design (CAD) systems were becoming quite sophisticated and accessible throughout the world. This led to an effort to overcome some of the obstacles in circuit optimization, such as its inherent complexity, by automating the

process. Numerous systems have since been developed, some of which are proving to be commercially viable.

Recently, efforts to automate circuit optimization have focused on the use of Hardware Description Languages (HDLs). HDLs were designed as a means to document, model, and in some cases simulate complex VLSI designs. Automated logic synthesis is driven by the desire to take designs described in an HDL and automatically generate an optimized circuit (66). R.K. Brayton described it in more general terms as follows:

> A long-term goal for computer-aided design (CAD) systems is the automatic synthesis from a behavioral description to silicon, producing near-optimal results that meet the specifications set by the designer and that are competitive with or better than manually aided designs. (17)

One of the major obstacles in realizing this goal is the *efficiency* of the optimization algorithms. It has been shown that most VLSI optimization problems are nondeterministic polynomial-time complete (NP-complete for short) (98). In essence this means that they belong to a class of problems that can't be solved in polynomial time. Their time and space complexity often increases at an exponential rate. When automated on a computer, this places serious limitations on the circuit optimization system. Because of the required storage and computations, computerized optimization using classical approaches became quite impractical for problems with many variables (57). This has led to numerous efforts to find and develop efficient optimization algorithms. New approaches involve everything from improved heuristics and new ways to represent Boolean networks, to applying AI techniques, such as rule-based systems, to the problem.

Today, the basic objectives of circuit optimization include (17):

- minimizing the overall area of the design,
- minimizing the critical path delay time,
- improving the testability and verifiability of the synthesized logic.

Reducing the overall area of the design is important for a number of reasons. Obviously, smaller digital circuits can be placed in smaller areas (watches, radios, laptop computers,

etc.). Nowhere is this more important than in advanced aerospace vehicles where space on the aircraft is a highly critical asset. Another reason for reduction of the surface area is that "the cost of fabricating a circuit is actually an exponential function of the area, at least if the circuit is large (98)."

Reducing the critical path delay time will obviously speed up the circuit. The fastest circuit would be one in which signals have to travel through the fewest gates. Since every system of Boolean functions can be expressed in an equivalent SOP form, this representation translates into a circuit with a depth of two gates. While this typically represents the fastest circuit possible, its implementation often requires a great amount of surface area and may lead to an unacceptable fan-in at some gates. Thus there generally needs to be a compromise between a reduction of surface area and a reduction of propagation delay. Recently a number of systems have incorporated a means to optimize a circuit based on a such a compromise. One such system is SOCRATES, which will be discussed in more detail later.

Gaining in importance as circuits become more complex is the idea of developing digital systems that are easily easily tested to ensure that they function correctly. Fortunately, producing optimized circuits that are testable and verifiable is often a by-product of automated optimization techniques (17). Future design systems will generate a complete set of test vectors resulting from the circuit synthesis and optimization process.

Another purpose for optimization systems is to redesign a given circuit, translating it from one technology to another while taking full advantage of the features of the target technology. Many minimization systems may reduce a circuit into a system consisting of standard AND and OR gates. However, they may be implemented using NAND and NOR gates or a more current technology. When an optimized AND-OR circuit is transformed into a new technology, the resulting circuit is generally not optimal and can undergo further reduction. If the target technology is known from the beginning, it is possible to design and optimize a digital circuit with the target technology in mind. A good measure of the quality of an optimization system is its ability to incorporate new technologies as they are developed and to convert back and forth between technologies.

To be competitive in the future, circuit optimization systems need to be able to quickly produce verifiable circuits from a behavioral specification. "This capability will become increasingly important as the application-specific integrated circuit (ASIC) market continues to meet its rapid growth projections (17).

3.2 Two-Level Optimization Techniques

The area of logic synthesis is typically divided into *two-level synthesis* and *multi-level synthesis*. While both approaches have been around for well over 40 years, two-level techniques received much of the early attention. A two-level logic circuit, as its name implies, consists of, at most, two-levels of gates. An example is shown in Figure 2.2. It results from a direct translation of a Boolean, SOP formula into an equivalent circuit representation. It has long been the preferred approach because of its innate simplicity. Efforts have concentrated on reducing a function into a minimal SOP form. Early designers used a variety of manual simplification techniques such as Boolean simplification and map-based approaches. Eventually, systems like the Quine-McCluskey technique were introduced to automate the process. With the introduction of LSI and the resulting popularity of programmable logic arrays (PLAs), PLA minimization techniques eventually dominated two-level optimization research.

3.2.1 Boolean Simplification.

"The complexity of digital logic that implements a Boolean function is directly related to the complexity of the algebraic expression from which the function is implemented (73)." Boolean simplification was probably the first technique used to optimize the design of digital circuits. Boolean simplification involves applying the fundamental axioms and theorems of Boolean algebra to mathematically reduce (simplify) a given Boolean formula. By simplifying the formula we in turn simplify the circuit it represents. This concept can probably be best illustrated by the use of a simple example. Starting with (2.38), the following is a sequence of simplification steps that will produce an optimal result:

1. Original Boolean equation (2.38):

$$f(x,y,z) = xy'z + xyz + xy' . \tag{3.1}$$

2. Term one is eliminated through *absorption* with term three:

$$f(x,y,z) = xyz + xy' . \tag{3.2}$$

3. Terms one and two form a *consensus* term:

$$f(x,y,z) = xyz + xy' + xz . \tag{3.3}$$

4. Term one is eliminated through *absorption* with term three:

$$f(x,y,z) = xy' + xz . \tag{3.4}$$

If we translate (3.4) into its equivalent circuit representation shown in Figure 3.1, we see that the original circuit shown in Figure 2.2 has been significantly reduced; one three-input AND gate has been removed completely and another three-input AND gate has been reduced to a two-input AND gate. Although this may seem impressive on the surface, we must keep in mind that it was a very simple example. The method used to apply the theorems and axioms is often a trial and error technique which relies heavily on heuristics and the experience of the designer. As a result, automating Boolean simplification systems based on these techniques is a largely undeveloped science. However, it is getting more attention as the numerous advantages of global optimization systems become more apparent.

3.2.2 Karnaugh Map Technique. One of the early optimization approaches used a map-based reduction technique. This approach, originally developed by Veitch and later modified by Karnaugh, became known as the *Karnaugh Map Method.* It enables a designer to place a Boolean function on a map and reduce it by recognizing adjacent terms.

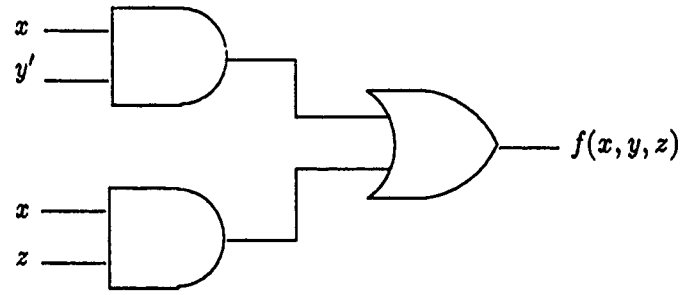


Figure 3.1. Circuit Implementation of $f(x, y, z) = xy' + xz$

Once again we will illustrate this technique with the use of an example. To reduce (2.38) a three-variable Karnaugh map, which consists of eight distinct areas, is required. The eight areas in the Karnaugh map correspond to the same eight areas in the Venn diagram (Figure 2.3) and the eight possible minterms. The reduction process begins by placing a '1' in each block which is covered by a term in the Boolean function. Next, one needs to recognize and combine adjacent terms. The rules regarding this process are summarized as follows (66):

- Blocks which are combined must be logically adjacent. Any given block has n adjacent blocks where n is the number of variables in the function represented by the map.
- The number of blocks combined must be a power of two.
- Blocks are combined to form the largest grouping possible.
- As few groups as possible are formed which cover all the blocks which enclose a '1'.

For our example, these steps resulted in the creation of two rectangles, each enclosing two 1's as shown in Figure 3.2. The left rectangle represents the area enclosed by xy'. The right rectangle represents the area enclosed by xz. The sum of these two terms forms the answer:

$$f(x, y, z) = xy' + xz \tag{3.5}$$

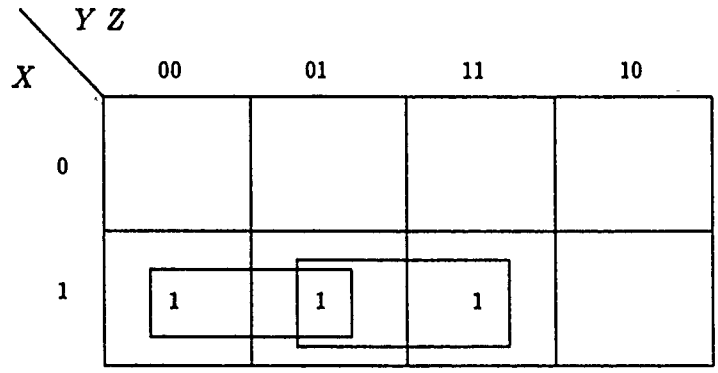


Figure 3.2. Karnaugh Map for $xy'z + xyz + xy' = xy' + xz$

As you can see the result is the same as that produced by visual observation of the Venn diagram. A more detailed treatment of Karnaugh map reduction techniques for a variety of examples is given by Morris Mano in his book *Digital Logic and Computer Design* (73). Mano points out that this method of simplification is convenient as long as the number of variables does not exceed five or six. As the number of variables increases, the excessive number of squares in the map makes it difficult for a human user to recognize patterns. For those reasons, Karnaugh map reduction techniques are only applied to the most simple circuit designs; such designs are becoming increasingly rare in this modern age of electronics.

3.2.3 Quine-McCluskey Method. In an effort to develop an automated circuit optimization system that doesn't rely on a human designer's skills, the Quine-McCluskey method was developed. Quine developed the original algorithm with McCluskey later improving on it. The Quine-McCluskey method takes a tabular approach to reducing a Boolean function into a minimal SOP form. This approach involves two basic steps (12):

1. Generation of all the prime implicants
2. Selection of the prime implicants which cover the given function with a minimal cost.

The result of Step 1 is a formula consisting of all the prime implicants of the Boolean function; this formula is in Blake canonical form. A Blake canonical form can be obtained using one of a variety of techniques.

The second step involves finding an optimal selection of prime implicants that covers the function. Each prime implicant may cover one or more minterms. Thus, the generally accepted goal is to cover the function using the fewest prime implicants. To find an optimal cover, a classic set-covering approach is used.

The advantages of the Quine-McCluskey technique are numerous. They include the following (73):

- It is suitable for machine computation
- It can be applied to problems with a moderate number of variables
- It requires no human cognitive skills as do map-based or axiom-based approaches
- It is guaranteed to produce a simplified standard-form expression for a function.

Despite its advantages, the Quine-McCluskey technique does have its drawbacks. In its most basic form, it only applies to the simplification of functions with one output variable. It also has a complexity which belongs to the class of NP-complete problems (62). "Since the number of elements in the covering problem may be proportional to the exponential of the number of input variables of the logic function, the use of these techniques is totally impractical even for medium sized problems (10-15 variables) (12)."

3.2.4 Programmable Logic Array Minimization

3.2.4.1 Background. "In the late 1970s, the introduction of LSI and later VLSI made regular structures such as programmable logic arrays (PLAs) desirable for the implementation of logic functions because of the reduction in design time they offered (12)." As a result, the pursuit of efficient PLA optimization algorithms dominated optimization research. A PLA is conceptually a two-level AND-OR circuit. The product terms are produced in the AND array and the SOP form is generated in the OR array as shown in Figure 3.3 (62). Because of this, most of the classical two-level approaches such as the Quine-McCluskey method will work for PLA optimization also. However, the limitations of these approaches spawned numerous efforts to develop systems that could handle the large number of inputs and outputs that are characteristic of most VLSI circuits. The

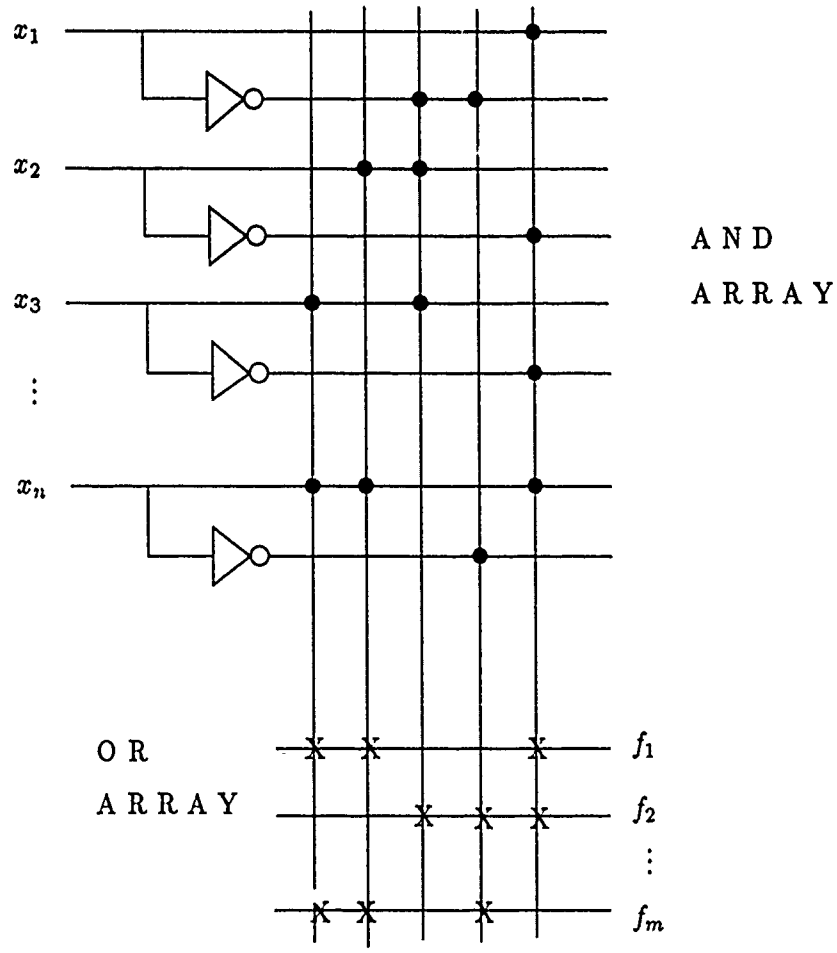


Figure 3.3. A Standard PLA Implementation

focus of most of these efforts involved the development of *heuristic* algorithms which do not guarantee a minimal solution but instead yield near-optimal solutions (62).

3.2.4.2 Early Methods. The goal of two-level (PLA) minimization is simply to reduce the number of product terms. Since each product term is implemented by a row in the PLA, a reduction in the number of product terms reduces the number of PLA rows required and in turn reduces the physical area of the PLA (12). Numerous PLA optimization techniques have been developed. One such technique involves generating all of the prime implicants and then using heuristic algorithms to select the best cover. This approach suffers from the possibility that the number of prime implicants may be extremely large (12). A second, and more popular approach, is the simultaneous identification and extraction of prime implicants.

Two popular identification/extraction methods were developed by Rhyne (87) and by Arevalo (1). Both methods take the following approach:

1. Select a base minterm from the on-set[1] of the logic function to be minimized.
2. Expand the term until it is prime.
3. Remove all minterms that are covered by this prime.
4. Repeat steps 1 to 3 until all the minterms of the care-set are covered.

In Rhyne's method, all prime implicants generated from the selected minterm are generated. If the prime implicant must be in any prime cover of the care-set of the logic function, or if it satisfies some predetermined heuristic criterion, then it is selected (12). The drawback to this method is that the number of prime implicants generated can be excessive. Arevalo's method is similar to Rhyne's except that it only generates a subset of prime implicants covering the base minterms. While it is a much faster method, its solutions are seldom as good (12). These methods and others provided some improvements over the Quine-McCluskey technique for selected sets of problems; however, they were still a long way from an ideal algorithm. They did begin to show how heuristics could be used to optimize circuits in a more efficient fashion.

[1] See section 2.2.4.1

3.2.4.3 Heuristic-Based Minimization Algorithms. There are a variety of PLA-based minimization algorithms that have been developed over the years. The following systems are a few of the most prominent in current literature.

MINI. MINI (57) is a heuristic-based logic minimization technique initially developed by IBM back in the middle 1970s. It was one of the first and most successful systems built around a heuristic approach to the problem. It was designed with the intent of dealing with practical problems of 20 to 30 input variables, which could not be handled using a classical approach. Hong describes the differences between MINI and classical approaches as follows (57):

- The cost function is simplified by assigning an equal weight to every implicant.
- The final solution is obtained from an initial solution by iterative improvement rather than by generating and covering prime implicants.

MINI attempts to eliminate the problems associated with a local minima by limiting the cost function to the number of implicants in the solution. Since only the number of implicants is important, their form can be altered as long as the coverage of the minterms remains proper. It modifies the implicants using formalized heuristic techniques similar to those used in Karnaugh map reduction. The process starts with an initial solution and then proceeds to make iterative improvements on it. There are three basic modifications that are performed on the implicants of the function (57):

1. Each implicant is reduced to the smallest possible size while still maintaining the proper coverage of minterms.
2. The implicants are examined in pairs to see if they can be reshaped by reducing one and enlarging the other by the same set of minterms.
3. Each implicant is enlarged to its maximal size and any other implicants that are covered are removed.

The order in which each of the steps above is applied is crucial to the success of the procedure. However, in general the three steps are iterated until no further reduction is

obtained in the size of the solution. The overall goal of the algorithm is to further optimize the function by considering groupings of minterms that may or may not be obvious from the statement of the problem. Over the years it proved to be quite successful in finding near-optimal representations of medium-complexity digital circuits

PRESTO. PRESTO, another heuristic minimization program, was introduced by D. Brown in 1981 (21). Like MINI, its operation focuses on expanding each implicant of the logic function and removing other implicants covered by the expanded one. However, the two methods differ in how the implicants are expanded (12). MINI expands each implicant to its maximum size in both the input and output part. PRESTO expands the input part of each implicant to its maximal size, but reduces the output parts of the implicants maximally by removing covered implicants from as many output spaces as possible. In other words, the covering step is implicit in the output reduction procedure; implicants are expanded and then all those implicants that are covered by this expansion are removed. This eliminates the problem of having to generate and store all of the minterms. PRESTO's output reduction step produces a cover that is irredundant (i.e., a cover such that no proper subset is also a cover of the given logic function) but not necessarily prime. Another way the two differ is the way the expansion process is carried out (12). MINI generates the complement of the logic function (i.e., produces the off-set) to see if the expansion of an implicant changes the coverage of the function. PRESTO avoids the computational costs associated with computing the complement, but its input expansion process requires a check to see if minterms covered by the expanded implicant are covered by some other implicant. Depending on the particular application, this in general costs more computation time.

ESPRESSO-I. A detailed record of the creation of the ESPRESSO algorithms and subsequent developments is provided in a book by its designers titled *Logic Minimization Algorithms for VLSI Synthesis* (12). In an effort to improve on the techniques developed in MINI and PRESTO, a new program, ESPRESSO-I, was developed during the summer of 1981. Experimentation with this prototype system enabled the designers to draw some important conclusions. They found that MINI's methods, which involved computing the complement of the logic functions, were superior because the initial cost

was typically offset by a more efficient expansion procedure. This method gained further acceptance with the discovery of more efficient methods to complement a logic function. They also found that the technique used by MINI of iteratively expanding and removing implicants until no further reduction in the number of product terms is obtained, generally produced better results, justifying any increases in computation time.

ESPRESSO-II. All of the accomplishments realized with ESPRESSO-I were incorporated into a new system during the summer of 1982. The result was a system called ESPRESSO-II which even today is one of the most popular PLA minimization techniques. The initial goals in the design were to:

1. Solve practical logic minimization problems with the use of limited computing resources.
2. Obtain a result that was close to the global optimum.

The details of how this system works can be found in the designer's book (12). In general it follows the same expansion/reduction techniques developed in MINI but modified to incorporate the improvements mentioned above. The ESPRESSO-II algorithm is designed to handle multiple-output as well as single-output circuits. The authors claim the system will produce a near-optimal or in some cases optimal PLA implementation of the circuit. In tests, ESPRESSO-II compared favorably with the Quine-McCluskey technique in its ability to minimize a circuit, but arrived at a solution on the average 10 to 100 times faster when working with large circuits (17).

3.2.4.4 An AI Approach To PLA Optimization. Within the last few years, a variety of new approaches to PLA minimization have emerged. One of these involves an AI approach to PLA optimization (62). This novel approach formulates the problem as a *state space search.* Several heuristic evaluation functions are used to guide the search which involves the construction of a binary decision tree. This technique has shown some promise in reducing large circuits with as many as 40 variables (62).

3.2.4.5 The Exact Two-Level Minimization of Logic Functions. While near-optimal solutions are adequate for most applications, there are situations when we would like to have an efficient algorithm that produces an exact global optimum of a logic circuit. The results from an exact optimizer could also be used to evaluate the performance of near-optimal, heuristic systems. It is always valuable to know exactly how close we are to a global optimum. Such exact optimizers do exist and McBOOLE (27) and EXPRESSO-EXACT are two examples. Most exact minimizers try to improve on the basic techniques developed by Quine and McCluskey. McBOOLE makes use of efficient graph and partitioning techniques to enable it to handle functions that are much larger than before. Others have proposed heuristics that greatly reduce the search space of the branch-and-bound algorithm (99).

3.2.5 Summary of Two-Level Optimization Techniques. Current optimization systems can consistently and efficiently find optimal or near-optimal two-level circuit representations for a variety of logic functions. These functions can vary from simple functions with single outputs to functions with hundreds of inputs and outputs. There are even systems available that are capable of operating efficiently on a personal computer. A considerable amount of research is currently underway which promises to deliver even more efficient two-level optimization systems in the near future. In summary, today two-level logic-function optimization is considered a very well developed science.

3.3 Multi-Level Optimization Techniques

While the area of two-level optimization is becoming well understood, multi-level optimization is still in its infancy. A multi-level implementation of a Boolean function (see Figure 3.4) allows the unrestricted use of intermediate signals. The option to reuse intermediate signals can lead to an unlimited number of ways to represent multi-level circuits. While this significantly enhances the degrees of freedom offered to a designer, it also is accompanied by a significant increase in complexity. It is this complexity that has hampered the creation of efficient automated multi-level optimization systems. Karen Bartlett describes multi-level optimization as "a science that suffers from the very thing

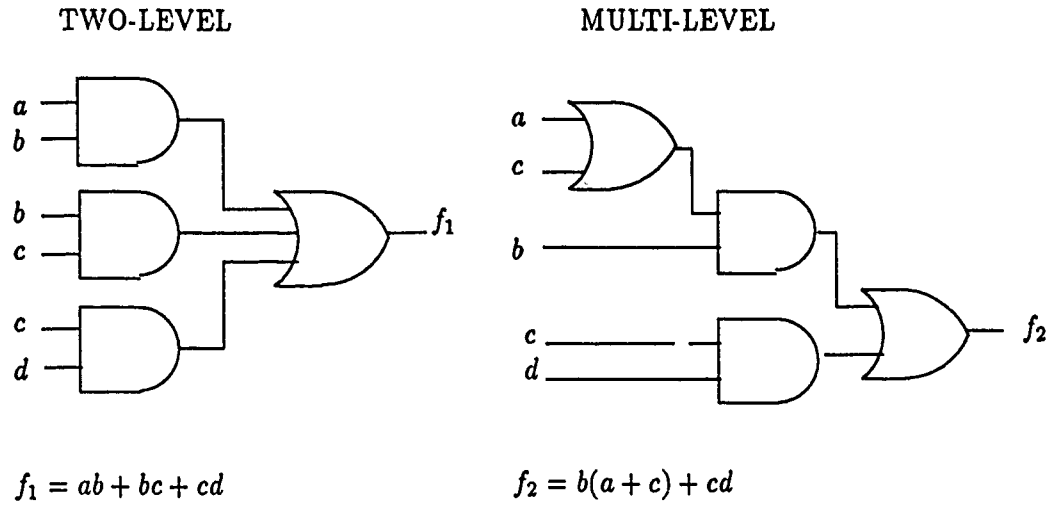


Figure 3.4. Two-Level versus Multi-Level Logic

that makes it attractive, its flexibility (3)." Even today, manual techniques for optimizing multi-level circuits frequently outperform automated systems, but fortunately the gap is closing.

There are many reasons why multi-level implementations of logic circuits are preferred over the simpler two-level variety (66):

1. Functions that are implemented in multi-level logic typically occupy less area than an equivalent two-level representation.
2. Multi-level implementations often provide a more natural structure for many digital systems. Logic has often been perceived as falling into two categories, control logic and data-flow logic. While control logic is best suited for PLA implementations, data flow logic fits more naturally into a multi-level format.
3. When a two-level design is implemented in standard gate technology, it often exceeds the fan-in limitations of standard gates. Multi-level implementations generally reduce the problem associated with fan-in and fan-out. Many multi-level optimization systems provide the designer with the necessary tools to carefully control it.

Through the years a variety of approaches have been taken to optimize multi-level logic circuits. A few of these systems have proven to be commercially viable and competitive with manual optimization techniques. Most of the approaches to multi-level optimization fall into two major categories. They are *local optimization* and *global optimization* (22).

3.3.1 Local Optimization

3.3.1.1 The Use of Artificial Intelligence. Before we investigate local optimization techniques in detail, it is imperative that we understand the motivation behind the recent interest. This motivation was sparked, in part, by notable advancements in the field of artificial intelligence. William Birmingham points out in his article that:

> Artificial intelligence (AI) is being applied to increasingly more broad and difficult problems. Recently, interest has grown significantly in AI applications of digital system design. Digital system design is an area which has eluded attempts of significant automation due to its very complex nature. AI in general, and rule-based systems in particular, provide an attractive means of managing complexity by applying problem-solving techniques which utilize heuristic, rather than algorithmic approaches. (8)

Local optimization techniques are appealing because in many respects, they model the process a human designer would take in optimizing a circuit. This possibility is one of the key features of rule-based systems and AI in general. A rule-based system tries to capture the knowledge of an expert in a non-algorithmic fashion and use this knowledge to guide a sequence of actions. It incorporates general, heuristic information into the system as rules. Recent advances in rule-based systems (often called expert systems) have made the application of AI principles to computer aided design and optimization much more practical (96). This has spawned numerous research efforts in the application of AI techniques to the local optimization of multilevel logic circuits.

3.3.1.2 Local Optimization Principles. In local optimization, an initial circuit is synthesized from a system of equations that specify the circuit. A rule-based system then scans repeatedly, applying local transformation rules to modify the circuit. These rules seek to reduce the circuit area or propagation-delay.

It is important to understand exactly when to apply local optimization techniques. The word *optimization*, in our context, implies that there exists an initial circuit to optimize. Thus, a local optimization system generally accomplishes its task in the following three distinct steps (31):

1. Minimize the Boolean equations specifying the circuit,
2. Synthesize an initial circuit,
3. Optimize the circuit for a given technology.

Only the third step actually involves the application of local transformations. In Step 1 the system of Boolean equations that describe the circuit is reduced using mathematical methods, taking full advantage of any "don't-care" conditions. In Step 2 a variety of techniques are utilized to synthesize the initial circuit. They often use a limited set of gate types such as NAND/NOR gates and multiplexers. Some systems incorporate multi-level synthesis techniques such as factorization to take advantage of any common intermediate terms. Finally, in Step 3, local optimization principles are applied.

The key to the success of local optimization is the use of an efficient rule-based system. Each rule encompasses replacing an existing configuration of one or more circuit elements with an equivalent but more desirable configuration as shown in Figure 3.5 (31). Since this replacement involves only a few circuit elements, it is considered local optimization versus a global approach that involves all of the circuit logic (44). The rules generally follow those that a designer uses when manually optimizing a logic circuit. New rules can be incorporated into a local optimization system by simply adding them to the library. "By using libraries of rules geared towards specific technologies, one can drive the optimization towards a particular technology or convert from one technology to another"(44).

During the optimization process, the rules are applied in a well ordered fashion. The order in which they are applied can have a dramatic affect on the final result (44). *General rules* that reduce both area and speed are applied first. They are prioritized by their relative importance. These rules fire repeatedly until no more general rules can be applied. At this point some of the more advanced systems like SOCRATES will perform a timing

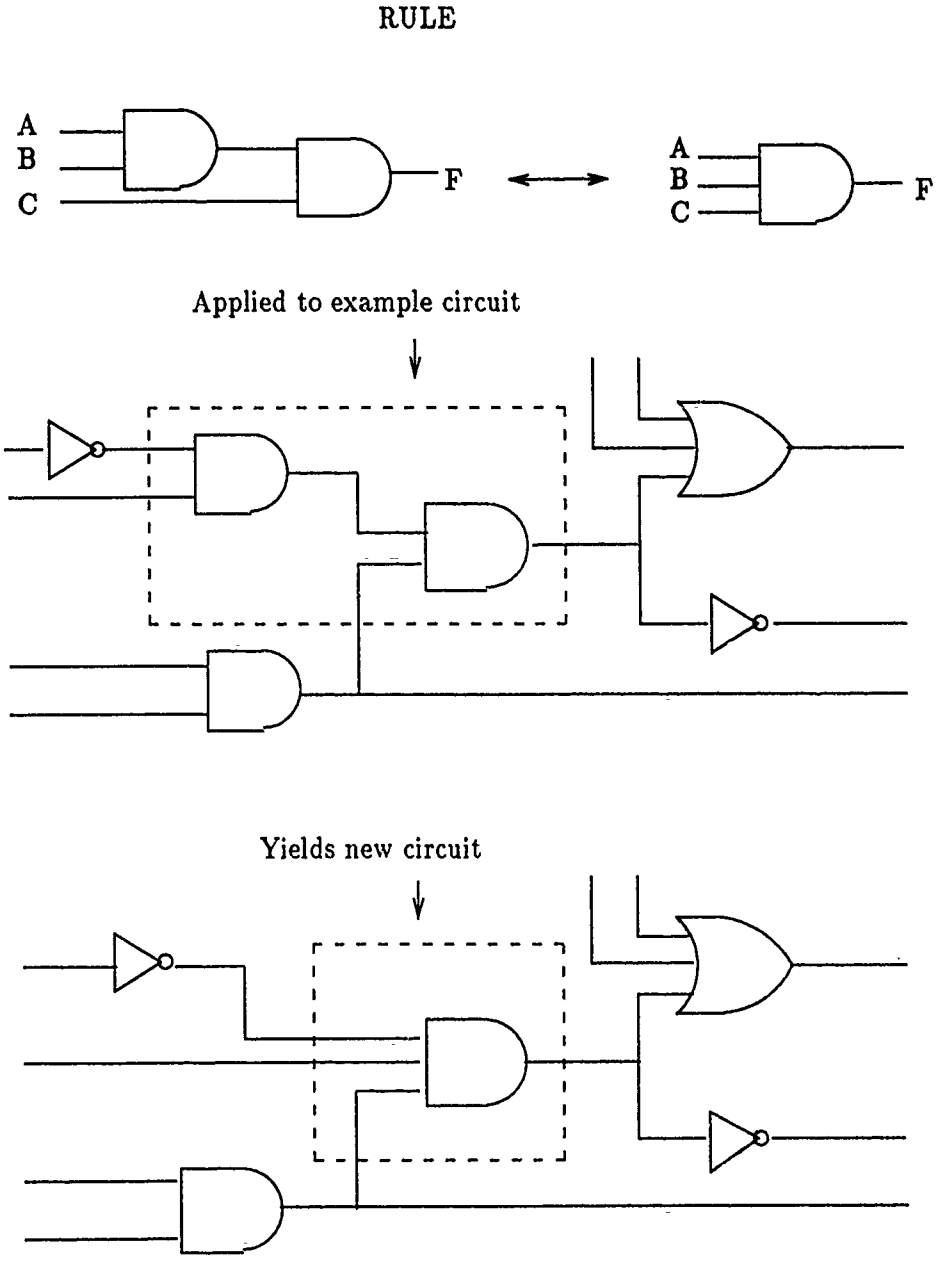


Figure 3.5. A Local Transformation Rule Example

analysis of the critical path. If the critical path delay time does not fall within established guidelines or predefined limits, then some *time-saving rules* will be applied to gates in the critical path. When the desired time has been achieved, *space-saving rules* will be applied to gates not on the critical path.

The final result of these local optimization systems is often a significantly improved design. Not only are the designs faster and smaller, but the time taken to synthesize them with a local optimization system is significantly reduced. Some systems have proven themselves commercially viable and able to consistently outperform human designers. A few of the more popular systems in use today are described in the following section.

3.3.1.3 Local Optimization Systems. The history of the development of local optimization systems is a rather recent one. It was aided by recent interest in applying rule-based systems to practical problems that had been difficult to solve using other approaches. IBM was one of the first companies to show an interest with their development of the Logic Synthesis System (LSS) in the early 1980s This was later followed by LORES, DAS/Logic, SOCRATES, and others.

LSS. LSS (30) was developed to transform a high-level circuit specification into a production-quality implementation through a series of local transformations. The primary goal was a system that produces feasible, not necessarily optimal, circuits that satisfy a series of constraints. They should meet the requirements of the target technology, take full advantage of the features of that technology, and produce logic with acceptable gate counts and path lengths.

The system proceeds in a step-by-step fashion. It begins by translating the system specification from a register-transfer language to a network consisting of ANDs, ORs, NOTs, decoders, adders, etc. Next, through a series of local transformations at several levels, the network is replaced by more primitive implementations such as NANDs and NORs and eventually, technology specific devices. The entire process evolves from initial stages which are relatively independent of the target technology, to the final stages that are technology-specific. The use of multiple stages in the synthesis and optimization of logic enables the system to take advantage of numerous simplification techniques that are

applicable at each level.

The LSS system has shown a considerable amount of success particularly in its use on IBM projects. "On one project in particular, it was used on 90 percent of the more than 100 chip designs. This cut their initial design time in half (30)." Despite the success, LSS also had its drawbacks. It was limited in its ability to handle timing and gate-count problems. It did not take full advantage of all the information that was available such as "don't-care" conditions. It also was often out-performed by systems that approached the problem from a more global point of view. Despite this it provided the framework from which future systems would follow.

LORES LORES (LOgic RE-Organization System) is a system which automatically partitions and restructures a logic circuit consisting of standard SSIs and MSIs so that the gate types and numbers of input/output terminals are compatible with the requirements of LSI (80). The advantages of LSI are numerous; they include smaller size, better performance, reduced power consumption, and often a reduction in overall cost. These advantages don't come free and the price we often have to pay is increased development time, greater complexity and a more difficult testing and verification process. LORES concentrates its efforts on achieving some of the benefits of LSI by overcoming some of the difficulties.

LORES is an automatic logic optimization system whose primary functions are described below (80):

- Extract one of the partitioned sections from the original design.
- Eliminate any unused or redundant logic.
- Convert or restructure the logic to those logic elements allowed in LSI.
- Partition the logic circuit so that the number of gates and input/output terminals are within the specified limits.

One of the major drawbacks to LORES is its inability to handle asynchronous circuits. Thus, for circuits containing an asynchronous component, a certain amount of simulation and manual modification may be necessary (80). On the other hand, "LORES directly

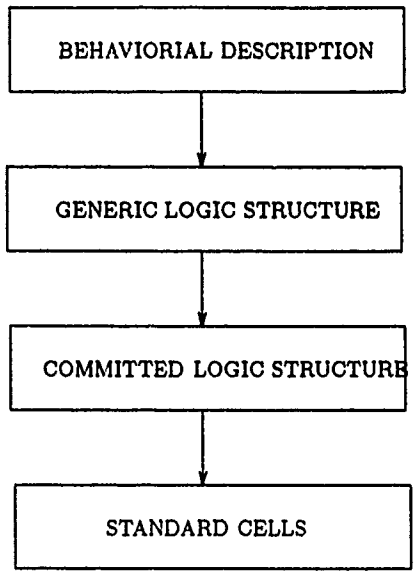


Figure 3.6. DAS/Logic Design Level Hierarchy

contributed to a reduction in development time and manpower by automating the circuit transformation process, and by reducing the necessity for design verification (41)."

A recent version of LORES has surfaced called LORES/EX (59). It is designed to transform an existing logic circuit from one technology to another. Not only is it flexible to changes in technology, but it also has a number of other special features. It introduces standardization rules which seek to reduce and simplify the size of the rule-base. It has incorporated features that employ conflict resolution to improve the overall quality of the circuits. It also has the ability to partition large circuits into smaller ones allowing LORES to handle circuits with up to 10,000 gates in a practical amount of time.

DAS/Logic. DAS/Logic (Design Assistant Series) is a tool being developed at Carnegie Group Inc. to aid in the design of ICs. It is described by Birmingham (8) as a rule-based system written in OPS5 which transforms a behavioral description of a system into a circuit schematic. The input is a high-level description of the desired behavior of the circuit and the output is a set of standard cells and an interconnection list. The sequence of design phases is illustrated in Figure 3.6 (8).

You will probably notice that the phases described in Figure 3.6 are very similar to those used in the LSS. Like LSS, DAS/Logic proceeds from technology-independent phases

into technology-dependent ones. There are a variety of steps that are performed at each design level. These steps include (8):

1. Generate a structure for the current design level.
2. Look for optimization opportunities or constraint violations.
3. Correct violations or apply optimizing transforms.
4. Generate constraints/opportunities for the next level structure.

DAS/Logic design follows a top-down approach combined with some bottom-up strategies. The top-down approach enables the system to work from a more global standpoint; however, knowing the features of the implementation technology makes bottom-up design strategies desirable also. For example, one wouldn't want a system that creates four-input OR gates if they do not exist in the target technology. DAS/Logic designers realized early that an ideal logic optimization system must include a combination of top-down and bottom-up strategies.

The system, at last report (8), was still in the development stages but showing considerable progress. The entire development effort stressed the use of "intelligent" AI techniques to solve difficult problems. "Through the use of domain knowledge extracted from designers, DAS/Logic is able to intelligently apply its optimization and constructive expertise, thereby drastically reducing the amount of search necessary to achieve high quality designs (8)."

SOCRATES. SOCRATES (Synthesis and Optimization of Combinatorics using a Rule-based And Technology-independent Expert System) is currently one of the most successful logic optimization systems. It was developed with the intent of incorporating the most attractive features of current optimization systems. The resulting system combines the use of two-level and multi-level optimization techniques with local, rule-based optimization algorithms. The resulting system can be generalized as proceeding in the following steps (50):

1. The equations or net-lists used to specify the desired behavior of the system are transformed into a format compatible with the PLA minimizer EXPRESSO.

2. EXPRESSO reduces each function into its minimal SOP form.

3. *Weak division*[2] is used to transform the two-level implementation into a reduced, but equivalent, multi-level circuit.

4. A rule-based expert system is used to apply local transformations to the circuit to optimize it for a particular technology.

SOCRATES also has the capability to input existing circuit designs in a predefined format to undergo further optimization or to convert them to a new technology. It is probably one of the most proven systems in its ability to handle area and timing constraints. A user can specify an optimal compromise between increased speed and reduced size. The circuit can then be optimized to achieve the desired goals. The rule base is built so that the user can easily enter additional rules which are automatically verified and classified in the knowledge base (31).

SOCRATES takes several measures to insure an efficient operation. It uses *metarules* to ensure that the search space is kept as small as possible. This, coupled with the fact that it is written in the C language rather than a typical expert system language, ensures comparatively short run-times. Because of these improvements, "combinational circuits that would have taken several days to synthesize and optimize can now be generated in minutes (31)." The bottom line is that for larger circuits (75 to 100 gates), SOCRATES achieved area-reductions ranging from 25 percent to 60 percent in most cases; these results compare extremely well to those achieved manually by experts (44).

3.3.1.4 Summary of Local Optimization Systems. While local optimization systems are quite successful and are being applied to increasingly difficult problems, they are not, by themselves, the solution to all optimization challenges. They have proven to be extremely effective when it comes to transforming a given circuit design into a new technology, but are still very weak when it comes to taking full advantage of all the information contained in a specification, including "don't care" conditions. SOCRATES has taken steps to make use of such information by incorporating simple global optimization

[2]Weak division is discussed in detail in section 3.3.2.4

algorithms into its system. Many local optimization systems are limited by the efficiency of their rule-based systems. While the speed of these systems is improving, efficiency still provides a major obstacle. The future of local redesign systems is a bright one and there are likely to be more hybrid systems like SOCRATES.

3.3.2 Global Optimization. Unlike local optimization, which works only on small portions of a circuit at any given time, global optimization considers the entire circuit's logic functions. The benefits are quite obvious. By considering all of the circuit's logic functions at once we can:

- Effectively eliminate any redundancies in the circuit.
- Take maximum advantage of any "don't care" conditions that exist.
- Facilitate the use of intermediate signals to create a more efficient multi-level structure.

Global optimization, as its name implies, is a more general approach to the multi-level optimization problem. This approach typically results in a more flexible system, capable of optimizing any circuit, independent of the ultimate target technology. While these features appear extremely attractive, they do not come without their drawbacks. Because global optimization works with the logic functions that describe a circuit rather than the circuit's physical representation, it is considerably more complex.

The current state-of-the-art in multi-level logic optimization, particularly as it relates to global approaches, is addressed in *Multi-Level Logic Synthesis* (17), an article by R.K Brayton, G.D. Hachtel and A.L. Sangiovanni-Vincentelli. Much of the following material in this chapter is a summary of that article; for further detail and understanding, one should take the time to look through the article in its entirety.

A variety of techniques have been developed to perform global, multi-level optimization. Most of these fall into two basic categories; *algebraic methods* and *Boolean methods* (17). Before we begin a detailed discussion of these two methods we need to address a more basic problem, namely, how to represent the system that we wish to optimize.

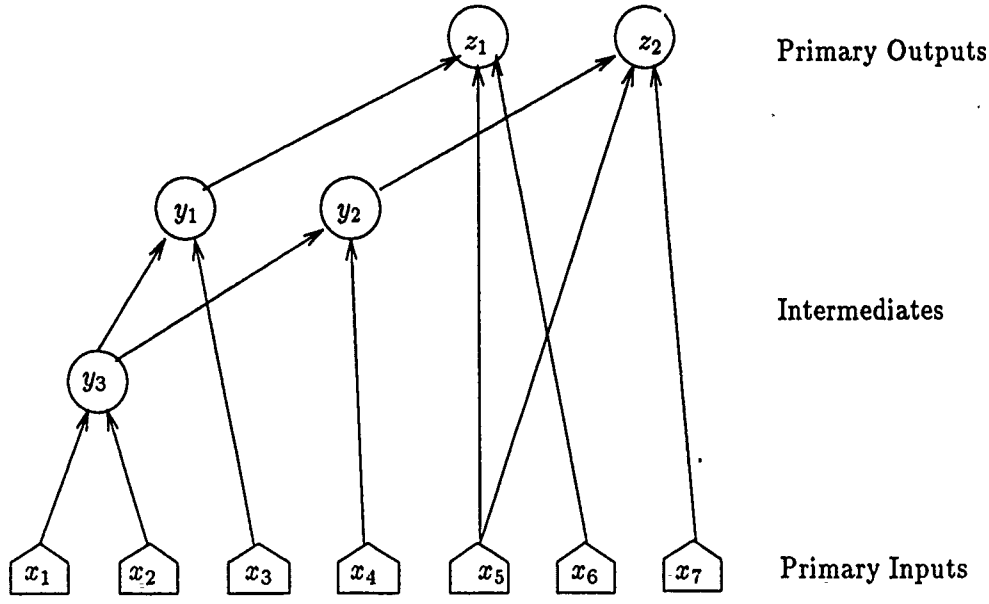


Figure 3.7. Multi-Level Boolean Network

3.3.2.1 Network Representation. Typically we think of a circuit specification as a system of boolean equations (see section 2.3). This system of equations can be described in terms of a *Boolean network.* A Boolean network is a representation of a combinational logic circuit that describes the desired behavioral characteristics of a given circuit design (see figure 3.7). According to Brayton:

> A Boolean network is a directed acyclic graph. Associated with each node of the graph is a variable, y_i, and a representation of a logic function, f_i. A directed arc from node i to node j is in the graph if node j uses the variable y_i explicitly in the representation f_j. (17)

In more general terms a network representation can be visualized as a transformation process that accepts primary input variables, acts on these variables at a variety of levels, and then produces primary output variables that are a function of the inputs. We can think of this representation in the same sense that we think of a PLA or SOP as a representation of a circuit. Our goal is to seek a representation that facilitates the use of global techniques to optimize multi-level logic circuits. Any network representation can be used as long as the values of all outputs, corresponding to inputs not in the don't care set, are equivalent.

3.3.2.2 Node Representation. Brayton points out that every node in a Boolean network has associated with it a representation of a logic function. The way this function is represented is extremely important. Although any valid representation is allowed, some representations are preferred because they are

- more efficient in memory
- more indicative of the complexity of the final implementation
- more efficient to manipulate

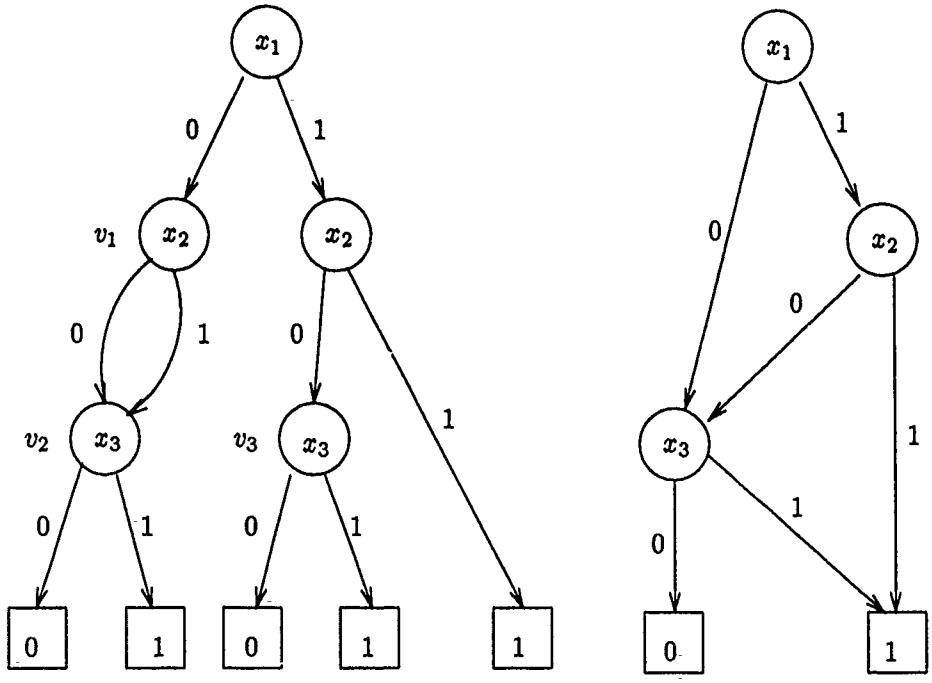
It is interesting to note that this issue does not appear in two-level theory since the initial representation and the final implementation are both in a SOP form. However, for multi-level implementations there are a number of choices available.

Sum-of-Products. Representation of a node as a SOP is probably the most common form. Manipulation techniques based on two-level (SOP) forms are well established as a result of efforts in PLA optimization. Most logic designers are also more comfortable working with standard AND, OR and NOT gates. "Even though we may wish to represent logic in some other way, present techniques generally require conversion to a SOP form, manipulation with developed algorithms, and conversion back (17)."

Factored Forms. Factored forms are a more natural representation for multi-level logic. An example of a factored form is

$$(ab + b'c)(c + d'(e + ac')) + (d + e)(fg) .$$

"A factored form is isomorphic to a tree structure, where each internal node is an AND or OR operator and each leaf is a literal. This leads to a simple and efficient multilevel implementation of the function of the node (17)." In addition, by simply counting the number of literals in a factored form, one can get an accurate idea of the complexity of that function. This provides an important heuristic when considering whether or not a particular transformation improves are worsens the overall complexity. Another argument in favor of factored forms is that they implicitly represent both the function and its complement. In other words a complemented factored form can be obtained, using DeMorgan's law, by



DECISION GRAPH BDD

Figure 3.8. Decision Graphs and BDDs

converting the ANDs to ORs, and vice versa, and negating the literals. This produces the complement of the factored form which has the same literal-count. This is in sharp contrast to the SOP form where the number of products in a function can be exponentially larger than the number of products in that function's complement (17). One problem with a factored form is that it is difficult, if not impossible, to know if the factored form is optimal. Another difficulty with this approach is that effective methods to manipulate factored forms do not exist. However, to fill this void a number of research projects are underway (17).

Binary Decision Diagrams. Binary decision diagrams (BDDs) are a relatively new and promising development in the logic optimization field. They basically involve a graph-based approach to representing logic circuits. A BDD is best described with the use of an example as shown in Figure 3.8 (72).

A BDD represents a logic function as an acyclic graph. The root node, labeled x_1, represents the entire function and the leaf nodes represent the functions 0 and 1 respectively. Each node has two children which represent the new function created when the variable in the parent function (node) is set to 0 or 1. The output value of any function can be found by tracing the assignment of values to its inputs through the graph. It is interesting to note in Figure 3.8 that when $x_1 = 0$ it doesn't matter what x_2 is. This is shown explicitly in the Decision Graph but is implicit in the BDD. For a given ordering of the variables, a BDD represents a canonical form. It is forced to be reduced in the sense that if two nodes represent the same function, then they must be the same node. Because of this characteristic, BDDs are often used to see if two multi-level networks are equivalent. If they are, both would have identical BDDs. In addition, because of its canonical form it occupies less space and is easier to manipulate using graphical techniques. It has been found that the ordering of the variables can have a drastic affect on the size and the shape of the BDD. As a result, numerous heuristic techniques have been developed to help find the optimal ordering of variables. It was shown by Bryant (20) that most logic operations on BDDs can be done in linear or log linear time in terms of the number of nodes in the BDD. Brayton describes a number of extensions to BDDs to handle incompletely specified logic functions, multivalued variables, multivalued outputs, and even if-then-else directed acyclic graphs (17). Overall, BDDs are providing an effective new representation for Boolean functions. As algorithms are developed to take full advantage of this representation, significant improvements in current optimization systems are likely to be realized.

3.3.2.3 Basic Operations. While the methods may vary, the goal of multi-level global optimization systems remains the same: to transform an initial circuit representation into an optimal one. There are a number of basic operations used to accomplish this. It is important to note that these operations could apply equally to algebraic or Boolean methods. Listed below are five of the most common:

Decomposition. The process of decomposition transforms a single Boolean function into a collection of new Boolean functions. The following example (17) shows the

process of translating

$$F = abc + abd + a'c'd' + b'c'd'$$

to

$$\begin{aligned} F &= XY + X'Y' \\ X &= ab \\ Y &= c + d\,. \end{aligned}$$

We can see from the results that the two-level representation was replaced with a multi-level representation. Note that the fan-in of F was reduced from four terms to two.

Extraction. The extraction operation is related to decomposition except that it applies to many functions. This transformation process seeks to identify and utilize common subexpressions among the functions. It creates intermediate functions with which the original functions are expressed. In terms of our Boolean network, we can look at this operation as one which creates new nodes to simplify the overall function representation. As an example (17) if we apply extraction to the following three equations

$$\begin{aligned} F &= (a+b)cd + e \\ G &= (a+b)e' \\ H &= cde \end{aligned}$$

one result yields

$$\begin{aligned} F &= XY + e \\ G &= Xe' \\ H &= Ye \\ X &= a + b \\ Y &= cd\,, \end{aligned}$$

where X and Y are newly created intermediate nodes.

Factoring. Factoring is the process of creating a factored form from a Boolean function expressed in a SOP form. For example (17),

$$F = ac + ad + bc + bd + e$$

can be factored to

$$F = (a + b)(c + d) + e\,.$$

The goal of many optimization techniques is to find a factored form with the minimum number of literals.

Substitution. This operation, often called *resubstituition*, is the process of expressing one function in terms of its inputs and another function. In the example shown below (17), the substitution of

$$G = a + b$$

into

$$F = a + bc$$

yields

$$F = G(a + c)\,.$$

In terms of our Boolean network, this operation creates an arc from the node of the function being substituted (G) to the node of the function it is being substituted into (F).

Collapsing. This operation, also called *elimination* or *flattening*, is essentially the inverse of substitution. It eliminates subfunctions by placing them back into the original expression. For example (17), if

$$\begin{aligned} F &= Ga + G'b \\ G &= c + d\,, \end{aligned}$$

then collapsing G into F results in

$$F = ac + ad + bc'd'$$
$$G = c + d .$$

If node G isn't an output, and no other function in the Boolean network depends on G, then G can be eliminated resulting in a network with one less node.

To implement all of the operations described above, techniques which are very similar to division and multiplication are used. "In fact, *division* plays a key role in multi-level logic optimization (17)." The concepts of division will be discussed in some detail as we take a closer look at the distinction between algebraic and Boolean methods.

3.3.2.4 Algebraic Methods. The choice between using algebraic methods and Boolean methods depends largely on the desired optimality of the result and the limitations on the time needed to produce it. Boolean methods generally produce a result that is closer to the global optimum but at the expense of increased computational intensity. Algebraic methods have proven to be much faster and can produce adequate, though not necessarily optimal, results. The challenge facing most designers is to find the most effective way to apply the various methods available to produce a quality result in a reasonable amount of time. The most commonly used approach to multi-level synthesis involves the following steps (17):

1. minimize each logic function to obtain an algebraic expression,
2. perform algebraic operations, including decomposition, extraction, factorization, resubstitution, and elimination, on these expressions,
3. optionally iterate steps 1 and 2. In some cases Boolean operations are used sparingly to improve the results but without significantly affecting the efficiency of the system.

Algebraic Division. To understand the concept of algebraic division we need to first introduce some basic definitions and terminology (18). If we express a function in SOP form as $f = f_1 + f_2 + \cdots + f_n$ then each term f_i is referred to as a *cube*. The *algebraic*

product of two expressions f and g is only defined if f and g depend on a disjoint set of variables. The algebraic product is the sum of all possible cross-terms $f_i g_j$. Since f and g have disjoint variable sets, no zero products can occur (a variable is never multiplied by its complement).

The *algebraic quotient* f/g is required to depend on variables other than those on which g depends. It is defined to be the largest expression such that

$$f = (f/g)g + r \tag{3.6}$$

where r is the remainder. Here the product of (f/g) and g must be algebraic, and the right and left sides of the equation are required to agree as expressions, not just logical functions. As an example (18), if

$$f = AB + AC + AD + BC + BD$$
$$g = A + B ,$$

then

$$f/g = C + D$$

since

$$f = (A + B)(C + D) + AB .$$

According to Brayton, "By using sorting techniques, the computation of the quotient f/g can be carried out very efficiently. In fact the division requires only O(n log n) steps, where n is the total number of cubes in f and g (18)."

Weak Division. Weak division is a specific example of algebraic division. It uses algebraic techniques to find divisors that are common to two or more functions. The term "weak" refers to its comparison with the more powerful technique of Boolean division (17). The following descriptions and examples of weak division are, for the most part, a summary from the article "Library Specific Optimization of Multilevel Combinational Logic" by Karen Bartlett and Gary Hachtel (2).

Finding common subexpressions among functions enables their logic to be shared and also leads to the formation of a multi-level structure that is often more efficient and easier to implement in a desired technology. Weak division is an iterative process which typically proceeds in a method similar to that shown below:

```
procedure Weak_division

begin
   /* Decomposition */
   while (common subexpressions exist)
      - Generate candidate subexpressions for current functions
      - Determine eligible subexpressions
      - Select "best" disjoint subexpressions
      - Associate new intermediate variables with subexpressions
        and substitute
   end while

   - Collapse subexpressions referenced by only one function

   /* Factorization */
   for each function
      - repeat above loop for single function
   end for

end Weak_division
```

Through a careful analysis of this algorithm, we discover that it is an iterative process consisting of four distinct steps: the generation of candidate subexpressions, the pruning of eligible candidates, the selection of the best disjoint subexpressions and the substitution of these subexpressions into the function which they divide. When we substitute a subexpression back into a function, there may exist new divisors which are expressed in terms of this new intermediate variable. Because of this, the iterative process continues until there are no more divisors of sufficient merit to warrant substitution. Weak division is broken down into two distinct phases, the decomposition of the system of functions and the factorization of each individual function. As a simple example, the result of applying

the weak division algorithm to the functions

$$f1 = aef + bef + cef$$
$$f2 = aeg + beg + deg$$

yields

$$f1 = Aef$$
$$f2 = Beg$$
$$A = C + c$$
$$B = C + d$$
$$C = a + b.$$

We can see that the outputs $f1$ and $f2$ are expressed in terms of the inputs and the newly created intermediate variables.

Kernels. One of the goals of global optimization systems is to extract a manageable set of promising divisors which is rich enough to allow all the common subexpressions to be located. For this reason the concept of *kernels* was developed (18). The set of kernels of an expression f is defined as the set of all quotients f/c such that c is a cube and f/c is "cube-free". As an example ABC + ABD is not cube-free since it can be expressed as AB(C + D). However, C + D by itself is cube-free. We note that in general it would be fairly easy to find a cube that is a divisor but much harder to find a divisor that is cube-free. This is the motivation behind creating the concept of kernels. Many techniques have been developed to extract a set of kernels from a system of functions and then use these kernels to produce an optimal or near-optimal factored form for the Boolean network.

Algebraic Algorithms. "The operations of extraction, decomposition, factoring, and substitution can be carried out quite effectively in the algebraic domain using weak-division and kernels (17)." The concept behind substitution was described briefly earlier. *Algebraic substitution* involves dividing the function f_i at node i in the network by

a function f_j (or by f_j') at node j. If it is found that f_j is an algebraic divisor of f_i then f_i is transformed into

$$f_i = (f_i/f_j)f_j + r\,; \tag{3.7}$$

similarly for f_j'. Since ideally we attempt this for every pair (f_i,f_j) in the network there may be as many as $2n^2$ algebraic divisions where n is the number of nodes in the network. Fortunately there are a number of techniques available which can identify if a function is not a potential divisor (17) and thus the numerous possibilities can be reduced to a manageable set.

From our earlier definitions of factorization and decomposition we see that the basic operations involved are the identification of a divisor and division of a function by that divisor. Decomposition is basically identical to factoring except that divisors yield new nodes in the network. These operations can be accomplished in an algebraic domain by using algebraic division and in particular weak division. Since heuristics are generally used to improve the efficiency of these processes, they can not be guaranteed to find the optimal solution.

The extraction operation identifies common subexpressions and manipulates the Boolean network accordingly. Algebraic decomposition and substitution can be combined to provide an effective extraction algorithm.

Typical Synthesis System. The algebraic techniques we have described so far, in conjunction with a few boolean simplification techniques, can be used to create a complete logic synthesis procedure. A typical sequence of operations described by Brayton (17) is shown below:

1. Collapse incoming data. As usual, those intermediates whose values exceed a given critical amount are not pushed back. By setting this cutoff higher or lower, we can control the degree to which the original decomposition is preserved.
2. Perform Boolean simplification, using the implicit don't care sets.
3. Extract common subexpressions. Even subexpressions with fairly low value should be extracted, since they help to disclose other subexpressions.

4. Collapse again. Any intermediates created above which prove to be of little value are now removed.

5. Simplify again, using don't care sets.

6. Decompose the logic into functions simple enough to be implemented in single circuits in the target technology.

7. Collapse one final time. At this point, do not re-substitute any expression if its removal creates a function which can no longer be realized as a single circuit. At the same time, attempt to reduce the number of circuits and delay stages by trying to re-substitute all but the most valuable intermediates.

8. Design circuits for each function in the network.

3.3.2.5 Boolean Methods. While algebraic techniques offer the advantage of increased speed, it is often at the expense of the quality of the solution. When optimal or near-optimal results are a critical objective, *Boolean methods* are used, often in conjunction with algebraic techniques. Boolean methods treat the logic expression as a true logic function. This enables one to take maximum advantage of Boolean identities and any "don't care" conditions that may exist.

Boolean Division. Boolean division is much like algebraic division with one distinct exception. We remove the restriction that the two expressions f and g must depend on a disjoint set of variables. Thus it is possible for zero products to occur (a variable is multiplied by its complement). For example,

$$(a+b)(c+d) = ac + ad + bc + bd$$

is an algebraic product and both

$$\begin{aligned}(a+b)(a+c) &= a + ac + ab + bc \\ (a+b)(b'+c) &= ab' + ac + bc\end{aligned}$$

are Boolean products. Thus in reference to our formula for division (3.6), when the product $(f/g)g$ is an algebraic product, then algebraic division takes place. Otherwise $(f/g)g$ is a Boolean product and Boolean division takes place.

One Boolean division technique that applies to problems in which the divisor is a term was introduced by M.J. Ghazala (45) and later expanded on by F.M. Brown (22). Given a function f and a divisor g where g is a term, this technique defines the quotient of f with respect to g to be the function formed from f by imposing the constraint $g = 1$ explicitly. As an example, let the Boolean functions f and g be given by

$$\begin{aligned} f(w,x,y,z) &= w'xz + xy'z' + wx'z \\ g(w,y) &= wy' \end{aligned}$$

The quotient of f with respect to g is

$$\begin{aligned} f/g = f/wy' &= f(1,x,0,z) \\ &= xz' + x'z\,. \end{aligned}$$

Boolean Algorithms. Some of the algebraic algorithms can be converted to Boolean algorithms by simply replacing algebraic division, in the operations discussed, with Boolean division. Boolean resubstitution is a common example. In the Boulder Optimal Logic Design system (BOLD) it is used in conjunction with algebraic decomposition to improve the quality of the results. In some cases the problems associated with Boolean techniques (namely the computational intensity) are overcome with the application of heuristics to the process. We must be careful so that the loss of optimality by using heuristic techniques does not outweigh the benefits of a Boolean approach.

Spectral Methods. This is a rather new and interesting approach to Boolean minimization. It seeks to transform the input space B^n into one represented in a different basis so that the functions have a more obvious and simple implementation. For example, if function were transformed into an AND or XOR, then the implementation requires only one gate plus the logic to perform the transformation. Brayton goes on to describe spectral

methods as follows:

> An interesting way to look at this topic is to envision the Boolean n-space as a Boolean cube, and a Boolean function f on this space as a set of vertices on this cube. All vertices where $f = 1$ are given a black dot. The objective of the input transformation is to rotate sequentially and transform (like a Rubik's cube) the faces of this cube so that most of the black dots are moved to or near the same face. The transformations of the faces represent intermediate logic functions which create an initial decomposition. When the function is expressed in terms of those intermediate variables, it is much simpler. For example, if all the black dots occupy, after the transformation, an entire face or cube of the space, then the function can be implemented as a single AND term. (17)

More detail on this unique approach can be found in (17) and its associated references.

Recursive Methods. A recursive realization of a combinational logic circuit allows selected outputs to act as inputs or intermediate inputs to another output function. This concept is illustrated in Figure 4.2. The goal of this technique is to take advantage of any redundancies or existing logic in the circuit to reduce the overall circuit cost. The challenge lies in carefully selecting the ordering and dependencies of the equations to generate an optimal solution. This optimization procedure typically produces a result with a multi-level structure. However, we will continue to call it a "recursive realization technique" to differentiate it from the more classical multi-level optimization procedures. The details of this technique are discussed in Chapter 4.

3.3.2.6 Global Optimization Systems. A variety of global optimization systems have been developed over the years and many of them have proven to be quite successful. "A distinguishing feature for most of these systems is the extent to which they are able to exploit the degrees of freedom of the design problem in the optimization process (17)." Below is a brief description of some of the most notable systems. The information has been extracted from articles on these systems. For a more detailed understanding refer to the associated references.

MIS. The Multilevel Logic Interactive Synthesis System (MIS) follows a global optimization strategy, making use of a variety of algorithms including decomposition, factorization, minimization, and timing optimization (14). It can be operated interactively

or incorporated as a batch routine in other automatic digital systems. It starts from the combinational logic extracted from a high-level description of a macro-cell and produces a multi-level set of optimized equations which preserves the original input-output behavior. Its optimization algorithms focus on improving both the area and propagation delays. It first optimizes with respect to the area and then modifies the circuit, at some cost, to meet the specified timing constraints. While it has been used primarily for CMOS designs, its algorithms are flexible enough to support a variety of target technologies. MIS is organized as a set of operators which act on the Boolean network and are controlled either interactively or by a predefined script. The system is capable of producing fast results or spending additional time to produce results that are closer to optimal. MIS currently makes use of don't care conditions only by collapsing the network to two levels and then making use of a two-level minimizer. This places some limitations on MIS. Don't cares can only be used for those networks that can be represented in two-level logic with reasonable efficiency. Also, MIS is unable to use the don't care information that is implicit in a multi-level description of a network. While this system incorporates primarily algebraic techniques, the design team has explored using some Boolean methods such as Boolean resubstitution to improve the overall results. The system has been written in C and run on Unix-based workstations and DEC and IBM mainframes.

BOLD. The Boulder Optimal Logic Design System (BOLD) is a system developed to map combinational logic optimally into standard cell or CMOS gate technologies (9). BOLD incorporates a highly modified version of ESPRESSO (called ESPRESSO_MLT) which has been tailored specifically towards the optimization of multi-level Boolean networks. As in MIS, this system also utilizes the relatively new optimization procedure known as Boolean resubstitution to provide results that are closer to optimal. These improvements, coupled with a new method for multi-level tautology checking, makes this system highly competitive with other optimization systems. While BOLD and MIS are similar in a lot of respects, there are some differences worth noting. The result of mapping from the behavioral description-language CHDL into a multilevel structure can be automatically mapped back to permit more accurate timing verification. BOLD contains some Boolean algorithms which are potentially more powerful but also slower. BOLD operates under a

C-Shell and includes features which allow a user to make inquiries about literal counts, levels of logic, circuit structure, and critical-path delay times. BOLD also incorporates a tautology checker that verifies the equivalence of the logic after each transformation. Experimental results have shown that BOLD can be extremely competitive with MIS, outperforming it in a number of cases. Designers attribute this to the greater power of its optimization primitives.

SOCRATES. A recent trend in optimization systems has been to combine the features of both global and local optimization into one hybrid system (4). SOCRATES has taken that approach. As is illustrated in Figure 3.9, SOCRATES begins with a system of equations that are extracted from a behavioral specification. A two-level minimization is then performed on these equations using the ESPRESSO routines. This reduced system of equations is then decomposed into a multi-level structure using weak division. At this point the Boolean network begins to reflect the structure of the final circuit. SOCRATES was one of the first optimization systems to make timing a critical design consideration. Timing optimization algorithms play a key role in the development of the network structure at this point. After a reasonable structure is developed, multi-level minimization techniques are employed to improve on this structure. These techniques take maximum advantage of any don't-care conditions introduced as a result of the decomposition into a multi-level structure. The next step does a raw conversion from the AND-OR representation to the particular target technology provided in a user-supplied library. Finally, the last step utilizes a rule-based system to perform local transformations in order to optimize the circuit. These techniques were discussed previously. The SOCRATES system has proven to be one of the most effective because it combines the best features of all the techniques we have discussed into one integrated system.

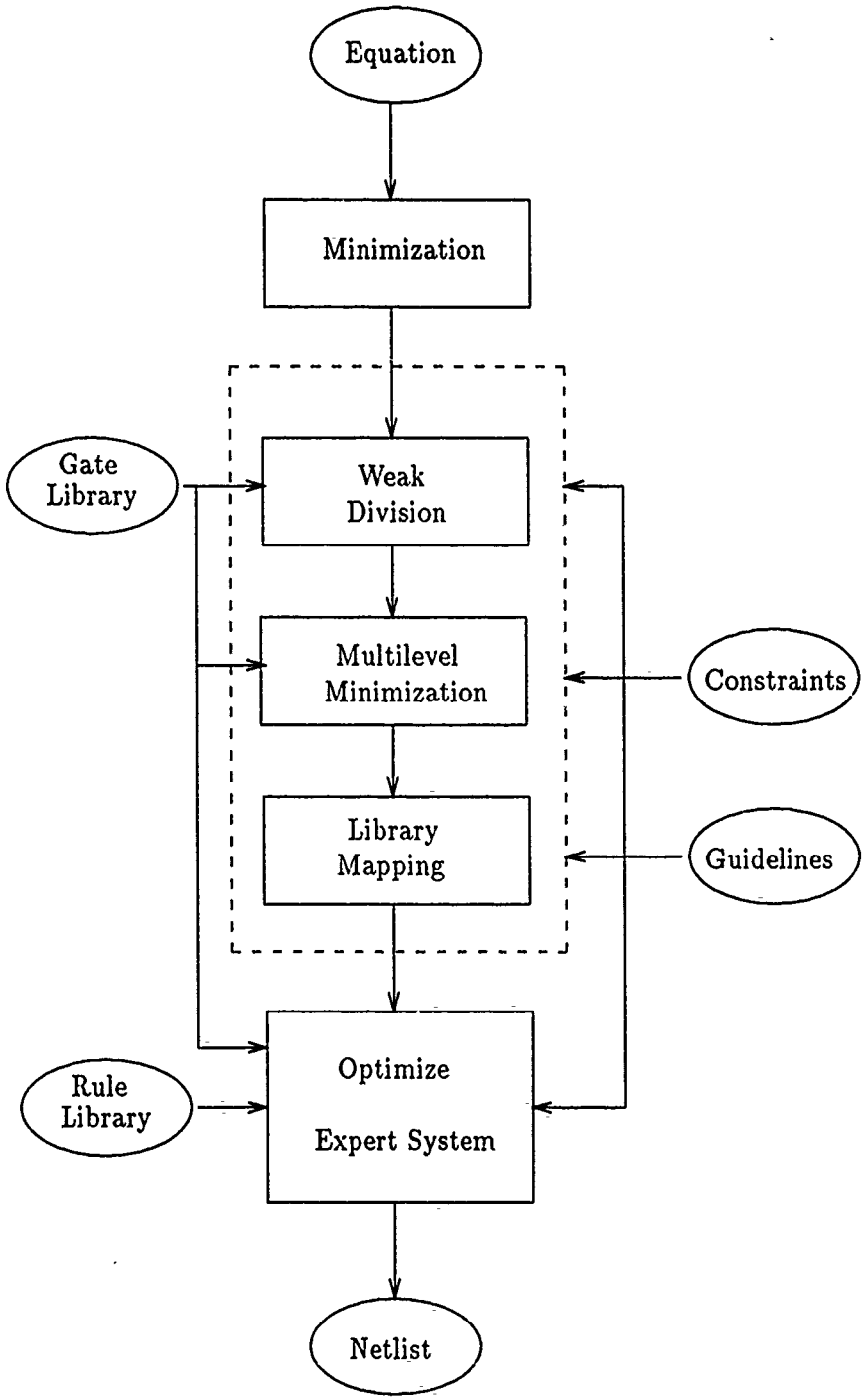


Figure 3.9. Overview of SOCRATES System

CARLOS. CARLOS is an automated multi-level logic design system for CMOS semi-custom integrated circuits (75). Like SOCRATES, it also is a hybrid system, combining both local and global optimization techniques. Its optimization process consists of three steps:

1. two-level multiple output logic minimization

2. multiple output and multi-level factorization

3. technology specific transformations.

The basic design objectives were to reduce the amount of gates necessary to realize the circuit and to reduce the propagation delay along critical paths. The system accepts input in the form of a truth table representation and generates an optimized multi-level multiple output logic circuit, which satisfies the given fan-in and fan-out constraints. Its multi-level synthesis system incorporates two factorization techniques which complement one another. The first is based on the "kernel algorithm" developed by Brayton and McMullen (19) which efficiently handles multi-output functions. A second factorization phase covers the input and output portions of the circuit separately. Factors are iteratively substituted until all fan-in and fan-out constraints are satisfied and no proper common factor exists. Finally a technology mapping process is used to perform local transformations into the target technology. Results have show a 30 to 50 percent improvement over a minimized two-level representation (75).

MACDAS. MACDAS (Multilevel AND-OR Circuit Design Automation System) designs a multi-level circuit with fan-in limited AND-OR gates (91). This system converts a given specification into a two-level AND-OR circuit. Input variables are then paired to produce a two-level AND-OR circuit with two-variable function generators. This is the part of the design process that sets this system apart from most others. The authors have developed efficient techniques to handle what they call "two variable function generators" or TVFGs. Next, some of the outputs are complemented to obtain a circuit with fewer AND gates. The circuit is then converted to a multi-level, fan-in limited AND-OR

circuit. Finally, the circuit is further optimized using local transformations. The system has been programmed in C and Fortran and runs on a personal computer. Experimental results have shown that MACDAS is a useful tool in the design of multi-level circuits, especially for arithmetic circuits.

Yorktown Silicon Compiler. The Yorktown Silicon Compiler was designed as an automated system that would go from an initial behavioral specification to a final structure that could be implemented on a silicon chip (23). The circuit structure is generated automatically from a behavioral description. The final structure consists of registers, ports and blocks of combinational logic and their interconnections. A specification in a specialized Yorktown Logic Language is generated for each block of combinational logic. The final logic is produced by the Yorktown Silicon Editor which incorporates some of the latest multi-level logic synthesis techniques. This system comes close to realizing our ultimate goal of an efficient synthesis/optimization program that takes accepts a behavioral specification and produces an optimal silicon chip design.

BEAT_NP. BEAT_NP is not an optimization system in itself but rather a tool designed to improve the capabilities of BOLD (26). It has long been known that the optimization problem is one of exponential (non-polynomial - NP) complexity. While this problem may not be apparent for smaller circuits, it becomes quite obvious when the numbers of inputs and outputs to a circuit are increased. One way of dealing with this problem is to use algebraic methods to find a quick solution. But to do this we often sacrifice optimality. BEAT_NP was designed as a way to partition a large circuit into smaller ones that could be handled more efficiently by the optimization system. Once each of the partitions has been optimized, the circuit is recomposed with all of the partitions. Naturally, one might expect this to introduce additional costs because one no longer has a true global optimization system. However, the benefits of this tool often outweigh the costs. It has been found that jobs that require weeks of cpu time without partitioning can be done in a matter of hours. It generally achieved 70% of the minimization in only one-third the time (26).

3.4 Summary

We have looked at both two-level and multi-level optimization systems, identifying both the benefits and drawbacks of each. Multi-level optimization systems can produce significant reductions in the costs associated with a circuit but at the expense of increased complexity and time. As the understanding of multi-level optimization techniques approaches that of two-level systems, revolutionary improvements will result.

We have also compared and contrasted the two basic approaches to multi-level optimization: local and global. Each approach has its own unique advantages. It was found that systems such as SOCRATES could effectively utilize both approaches during different phases of the design. Global techniques are preferred for the technology independent portions and local techniques preferred for optimally mapping a circuit into a given technology.

We have discussed the two types of global minimization: namely algebraic and Boolean. Once again it was discovered that each had its own advantages and the most successful systems incorporated both techniques. As we are able to develop more efficient Boolean operations, this will likely become a more popular method because of its ultimate potential to find a global optimum.

IV. Recursive Realizations of Combinational Logic Circuits

4.1 Introduction

With an adequate background behind us, we can now address the central topic of this research effort: building a recursive optimization system. A global approach was investigated, considering all of the equations that specify the circuit at once. This approach was also Boolean-based, taking full advantage of Boolean reasoning principles to achieve the desired results. Specifically, this optimization system involved the recursive realization of combinational logic circuits.

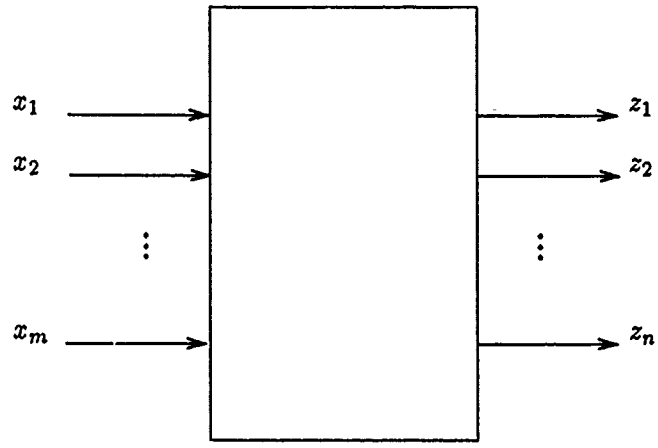
The challenge we face is how to take a set of specifications that describe the desired behavior of a multiple-output system and transform them into an optimal circuit representation. We can think of a multiple-output circuit as a system consisting of an input-vector $X = (x_1, x_2, \cdots, x_m)$ and an output-vector $Z = (z_1, z_2, \cdots, z_n)$ as shown in Figure 4.1. Each signal x_i is a binary stimulus applied as an input to the circuit while each signal z_j is the binary response resulting from some combination of the input signals. We will limit ourselves to the discussion of *combinational circuits*, whose outputs at any given time are a function of the inputs at that time; no dependencies on previous inputs are allowed. The system illustrated in Figure 4.1 can be represented as

$$Z = F(X)\,. \tag{4.1}$$

Typically, the optimization of such a system involves reducing each of the equations $f_1, f_2, \cdots, f_n$ to a minimal form. However, this approach fails to take advantage of any global "don't-care" conditions or redundancies in the circuit. Consequently we will investigate an alternative approach to the optimization problem.

We can begin by asking ourselves what would happen if we allowed the output signals to be used in conjunction with inputs to produce other outputs. In other words, we allow our system to have the form

$$Z = F(X, Z)\,. \tag{4.2}$$



$$z_1 = f_1(x_1, x_2, \cdots, x_m)$$
$$z_2 = f_2(x_1, x_2, \cdots, x_m)$$
$$\vdots$$
$$z_n = f_n(x_1, x_2, \cdots, x_m)$$

Figure 4.1. Multiple-Output Circuit

It turns out that this approach can significantly reduce the the overall cost of a circuit while overcoming problems such as fan-in limitations. Making effective use of available signals, including outputs, is not a new idea. This concept received considerable interest, from early works by the Staff of the Harvard Computation Laboratory (54) and Kobrinsky (63) to more recent efforts by Ho (56), Mithani (78), Pratt (84), and Brown (22). Brown devotes a whole chapter in his book *Boolean Reasoning* to one method of utilizing available outputs; this method involves the *recursive realization* of combinational logic circuits. He presents not only a means of synthesizing a recursive realization of a circuit, but also an approach to automating the process. Much of the following chapter is based on Brown's book and numerous personal discussions.

4.2 Recursive Realizations

While talented designers often find ways to utilize existing signals (including outputs) to produce new ones, most of their techniques rely on skilled visual cognition or heuristic

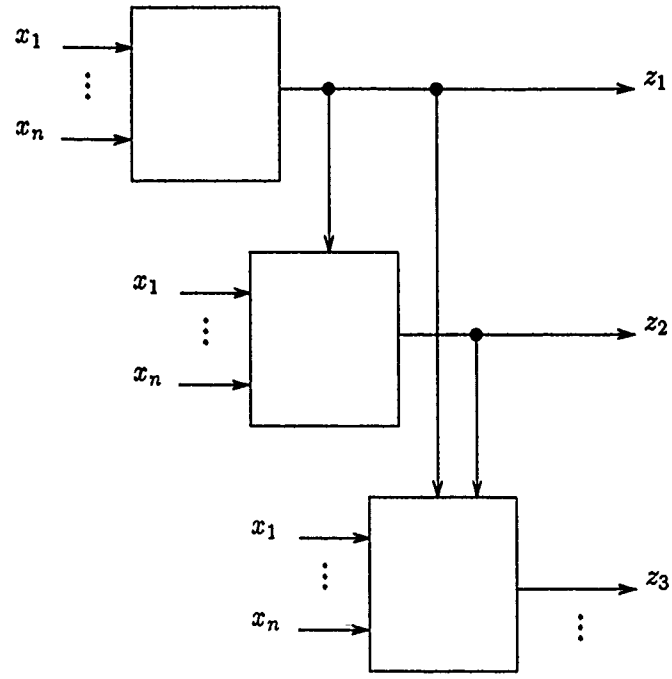


Figure 4.2. Recursive Realization of Combinational Logic

knowledge. These techniques are extremely difficult, if not impossible, to automate. The concept of a recursive realization of combinational logic was developed out of this need for a consistent, repeatable, algorithmic process—a process capable of transforming a circuit specification into a multi-level structure that makes optimal use of existing output signals. This structure, and its underlying form, are illustrated in Figure 4.2. It is important to note that at least one of the outputs in Figure 4.2 (in this case z_1) must depend entirely on inputs. Subsequent outputs can consist of a combination of inputs and previously defined outputs (e.g., z_3 can depend on any input and/or z_1 and/or z_2). This process of defining outputs and then using those outputs in the definitions of subsequent outputs has a *recursive* quality.

4.3 A Recursive Optimization System

A recursive optimization system should be capable of accepting a behavioral specification and producing a recursive realization of the form shown in Figure 4.2. We need to ensure that this specification represents a combinational circuit (not a sequential one). In other words, no output should be defined as a function of itself. Boolean reasoning techniques are applied at various stages in the development of the recursive optimization system. This approach, developed by Brown (22), involves five basic steps:

1. Transform a behavioral specification into a system of Boolean equations.
2. Reduce the system of Boolean equations to a single equation representing the circuit.
3. Perform a dependency analysis to find the *minimal determining subsets*[1] (MDSs) for each output.
4. Assign costs to each of the MDSs.
5. Search the state space for an optimal solution based on the costs incurred.

Each of these steps will be described in more detail in the sections that follow.

4.4 Specifications

There are a variety of ways that the desired behavioral characteristics of a particular system can be represented. They can be in the form of a high level description language such as VHDL. Specifications can also be verbal descriptions, predicate calculus formulas, exhaustive enumerations of input-output pairs, truth tables or a system of Boolean equations. The behavioral characteristics of an AND gate are used in Figure 4.3 to illustrate some of these specification forms.

Through experience, it becomes obvious that truth-tables and Boolean equations are far more convenient than verbal descriptions or exhaustive enumeration. While a truth table helps us to visualize the desired input-output characteristics, Boolean functions provide a compact form that can be easily manipulated using Boolean reasoning techniques.

[1] defined in section 4.6

VERBAL DESCRIPTION

The output of an AND gate is high if and only if all of the inputs are high; otherwise the output is low.

PREDICATE CALCULUS FORMULAS

$\forall x_1, x_2, z_1[z_1 \leftrightarrow x_1 \wedge x_2]$

EXHAUSTIVE ENUMERATION

Input: $X = (x_1, x_2)$

Output: $Z = z_1$

$F = ((0,0),0),((0,1),0),((1,0),0),((1,1),1)$

TRUTH TABLE

x_1	x_2	z_1
0	0	0
0	1	0
1	0	0
1	1	1

BOOLEAN EQUATION(S)

$z_1 = x_1 x_2$

Figure 4.3. Specification Forms For An AND Gate

x_1	x_2	x_3	z_1
0	0	0	1
0	1	0	1
1	0	0	X
1	0	1	0
1	1	0	0
1	1	1	X

Table 4.1. Incomplete Specification

4.4.1 Complete Specifications. One important characteristic of a specification is not what it contains, but what it doesn't contain. A specification is considered *complete* if for every possible combination of inputs there exists one, and only one specified output. Conversely, a specification is considered *incomplete* if a specific combination of inputs results in an output that can be either 0 or 1, or if a specific combination of inputs is forbidden by the specification. Both of these are features of an incomplete system and describe what we have referred to earlier as "don't-care" conditions.

An example of an incomplete system is shown in Table 4.1. The "X" refers to an output that can be either 0 or 1. The input vectors (1,0,0) and (1,1,1) result in an output that can be either 0 or 1. Since the output does not depend on the input, these vectors both represent don't-care conditions. We also note that there are two combinations of inputs (rows in the truth table) that are missing: they are (0,0,1) and (0,1,1). These too represent don't-care conditions.

Don't-care conditions are extremely important because they present the designer with additional degrees of freedom for optimizing a network. These don't-care conditions are *explicit* in the sense that they can be extracted directly from a specification. Recently attention has focused on additional degrees of freedom that are *implicit* to a system (10) (17). These don't-care conditions are often not readily apparent and typically arise from a hierarchically defined specification. In the context of this research effort, we will assume that the specification contains no definitive information concerning the structure of the system. Fortunately by utilizing Boolean reasoning techniques, we can take global advantage of the don't-care conditions that exist.

4.4.2 Tabular Specifications. Another important aspect that we need to consider is whether or not a given specification is *tabular*. A tabular specification is one that can be represented, in its entirety, by means of a truth table (22). In more definitive terms, a specification in normal form,

$$\phi(x_1, x_2, \ldots, x_m, z_1, z_2, \ldots, z_n) = 1 \,, \tag{4.3}$$

is tabular if and only if for each $A \in \{0,1\}^m$, the discriminant $\phi(A, Z)$ is either zero or reduces to a term on the z-variables (22). Currently, almost all circuit synthesis and optimization techniques require a specification to be tabular.

An example of a non-tabular specification can be shown using the information necessary to convert between a JK and an RST flip-flop. This is expressed by the system

$$\begin{aligned} Q'J + QK' &= S + Q'T + QR' + QR'T' \\ 0 &= RS + RT + ST \end{aligned}$$

where the inputs X are $\{J, K, Q\}$ and the outputs Z are $\{R, S, T\}$. This system can be converted to the normal form of specification

$$\phi(J, K, Q, R, S, T) = 1$$

given by

$$\begin{aligned} \phi(J,K,Q,R,S,T) &= J'Q'S'T' + JQ'R'S'T + JQ'R'ST' \\ &\quad + K'QR'T' + KQRS'T' + KQR'S'T \,. \end{aligned}$$

The discriminants of $\phi(J, K, Q, R, S, T)$ with respect to J, K and Q are

$$\begin{aligned} \phi(0,0,0,Z) &= S'T' \\ \phi(0,0,1,Z) &= R'T' \\ \phi(0,1,0,Z) &= S'T' \end{aligned}$$

$$
\begin{aligned}
\phi(0,1,1,Z) &= S'T' + R'S'T \\
\phi(1,0,0,Z) &= R'S'T + R'ST' \\
\phi(1,0,1,Z) &= R'T' \\
\phi(1,1,0,Z) &= RS'T + R'S'T \\
\phi(1,1,1,Z) &= RS'T + R'S'T\,.
\end{aligned}
$$

Four of the discriminants above do not evaluate to either 0 or a term on Z. Consequently the specification is non-tabular. However, it is important to note that any non-tabular specification can be decomposed into a collection of tabular specifications, each of which is sufficient to describe the desired behavioral characteristics of the original specification. In our example there are four discriminants with two terms each. By limiting a discriminant to one term, there are $2^4 = 16$ possible combinations of terms and hence 16 possible tabular representations of the given specification. One such tabular representation in the normal form of specification is

$$
\begin{aligned}
f(J,K,Q,R,S,T) &= J'K'Q'S'T' + J'K'QR'T' + J'KQ'S'T' + J'KQS'T' \\
&\quad + JK'Q'R'S'T + JK'QR'T' + JKQ'RS'T + JKQRS'T\,.
\end{aligned}
$$

The reason we need to introduce this topic is twofold. First, most digital design and optimization systems aren't capable of handling non-tabular specifications. We therefore need to verify that a given specification is tabular before proceeding with the optimization process. Secondly, requiring a specification to be tabular places limitations on the freedom of the designer to describe a system's desired behavioral characteristics in the most general terms. As automated optimization systems improve, we have the potential to take advantage of a non-tabular specification by carefully extracting the tabular representation (one of many) that leads to a least-cost solution. This is currently an active area of research (66) and could provide some improvements to optimization systems in the future.

4.5 System Reduction

For our recursive optimization system we require that the specification be tabular and consist of a system of Boolean equations. This system of equations undergoes a Boolean reduction process that transforms it into a single Boolean equation of the form

$$\phi(X, Z) = 1 \,. \tag{4.4}$$

Equation (4.4) is referred to as the *normal form* for the specification. The process of transforming a system of equations into the normal form was discussed in Section 2.3.2. As an example, this process transforms the system

$$\begin{aligned} z_1 &= x_1 + x_2' x_3' + x_2 x_3 \\ z_2 &= x_1' x_2 + x_1' x_3 \\ z_3 &= x_1' x_2 x_3 \end{aligned} \tag{4.5}$$

into an equivalent specification of the form $f(x_1, x_2, x_3, z_1, z_2, z_3) = 1$, where f is given by

$$\begin{aligned} f = x_2' x_3' z_1 z_2' z_3' &+ x_1' x_2' x_3 z_1' z_2 z_3' + x_1' x_2 x_3' z_1' z_2 z_3' \\ &+ x_1' x_2 x_3 z_1 z_2 z_3 + x_1 z_1 z_2' z_3' \,. \end{aligned} \tag{4.6}$$

By reducing a specification to a single equation, global dependencies and don't-care conditions can be handled uniformly and systematically (22).

4.6 Dependency Analysis

For a multi-input, multi-output system, the number of ways to recursively combine the inputs with previously defined outputs could become inordinately large. We find that the number of possible combinations increases in an exponential (NP-complete) fashion as a function of the inputs and outputs. Without proper constraints, designing systems with even a moderate number of inputs and outputs could become too computationally intensive using recursive means. We therefore seek ways to eliminate the necessity for

performing an exhaustive search through all of the possible combinations. At the same time we must be careful that we don't eliminate any combinations that may result in a good global solution.

One way of constraining the number of possibilities is to develop effective heuristic techniques. Our method accomplishes this by first performing a dependency analysis on the system's variables. We already know that all of the outputs can be expressed in terms of their inputs, but additional dependencies can also be derived. Our goal is to find the *minimal determining subsets* for each output. A *determining subset* for an output is a set of inputs and outputs that can be used, in some combination, to produce that output. A minimal determining subset (MDS) is a determining subset from which the removal of any variable would result in a subset that is no longer sufficient to describe the desired output. For example, the output z_1 from the normalized equation (4.6) can be produced from any one of three possible MDSs: they are $\{x_1, x_2, x_3\}$, $\{x_2, x_3, z_2\}$, and $\{z_2, z_3\}$. While some examples have shown that MDSs do not always yield an optimal solution, they do provide an effective means of reducing the search space. Two techniques for deriving minimal determining sets are summarized below (22):

4.6.1 Redundancy Elimination Technique. A redundancy elimination process is one technique that can be used to find minimal determining subsets. To find the minimal determining subsets for an output z, we begin by taking a Boolean specification and reducing it to a single equation in normal form ($f = 1$). We then can express the output as an *interval* [2] of the form

$$[g, h] = \{z \mid g \leq z \leq h\} , \tag{4.7}$$

where g and h are both Boolean functions. This interval has the effect of bounding the number of possible functions that can be used to express the output z. g represents the lower bound of the interval and is given by:

$$g = f/z' . \tag{4.8}$$

[2] see section 2.1.2

h represents the upper bound of the interval and is given by:

$$h = f/z\,. \tag{4.9}$$

The interval represented by (4.7) is non-empty if and only if the condition $g \leq h$ is satisfied.

Before we can calculate the minimal determining subsets, we need to understand a related topic, *maximal redundancy subsets.* Let $X = x_1, \cdots . x_n$ denote the set of arguments of g and h, and let S be a subset of X. S is considered a *redundancy subset* on an interval if there exists at least one function on that interval in which all of the arguments of S are redundant. An argument is redundant if it can be removed without changing the Boolean formula to a non-equivalent formula. We consider S a maximal redundancy subset on the interval if it is a redundancy subset on the interval and if it is not a proper subset of any other redundancy subset on the interval.

Given a complete collection of maximal redundancy subsets, the problem of finding the corresponding minimal determining subsets is trivial. Each minimal determining subset, T, on $[g, h]$ is nothing more than the relative complement with respect to X of a maximal redundancy subset, S on $[g, h]$ *i.e.,*

$$T = X - S\,. \tag{4.10}$$

Since finding the minimal determining subsets, given the maximal redundancy sets, is very simple, the major problem involves developing efficient techniques for finding the maximal redundancy subsets. Although a variety of apprcaches have been used, this section will concentrate on a search-based technique developed by F.M. Brown (22). To find the minimal determining subsets, we perform a depth-first search, successively removing variables from the interval until the condition described in (4.7) no longer holds. That point on the path, excluding the failure point, represents the set of redundant variables that can be removed without any effect on the interval. The search process proceeds in a depth-first fashion until all of the branches (possible combinations) have been explored. This results in a list of maximal redundancy sets which are then transformed into minimal determining

Subset	Test			Redundant?
ϕ	$v'w'xy'z + vw'x'yz'$	$\leq$	$v'x + vx' + w' + y + z$	yes
$\{v\}$	$w'xy'z + w'x'yz'$	$\leq$	$w' + y + z$	yes
$\{v, w\}$	$xy'z + x'yz'$	$\leq$	$y + z$	yes
$\{v, w, x\}$	$y'z + yz'$	$\leq$	$y + z$	yes
$\{v, w, x, y\}$	$z + z'$	$\leq$	z	no
$\{v, w, x, z\}$	$y' + y$	$\leq$	y	no
$\{v, x\}$	$w'y'z + w'yz'$	$\leq$	$w' + y + z$	yes
$\{v, x, y\}$	$w'z + w'z'$	$\leq$	$w' + z$	yes
$\{v, x, y, z\}$	w'	$\leq$	w'	yes
$\{w\}$	$v'xy'z + vx'yz'$	$\leq$	$v'x + vx' + y + z$	yes
$\{w, x\}$	$v'y'z + vyz'$	$\leq$	$y + z$	yes
$\{w, x, y\}$	$v'z + vz'$	$\leq$	z	no
$\{w, x, z\}$	$v'y' + vy$	$\leq$	y	no
$\{w, y\}$	$v'xz + vx'z'$	$\leq$	$v'x + vx' + z$	yes
$\{w, y, z\}$	$v'x + vx'$	$\leq$	$v'x + vx'$	yes

Table 4.2. Development of Maximal Redundancy Subsets

subsets using (4.10). To illustrate this process, let us find the minimal determining subsets for the specification described by the interval $[g, h]$ where g and h are given by the formulas

$$g = v'w'xy'z + vw'x'yz'$$
$$h = v'x + vx' + w' + y + z\,.$$

The depth-first search process proceeds through the space of intervals derived from $[g, h]$ by the successive removal of variables as shown in Table 4.2. The maximal redundancy subsets found with this search are $\{v, w, x\}$, $\{v, x, y, z\}$ and $\{w, y, z\}$. The corresponding minimal determining sets are $\{y, z\}$, $\{w\}$ and $\{v, x\}$ respectively. The function intervals associated with each of these minimal determining sets are shown in Table 4.3.

It becomes obvious that the efficiency of such an algorithm depends heavily on our ability to determine if a given variable is redundant on a certain interval. Brown's approach to removing variables from a Boolean equation is based two operators, ECON and EDIS (22). These operators, which will be defined later, were developed to implement different forms of a more general process called *elimination.*

Minimal Determining Subset	Function-Interval
$\{v, z\}$	$[y'z + yz', y + z]$
$\{w\}$	$[w', w']$
$\{v, x\}$	$[v'x + vx', v'x + vx']$

Table 4.3. Minimal Determining Subsets and Associated Intervals

4.6.1.1 Elimination. Elimination is one of the fundamental processes of Boolean reasoning. Eliminating a variable x from a Boolean equation involves deriving another Boolean equation that expresses all that can be deduced from the original equations without any knowledge of x. If the deduced equation resulting from the elimination of x expresses the same information as the original equation, we then know that the variable x must be redundant. The concept behind elimination was first introduced by Boole over a century ago (22). Its central point states, that if $f(x) = 0$ is any logical equation involving the literal x with or without other literals, then the equation

$$f(1)f(0) = 0 \tag{4.11}$$

is true, independent of the interpretation of x. It represents the complete result of eliminating x from the equation above. In other words, to eliminate a literal x from a given Boolean equation of the form $f(x) = 0$, we need to successively change x into 1 and x into 0 and then multiply the resulting formulas together. Similarly, if $f(x) = 1$ then the equation

$$f(1) + f(0) = 1 \tag{4.12}$$

is true, independent of the interpretation of x.

Elimination can be illustrated by removing one of the inputs from an AND gate (22). An AND gate with inputs x_1 and x_2 and output z_1 can be characterized by the form $f(x_1, x_2, z_1) = 0$ where f is defined by the equation

$$f = x_1'z_1 + x_2'z_1 + x_1x_2z_1' \,.$$

The result of eliminating x_2 from our equation is

$$g(x_1, z_1) = 0\,, \tag{4.13}$$

where g is given by

$$\begin{aligned} g &= f(x_1, 0, z_1) \cdot f(x_1, 1, z_1) \\ &= (x_1' z_1 + z_1)(x_1' z_1 + x_1 z_1') \\ &= x_1' z_1\,. \end{aligned}$$

From (4.13) we can deduce all possible information about the AND gate in the absence of knowledge concerning x_2. This information is represented by the following equivalent statements:

$$\begin{aligned} x_1' z_1 &= 0 \\ x_1 + z_1' &= 1 \\ z_1 &\leq x_1 \\ (x_1, z_1) &\in (0,0), (1,0), (1,1)\,. \end{aligned}$$

4.6.1.2 ECON and EDIS Operators. Armed with a basic understanding of the concepts behind elimination, it is now possible to define two of the common operators involving elimination (22).

ECON Let $f : \mathbf{B}^n \longrightarrow \mathbf{B}$ be a Boolean function expressed in terms of arguments $x_1, x_2, \ldots, X_n$, and let R, S, and T be subsets of $\{x_1, x_2, \ldots, x_n\}$. We define the function $ECON(f,T)$, called the *conjunctive eliminant* of f with respect to T, by the following rules:

$$\begin{aligned} &(i) \quad ECON(f, \phi) &&= f \\ &(ii) \quad ECON(f, \{x_1\}) &&= f(0, x_2, \ldots, x_n) \cdot f(1, x_2, \ldots, x_n) \\ &(iii) \quad ECON(f, R \cup S) &&= ECON(ECON(f, R), S)\,. \end{aligned} \tag{4.14}$$

EDIS Using the same notation as for $ECON$, we define the function $EDIS(f,T)$, called the *disjunctive eliminant* of f with respect to T, by the following rules:

$$\begin{aligned} &(i) \quad EDIS(f,\phi) &&= f \\ &(ii) \quad EDIS(f,\{x_1\}) &&= f(0,x_2,\ldots,x_n) + f(1,x_2,\ldots,x_n) \\ &(iii) \quad EDIS(f,R\cup S) &&= EDIS(EDIS(f,R),S)\,. \end{aligned} \tag{4.15}$$

It is important to note that because of the way these functions are recursively defined, they can be used to eliminate more than one variable at a time. This is illustrated in the example shown below:

$$\begin{aligned} ECON(f(w,x,y,z),\{w,y\}) &= ECON(ECON(f(w,x,y,z),\{w\}),\{y\}) \\ &= ECON(f(0,x,y,z)\cdot f(1,x,y,z)),\{y\}) \\ &= f(0,x,0,z)\cdot f(0,x,1,z)\cdot f(1,x,0,z)\cdot f(1,x,1,z)\,. \end{aligned}$$

If T is a singleton set, *i.e.*, if $T = \{x\}$, then the eliminants of f are related to the quotients f/x' and f/x as follows:

$$ECON(f,\{x\}) = f/x'\cdot f/x \tag{4.16}$$

$$EDIS(f,\{x\}) = f/x' + f/x\,. \tag{4.17}$$

It can be shown that the conjunctive eliminant of a function in Blake canonical form with respect to a given variable is simply the sum of terms in that form that do not involve that variable (22). In other words,

$$ECON(f,\{T\}) = \sum(\text{terms of } BCF(f) \text{ not involving arguments in } T)\,. \tag{4.18}$$

4.6.1.3 Resultant of Removal of a Variable. Using the elimination-operators described above, F.M. Brown was able to define the *resultant of removal* of variable x from an interval $[p,q]$ to be the interval [EDIS(p,{x}), ECON(q,{x})] (22). It should be noted that the resultant of the removal of a variable from an interval is a subset

of that interval. It follows from this that x is redundant on $[p, q]$ if and only if the condition

$$EDIS(p, \{x\}) \leq ECON(q, \{x\}) \tag{4.19}$$

is satisfied. If p is expressed in an arbitrary sum-of-products form and q is expressed in Blake canonical form, then

- $EDIS(p, \{x\})$ is found by deleting x and x' wherever they occur in a term (if either x or x' appears alone as a term, then $EDIS(p, \{x\}) = 1$);
- $ECON(q, \{x\})$ is found by deleting any term in q that contains either x or x'; and
- condition (4.19) is satisfied if and only if each term of $EDIS(p, \{x\})$ is included in some term of $ECON(q, \{x\})$.

This methodology was applied to the earlier example whose resulting search and variable-removal steps were illustrated in Table 4.2.

Using the inclusion relation shown in (2.10) we can re-express the relation (4.19) as

$$EDIS(p\{x\}) \cdot EDIS(q', \{x\}) = 0 . \tag{4.20}$$

This is often the preferred way to test the resultant of removal of a variable.

4.6.1.4 Summary of Redundancy Elimination Technique. We have discussed this technique of finding minimal determining subsets in moderate detail because it plays such a vital role in our recursive optimization system. To calculate the minimal determining subsets from a given specification, we need to first reduce the system to a single Boolean equation expressed in normal form. For each specified output we find the corresponding interval on f that bounds that output. With each of these intervals, we proceed with a depth-first search that successively removes variables at each level until all paths have been explored. To remove a variable from an interval, the elimination operators $EDIS$ and $ECON$ are used. Once the variable is removed, we check to ensure that the resultant of removal is non-empty, *i.e.*, that the variable is redundant on that interval. As shown in Table 4.2, when the test condition (4.19) fails, the variables removed up to that

point constitute a maximal redundancy subset. Once we have the maximal redundancy subsets, we merely take the complement of each subset, with respect to the set of all variables in our original specification, to find the minimal determining subsets.

4.6.2 Opposing Literals Technique. Since minimal determining subsets play such a key role in the recursive realization of combinational logic, it is worthwhile to explore alternative techniques. One such technique, described by Brown, involves the calculation of what he terms *minimal u-determining subsets* (22). We will assume that we are provided a consistent Boolean equation of the form

$$f(x_1, x_2, \cdots, x_n) = 1 \,. \tag{4.21}$$

Given a partition $\{\{u\}, V\}$ of $\{x_1, \ldots, x_n\}$, we say that the variable u is *functionally deducible* from (4.21) if there exists a Boolean function g such that the equation

$$u = g(V) \tag{4.22}$$

is implied by (4.21). The following procedure will produce a sum-of-products formula, each of whose terms corresponds to a minimal determining subset in terms of u.

1. Express f/u and f/u' in a sum-of-products form as follows:

$$f/u = \sum_{i=1}^{M} p_i \tag{4.23}$$

$$f/u' = \sum_{j=1}^{N} q_j \tag{4.24}$$

   where $p_1, \ldots, p_M$ and $q_1, \ldots, q_N$ are terms.

2. Associate with each pair (p_i, q_j) a sum of literals s_{ij} defined by

$$s_{ij} = \sum (\text{letters that appear opposed in } p_i \text{ and } q_j) \,. \tag{4.25}$$

3. Define a Boolean function F_u by the product-of-sums formula

$$F_u = \prod_{i=1}^{M} \prod_{j=1}^{N} s_{ij}. \tag{4.26}$$

4. Multiply out, deleting absorbed terms, to form a complement-free sum-of-products formula for F_u. With each of the terms in the resulting formula, associate a set of arguments having the same letters; the resulting sets are the minimal determining subsets with respect to the variable u.

The best way to illustrate this concept is with the use of an example taken from (22). We begin with the system specification shown below:

$$\begin{aligned} d &= ab + ac + bc \\ s &= a \oplus b \oplus c \\ u &= abs' + a'b's\,. \end{aligned}$$

This system is equivalent to an equation of the form $f = 1$, where f is given by

$$\begin{aligned} f &= a'b'c'd's'u' + a'b'cd'su + a'bc'd'su' + \\ &\quad ab'c'd'su' + ab'cds'u' + abc'ds'u + abcdsu'\,. \end{aligned}$$

We begin by expressing f/u and f/u' as sum-of-product formulas as follows:

$$\begin{aligned} f/u &= a'b'cd's + abc'ds' \\ f/u' &= a'b'c'd's' + a'bc'd's + a'bcds' + \\ &\quad ab'c'd's + ab'cds' + abcds\,. \end{aligned}$$

We then carry out the labeling procedure described in Step 1

$$\begin{aligned} p_1 &= a'b'cd's & q_1 &= a'b'c'd's' \\ p_2 &= abc'ds' & q_2 &= a'bc'd's \\ & & q_3 &= a'bcds' \\ & & q_4 &= ab'c'd's \\ & & q_5 &= ab'cds' \\ & & q_6 &= abcds\,. \end{aligned}$$

The s_{ij}'s that result from Step 2 are as follows:

$$\begin{aligned} s_{11} &= c+s & s_{21} &= a+b+d \\ s_{12} &= b+c & s_{22} &= a+d+s \\ s_{13} &= b+d+s & s_{23} &= a+c \\ s_{14} &= a+c & s_{24} &= b+d+s \\ s_{15} &= a+d+s & s_{25} &= b+c \\ s_{16} &= a+b+d & s_{26} &= c+s\,. \end{aligned}$$

Carrying out Step 3 and deleting repeated factors, we have

$$F_u = (c+s)(b+c)(b+d+s)(a+c)(a+d+s)(a+b+d)\,.$$

The result of multiplying out and deleting absorbed terms (Step 4) is

$$F_u = abc + cd + abs + bcs + acs\,,$$

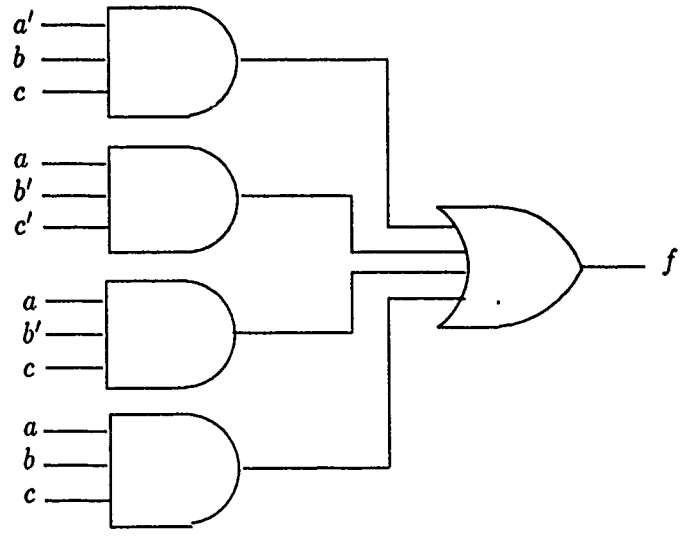
from which the minimal determining subsets are $\{a,b,c\}$, $\{c,d\}$, $\{a,b,s\}$, $\{b,c,s\}$ and $\{a,c,s\}$.

We can see that this technique for finding the minimal determining subsets differs radically from the redundancy elimination technique. Depending on the implementation, it is possible that one or the other might operate more efficiently on a given specification. Therefore it is important that we explore the advantages and disadvantages of these two

BOOLEAN FORMULA

$f = a'bc + ab'c' + ab'c + abc$

TWO-LEVEL CIRCUIT REPRESENTATION



12 inputs + 4 inputs = cost of 16

Figure 4.4. The Cost Based on Gate Inputs

methods in order to make an intelligent selection of the best technique to use.

4.7 Assigning Costs to the MDSs

One of our initial goals in developing the recursive realization technique was to reduce the overall cost of our system design. We seek a measure of cost that is relatively simple to derive from the form of a solution, yet provides a powerful heuristic that will guide us towards an optimal or near-optimal design. While a variety of methods are used, we choose as our cost-measure the *gate-input count.* As its name implies, it simply represents the number of gate-inputs that are present in a system. The cost of any given function in our system will be the gate-input cost of that function if it were implemented in a minimal, two-level, AND-to-OR circuit. We will assume that both the inputs and their complements are available at a cost of zero. This technique of assigning costs is illustrated in Figure 4.4.

To relate a cost to a minimal determining subset, it is first necessary to find a SOP

Output u_i	MDS	Function f_i	Cost
z_1	$\{z_2 z_3\}$	$z_1 = z_3 + z_2'$	2
z_1	$\{x_1 x_2 x_3\}$	$z_1 = x_1 + x_2 x_3 + x_2' x_3'$	7
z_1	$\{x_2 x_3 z_2\}$	$z_1 = x_2' + x_2 x_3$	4
z_2	$\{z_1 z_3\}$	$z_2 = z_1' + z_3$	2
z_2	$\{x_1 x_2 x_3\}$	$z_2 = x_1' x_3 + x_1' x_2$	6
z_2	$\{x_1 x_2 z_1\}$	$z_2 = z_1' + x_1' x_2$	4
z_2	$\{x_1 x_3 z_1\}$	$z_2 = z_1' + x_1' x_3$	4
z_3	$\{z_1 z_2\}$	$z_3 = z_1 z_2$	2
z_3	$\{x_1 x_2 x_3\}$	$z_3 = x_1' x_2 x_3$	3
z_3	$\{x_1 x_2 z_1\}$	$z_3 = x_1' x_2 z_1$	3
z_3	$\{x_1 x_3 z_1\}$	$z_3 = x_1' x_3 z_1$	3
z_3	$\{x_2 x_3 z_2\}$	$z_3 = x_2 x_3 z_2$	3

Table 4.4. Minimal Determining Sets and Associated Costs

formula composed solely of arguments contained in the minimal determining subset. This can be accomplished using a variety of methods including ones that closely model the Quine-McCluskey technique. Whatever the method, one must ensure that the two-level representation is reduced to a minimal form. That way we can ensure that the cost is as small as possible.

To emphasize these points, consider the system given in (4.5). The results of extracting the minimal determining subsets associated with each output, generating a minimal two-level representation from the arguments in the minimal determining subset, and calculating the associated costs, are shown in Table 4.4.

4.8 Search for the Least-Cost Recursive Solution

Using the techniques we have described thus far, we can find the MDSs with respect to each output of our specification and assign a cost to each. The problem now becomes in what order we should select the outputs to produce an optimal recursive solution. First we must recognize that the set of all solutions of a specification in the form $\phi(X, Z) = 1$

can be represented by a *general solution* expressed as the system

$$\begin{aligned}
\alpha_1(X) &\leq u_1 \leq \beta_1(X) \\
\alpha_2(X, u_1) &\leq u_2 \leq \beta_2(X, u_1) \\
\alpha_3(X, u_1, u_2) &\leq u_3 \leq \beta_3(X, u_1, u_2) \\
&\vdots \\
\alpha_n(X, u_1, u_2, \ldots, u_{n-1}) &\leq u_n \leq \beta_n(X, u_1, u_2, \ldots, u_{n-1})
\end{aligned} \tag{4.27}$$

where $(u_1, u_2, \ldots, u_n)$ is a permutation of the output vector $(z_1, z_2, \ldots, z_n)$.

From a general solution (4.27), we can construct solutions of the form

$$\begin{aligned}
u_1 &= f_1(X) \\
u_2 &= f_2(X, u_1) \\
&\vdots \\
u_n &= f_n(X, u_1, u_2, \ldots, u_{n-1})
\end{aligned} \tag{4.28}$$

by independently selecting the functions $f_1, f_2, \ldots, f_n$ that are implicitly represented in the intervals displayed in (4.27). While the set of particular solutions represented by a general solution is unique, the *form* of a general solution may vary widely from one permutation of the output variables to another (22). Let us expand on an earlier example to illustrate this point.

Assume that the desired behavioral specification of a system is of the form

$$f(x_1, x_2, x_3, z_1, z_2, z_3) = 1 \tag{4.29}$$

where f is given in (4.6). Choosing the "natural" permutation $(u_1, u_2, u_3) = (z_1, z_2, z_3)$ of the output variables, a general solution of (4.29) is

$$\begin{array}{rcccl} x_1 + x_2'x_3' + x_2x_3 & \leq & z_1 & \leq & x_1 + x_2'x_3' + x_2x_3 \\ x_1'x_2'x_3z_1' + x_1'x_2x_3'z_1' + x_1'x_2x_3z_1 & \leq & z_2 & \leq & x_1'x_2 + x_1'x_3 + z_1' \\ x_1'x_2x_3z_1z_2 & \leq & z_3 & \leq & z_1'z_2' + z_1z_2 + x_2'x_3'z_1' + x_1z_1' \\ & & & & x_1'x_2x_3 + x_1'x_3z_1 + x_1'x_2z_1 \, . \end{array} \tag{4.30}$$

There exists a large number of particular recursive solutions that can be derived from (4.30). Among the simplest of these is

$$\begin{aligned} z_1 &= x_1 + x_2'x_3' + x_2x_3 \\ z_2 &= x_1'x_2 + z_1' \\ z_3 &= z_1z_2 \, , \end{aligned} \tag{4.31}$$

for which the total cost is $7 + 4 + 2 = 13$. While this represents an improvement over the original system (4.5) whose cost is $7 + 6 + 3 = 16$, it is by no means an optimal solution.

We find by modifying the order in which we recursively choose the outputs, and by intelligently selecting the specific functions within each interval, we can often produce better solutions. For example, the permutation $(u_1, u_2, u_3) = (z_2, z_3, z_1)$ leads to a general solution for which a simplified recursive solution is

$$\begin{aligned} z_2 &= x_1'x_2 + x_1'x_3 \\ z_3 &= x_1'x_2x_3 \\ z_1 &\doteq z_2' + z_3 \, , \end{aligned} \tag{4.32}$$

with an associated cost of $6 + 3 + 2 = 11$. This is an improvement of two gate-inputs over (4.31) and a savings of five gate-inputs over the original solution (4.5).

The preceding discussion leads us to an understanding of the primary motivation for introducing *search* into our recursive realization algorithm. We have shown that by using a purely arbitrary ordering of the output selections, we may find ourselves arriving at a

solution that is far from a global optimum. However, by utilizing an intelligent search process, we can take advantage of available information to guide us on a path towards a better solution.

The type of search process that was originally selected for this problem is a *branch-and-bound search.* It proceeds towards a solution by continually seeking and expanding nodes in the search space that represent the least accumulated cost. In other words, once the node that represents the least accumulated cost is determined, it is replaced by its children on an "open" list. At that point, the search tree is re-examined to determine the next open node with the smallest accumulated cost. This process is repeated until a complete solution path is obtained. What is interesting about this search process is that we are actually constructing the solution as we traverse the tree. Each node in our search tree represents a partial solution to the problem. The solution is found by collecting all of the nodes traversed along the solution path.

Given the context of our problem, there are other observations that can be made with respect to the search process. The first node in our search must represent an output consisting entirely of inputs. Subsequent nodes can represent outputs defined in terms of inputs and/or previously defined outputs. Excluding the root node, the number of levels in the search tree will be equivalent to the number of outputs in our system since at each level, one of the outputs is defined. It is possible that there exist multiple solution paths with identical cost. In this case, all of these solutions will be produced.

With these concepts in mind, we can best illustrate this search process with an example. We will use the system specification given by (4.5) and the associated minimal determining subsets, functions, and costs shown in Table 4.4. The resulting search for the least-cost solution is shown in Figure 4.5. Each step is described in detail below (each node has the form $\{$*output, cost, MDS*$\}$).

The first step is to find an output, expressed entirely in terms of inputs, that has the least cost. In this case the associated node is $(z_3, 3, (x_1, x_2, x_3))$ which defines the output z_3 using the minimal determining subset $\{x_1, x_2, x_3)\}$ at a cost of 3. We then find the

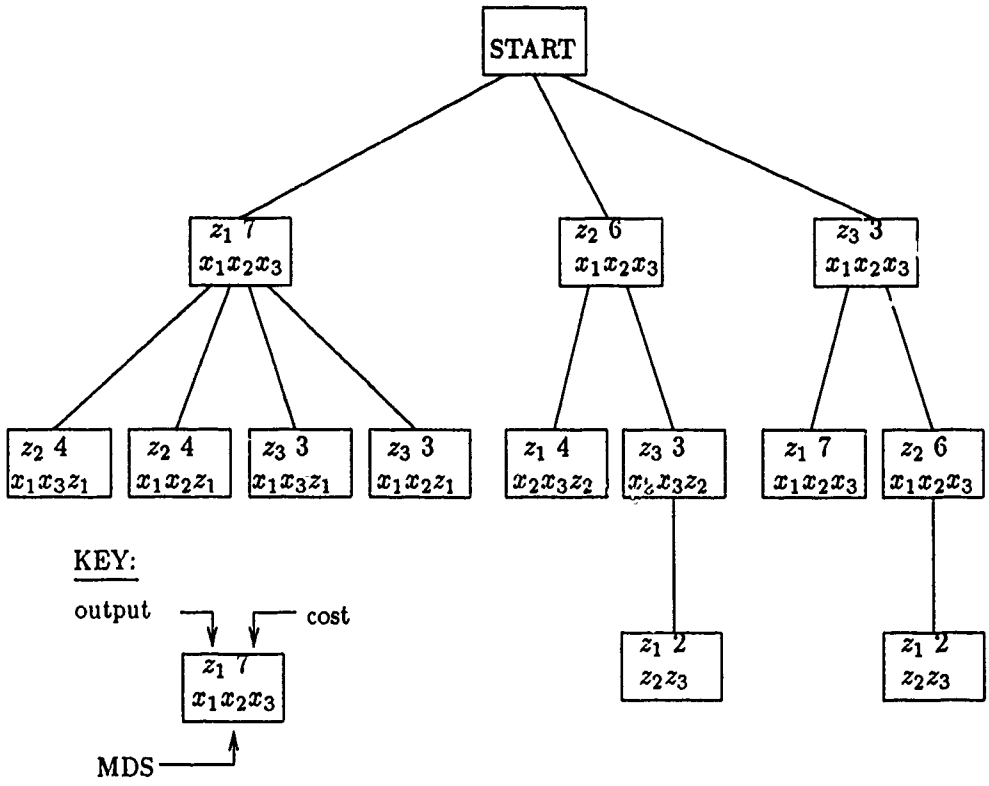


Figure 4.5. Traversing the State Space Using Best-First Search

children of this node[3]. Since this node defines z_3, its children consist of nodes that define z_1 or z_2 and whose minimal determining subsets are composed of some combination of inputs and/or z_3. In this case the children are $(z_1, 7, (x_1, x_2, x_3))$ and $(z_2, 6, (x_1, x_2, x_3))$. We note that their *accumulated costs* are $3+7 = 10$ and $3+6 = 9$ respectively. Since these costs are higher than other nodes which are currently unexplored, we will not expand these children any further at this time.

The next best solution is node $(z_2, 6, (x_1, x_2, x_3))$ which defines the output z_2 using the minimal determining subset $\{x_1, x_2, x_3\}$ at a cost of 6. Once again we find the children of this node and continue the process as before. We can see how the branch-and-bound search process always selects the best available "open" node to expand. This process continues until the search path contains all of the original output arguments; in this example we continue until the solution path contains the arguments x_1, x_2 and x_3 in any order. The first solution we arrive at is our best solution. In our example, the solution consisted of the combination of nodes $(z_3, 3, (x_1, x_2, x_3))$, $(z_2, 6, (x_1, x_2, x_3))$ and $(z_1, 2, (z_2, z_3))$. Table 4.4 lists the functions associated with each of these minimal determining subsets. Thus our final solution,

$$\begin{aligned} z_1 &= z_3 + z_2' \\ z_2 &= x_1' x_3 + x_1' x_2 \\ z_3 &= x_1' x_2 x_3 \,, \end{aligned}$$

is a system of equations representing a recursive implementation of the original specification. The cost of $2+6+3 = 11$ is a significant improvement over the cost of 16 associated with an optimal, non-recursive implementation of this system. Its corresponding circuit representation is shown in Figure 4.6.

As mentioned earlier, the search process does not stop once a solution is found. It will determine any solution sets that have the same cost as the initial solution. In terms of our example, it will list all solutions that can be obtained with a cost of 11. In this case there is one additional such solution, described by the nodes $(z_3, 3, (x_2, x_3, z_2))$, $(z_2, 6, (x_1, x_2, x_3))$

[3]The children of a given node are not shown unless they are later expanded

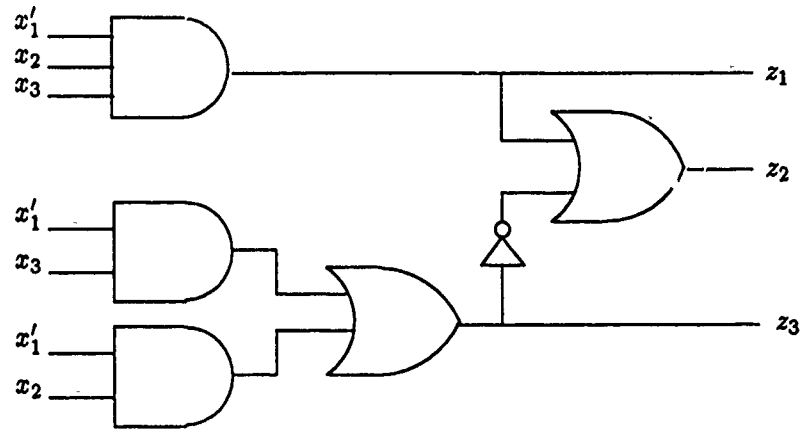


Figure 4.6. Recursive Realization

and $(z_1, 2, (z_2, z_3))$. At this point no more solutions can be found without exceeding the accumulated cost of 11 so the search terminates.

4.9 Summary

While our example achieved a 31 percent reduction in cost versus the original, non-recursive representation, cost reductions of 50 to 75 percent using a recursive realization technique are not uncommon. It is a global optimization approach that holds a great deal of promise for the future. The primary drawback to this approach is the potential for an exponential explosion that makes it too computationally intensive to work for large-scale problems. It is also not necessarily guaranteed to find an optimal, global solution. However, improvements in Boolean reasoning and search techniques are beginning to make progress towards solving some of these problems. This is where much of our effort will concentrate in the remaining chapters.

V. Building a Recursive Circuit Optimization System

5.1 Introduction

At this point it is worthwhile to re-emphasize one of the primary goals of this research effort: to build a recursive optimization system. It should accept a set of Boolean equations that define the behavior of a multiple-output circuit, and return a set of equations that satisfies the specification at a reduced cost. Naturally, we are forced to make several simplifying assumptions to keep this effort within realistic limits. This chapter will provide further information on the scope of the development. It will discuss a variety of design details such as the programming language to be used, the structure and flow of our prototype system, how the system fits into the overall optimization scheme, and the various problems that will be addressed.

5.2 Selection of a Programming Language

One of the early decisions was the selection of an appropriate programming language from which we could develop our prototype system. Our selection was **Scheme**, a small yet powerful dialect of Lisp. Our choice of Scheme was based on a variety of considerations:

1. Scheme facilitates rapid prototyping.
2. Circuits can be easily represented in a list-based form.
3. An extensive library of Boolean reasoning tools has already been developed in Scheme along with a simple global design system.
4. Scheme is available for use on a personal computer (PC).

Without actually working with Scheme, it is difficult to appreciate its power as a prototyping language. It is a language without much structure; there is no distinction between data and functions. One can quickly and easily go from a set of desired behavioral characteristics to a working module. There is no need, as in a classical programming language, for a main program that controls all the subroutines. Each subroutine (algorithm) is capable of running independently, which enables one to test each module as it is developed.

Scheme also facilitates the ability to make frequent modifications to a system without creating unforeseen problems. As long as we are consistent with the inputs and outputs of a given module, we can change its internal composition without affecting other parts of the program. This becomes extremely important when we are continually modifying modules in our system to improve their performance. Often, prototype systems developed in Scheme will eventually be translated into a more conventional language such as "C" to improve their speed and take advantage of more sophisticated graphics and input-output capabilities. However, this generally occurs only after the concept that is being explored has been thoroughly developed in Scheme and is well tested and understood.

The fact that Scheme is a list-based language, and that circuits can easily be described in a list-based form, makes Scheme an obvious choice for this development. To illustrate this concept let us consider an example. The normal-form specification for an Exclusive-OR gate with inputs x_1 and x_2 and output z_1 is $f(x_1, x_2, z_1) = 1$ where f is given by the formula

$$x_1' x_2' z_1' + x_1' x_2 z_1 + x_1 x_2' z_1 + x_1 x_2 z_1' \ .$$

This formula can easily be represented in the list-based form

```
(((X1) (X2) (Z1)) ((X1) X2 Z1) (X1 (X2) Z1) (X1 X2 (Z1)))  ,
```

where the complement of a literal is enclosed in parentheses (*e.g.*, `(X1)`). The OR operators are not shown but are inferred to exist between terms, as illustrated below:

```
(term + term + term + ... + term)  .
```

Scheme is designed to handle these list-based representations in an efficient manner and it is quite easy to develop functions and procedures that manipulate these lists.

A library of tools has been developed in Scheme specifically to facilitate the manipulation of Boolean functions. This collection of tools is called **BORIS**, which stands for BOolean Reasoning In Scheme. It was initially developed by Brown (22) to aid his work with Boolean equations and the concepts behind Boolean reasoning. It includes functions that perform complementation, absorption, Boolean multiplication and division,

elimination, and countless other tasks. It has procedures that enable one to find the Blake canonical form of a Boolean function or perform a variety of other simplification tasks. It includes a parser that accepts a system of equations and translates it into a list-based form that Scheme can understand. P RIS also includes a simple global design system based on some of the concepts discussed in Chapter 4. This wealth of tools available in Scheme, coupled with the fact that Scheme is available in a PC version, led us to the selection of SCHEME as our language for this project.

5.3 Modeling a General Circuit Optimization System

Before we begin discussing the details of our design, it is worthwhile to take a step back and adopt a more general view as to what our requirements are. We would ideally like to build a global, multi-level design system capable of accepting a behavioral specification of a multiple-output system and producing a technology-specific implementation. We would like this implementation to be an optimal or near-optimal solution. One possible approach to such a system is shown in Figure 5.1. We begin by taking a behavioral specification and transforming it into a system of Boolean equations. From this system of equations, our multi-level optimizer produces a least-cost, multi-level representation. This logic structure is generic in the sense that we have not adapted it to any particular technology. That is our next step. Unfortunately once we map our system into a target technology it is often no longer in an optimal form. Consequently, further optimization must take place. This optimization-step may be accomplished using a local redesign technique involving the use of an expert system. The final level in our transformation process is the standard cell; here transistor-level circuits, required to implement a particular logic function, are specified. It should be pointed out that although the process described in Figure 5.1 consists of six distinct steps, our recursive design system addresses only two of them, the two enclosed by a dashed line. This involves taking a system of equations and performing a recursive optimization on them. While all six steps are important, the other four are beyond the scope of this research effort.

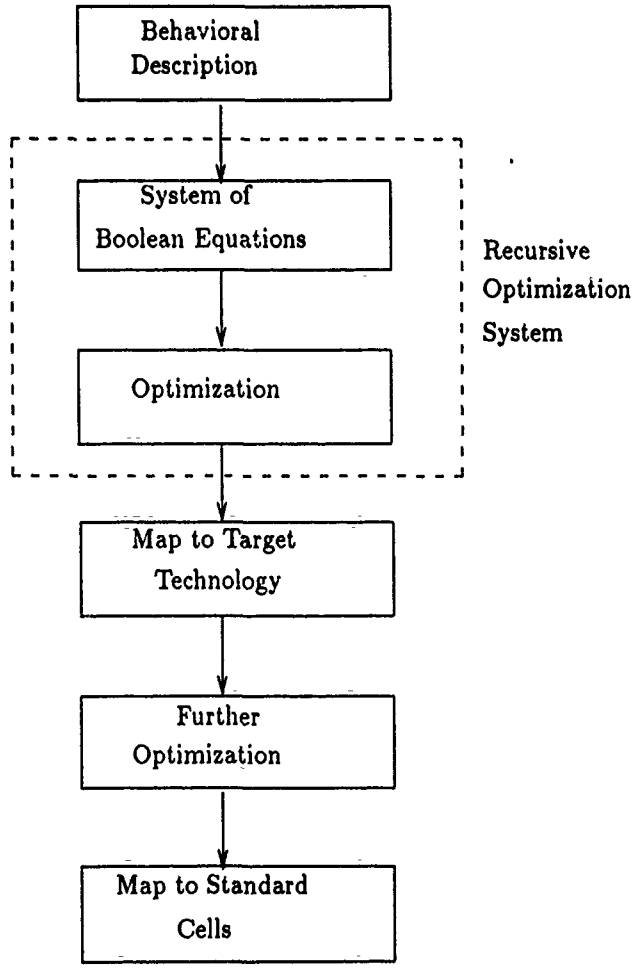


Figure 5.1. A General Circuit Optimization System

5.4 Developing A Recursive Circuit Optimization System

The process involved in the recursive realization of combinational logic circuits was discussed in detail in Chapter 4. Here we intend to discuss how that process can be incorporated into an automated optimization system. We can begin by decomposing the process into smaller modules, each of which perform a specific function. The resulting data flow diagram is shown in Figure 5.2. We begin with a system of equations that describes the desired behavior of a circuit. Each equation is parsed into a list-based form that can be manipulated by Scheme. Using the principles of Boolean reduction, the system of equations is reduced to a single Boolean equation in normal form. We then must ensure that this equation represents a tabular specification. If not, we can convert it to a tabular form that still satisfies the original specification. Once we are sure that our specification is tabular, we perform a dependency analysis on it. This dependency analysis calculates the minimal determining subsets for each specified output. Next, a cost and associated function are found for each of the minimal determining subsets. Finally, using the minimal determining subsets, costs, and functions, a branch-and-bound search is performed to determine a least-cost solution. By "least-cost" we mean a least-cost solution in terms of the search space we have defined using the minimal determining subsets and their associated costs. The final output is a system of equations that represents a least-cost, recursive realization of the original specification.

5.5 Modification of the BORIS Multi-Level Design System

At the point this research effort was undertaken, all of the steps shown in Figure 5.2 were implemented and operating in Scheme except a module that determines whether or not a given specification is tabular. However, it is important to note that the design system was initially built as a "proof of concept" with little emphasis given to its ultimate speed. This became readily apparent when the design system took unreasonably long times to arrive at solutions from simple specifications involving no more than four or five variables. However, since it has been shown that an automated recursive realization system is achievable and can significantly reduce the cost of a circuit, we can concentrate our efforts on improving the speed and efficiency of every aspect of this system. The success or failure

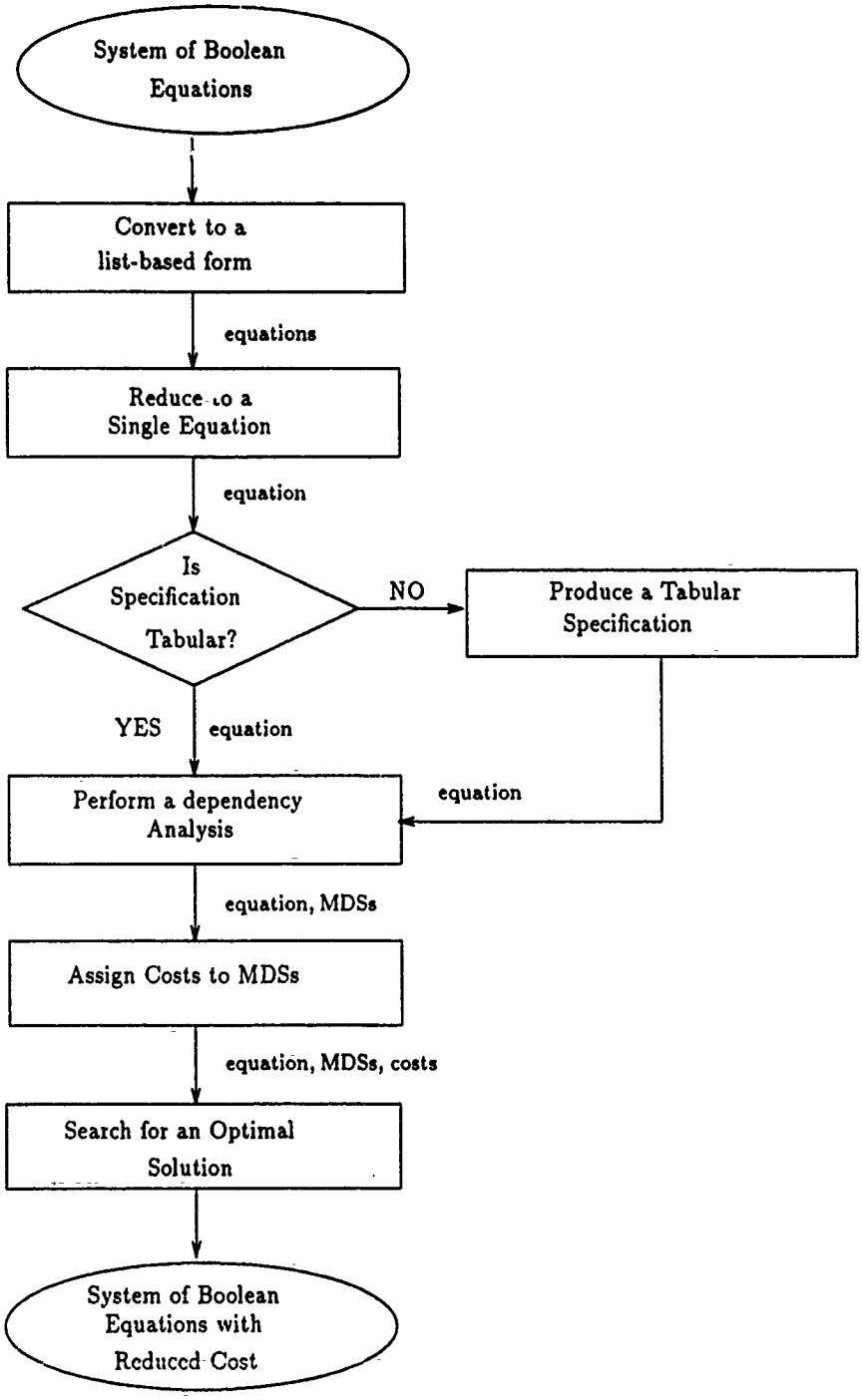


Figure 5.2. Data Flow Diagram

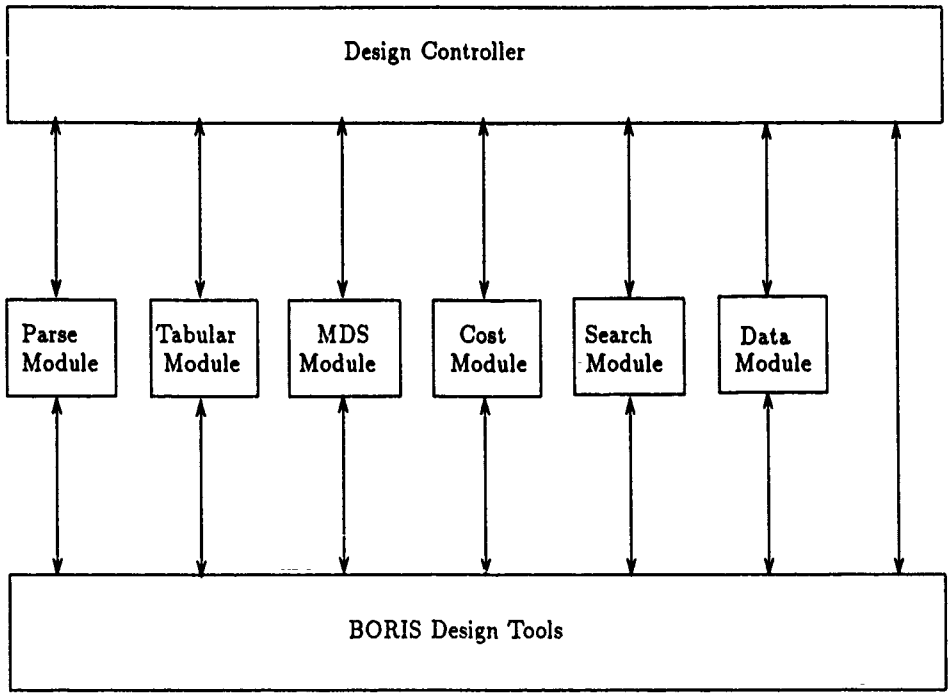


Figure 5.3. The BORIS Multi-Level Design System

of this system hinges on whether or not it can produce adequately optimized circuits in a reasonable amount of time.

Our modified BORIS design system is divided into the separate functional modules shown in Figure 5.3. Each of these modules represents a Scheme file that contains a collection of functions and procedures designed with a specific goal in mind. The *Design Controller* is the main program. It calls all the appropriate functions and passes them the appropriate information. It is responsible for the flow and control of the whole process including the format of the output. The *Parse Module* actually has a dual function; it parses a system of equations into a list-based form and then reduces the equations to a single equation in normal form. The *Tabular Module* determines whether or not a given specification is tabular. If it is not, it converts it to a tabular form. The *MDS Module* finds the minimal determining subset for a given output. The *Cost Module* finds the cost and associated function for each of the minimal determining subsets. The *Search Module* uses the information gathered concerning each minimal determining subset to find a least-cost solution. The *Data Module* is where all pre-defined specifications are located. The

user has the option of adding a desired specifications to this file or simply inputting the specifications as the optimization procedures are used. All of these modules make use of specific functions and procedures defined in the *BORIS Design Tools* file.

5.6 Summary

The goal of this chapter was to provide an overview of the thought that went into the design and development of our global, recursive optimization system. We discussed the reasons for the selection of Scheme as a programming language. We introduced the functions of a general optimization system and pointed out those functions that are developed in this research effort. We addressed the development of an automated, recursive circuit optimization system and illustrated the flow of data through it. We introduced the BORIS design system and identified its various existing and proposed features. With all of this in mind, we can now focus our attention on the specific problems at hand and the design approaches aimed at solving them.

VI. Detailed Problem Analysis and Design

6.1 Introduction

A recursive optimization system was successfully implemented in Scheme as part of the BORIS toolset. It was not our intent to discuss the thought and detail that went into the design of the original system. Instead, the primary goal of our research has been to analyze this system, seeking ways to improve its speed and efficiency while at the same time evaluating its effectiveness. We proceeded by carefully studying the performance of the BORIS design system and identifying specific areas where improvements could be made. Once problem areas were identified, we designed, developed and implemented specific techniques aimed at solving them. It is important to point out that the problem analysis and design tasks were accomplished simultaneously with the testing. It was ultimately the results of periodic testing that identified specific problem areas and drove the design process. Most of the problems were handled through the modification of existing algorithms, but some involved the introduction of new techniques. Not all of the modifications resulted in dramatic improvements to the design system. This is not a major concern of ours since it is just as important to rule out those techniques that do not improve the system's performance as it is to identify those that do.

6.2 Performance of the BORIS Optimization System

To effectively enhance the system's performance, it is imperative that we know all we can about its current operation. For this reason a series of preliminary tests were conducted on the BORIS optimization system. For this testing, the system was divided into three distinct parts:

1. Parsing and reduction of a system of Boolean equations.
2. Calculation of the minimal determining subsets and their associated costs.
3. Searching for an optimal or near-optimal solution.

Using a variety of sample circuits the resulting run-times, for each of these categories, were recorded in Table 7.1. While we can not draw too many conclusions from such a small

sampling of circuits, it was obvious that the time spent calculating the minimal determining subsets and parsing the Boolean equations far outweighed any time spent searching for a least-cost solution. The average relative proportions of the runtimes, for each circuit, are shown on the pie chart in Figure 7.1. These findings contributed significantly to the direction this research would take. It would have been fruitless, for example, to concentrate all our efforts on improving the speed of the search process when it currently constitutes only a small percentage of the total run-time. Therefore, our intent was to distribute our efforts amongst a variety of key objectives, with each objective addressing a specific deficiency in the system.

6.3 Integrating a Tabular Design Module

In Section 4.4.2 we justified the need for a tabular design-filter. We found it was necessary that a given specification be tabular in order for the design system to function properly. The question becomes how to handle non-tabular specifications. To address this problem, a *Tabular Module* was developed in Scheme with two specific goals in mind. The first goal was to design a filter that could be used to determine whether or not a given specification is tabular. If the specification was found to be non-tabular, then our second goal was to convert the specification to a tabular form. A non-tabular specification may have multiple tabular forms; we need to select only one of the tabular representations to pass on to the optimization system.

The code for the Tabular Module, which was developed in Scheme, can be found in Appendix B. It should be emphasized that this module was developed to work independently or in conjunction with our design system. When working with the design system, we already have available a specification that has been reduced to normal form. In addition, at this point we are not interested in displaying the result, but instead simply returning it. For these reasons, the slightly modified calling procedures TABULAR-SPEC? and MAKE-TABULAR-SPEC were introduced; the first procedure checks to see if a specification is tabular and the second procedure converts a non-tabular specification into a tabular form.

6.3.1 Tabular Design Filter. The first challenge we faced was building a tabular design filter. It should accept as its input an equation (or set of equations) and a list of specified outputs (arguments). It should return a "true" response if the system described by the equation(s) is tabular with respect to its inputs or return a "false" response if the system is non-tabular. In Scheme, a "true" response is indicated by #T while a "false" response is indicated by (). Also, we need to assume that the inputs consist of all of the arguments in the system that are not specified as outputs.

The algorithm we developed uses a recursive process to generate all of the *discriminants*[1]. By definition, if any of the discriminants evaluates to something other than zero or a term on the designated output variables, then the specification is non-tabular and the algorithm returns '(). Otherwise, if all of the discriminants evaluate to either zero or a term on the output variables, then the specification is tabular and the algorithm returns #T.

The tabular design filter is called in Scheme using the format

```
(TABULAR? EQUATIONS ARGS)
```

where EQUATIONS represents a Boolean equation (or set of Boolean equations) and ARGS represents the desired outputs. This operation is illustrated in the example shown below:

```
[1]  (tabular? '("f = x' + y z"
                 "g = x y' + z'"
                 "h = x' + y' + z'")    '(f g h) )
#T
```

When used in conjunction with the Design Module, the format used to call this algorithm is

```
(TABULAR-SPEC? SPEC ARGS)
```

where SPEC represents a previously parsed specification in normal form and ARGS represents the desired outputs.

[1] Discriminant is defined in Section 2.2.4

6.3.2 Non-Tabular to Tabular Conversion Algorithm. We need to build an algorithm that is capable of converting a non-tabular specification into a tabular form. It should accept as its input a specification consisting of one or more Boolean equations and a list of proposed outputs. By knowing the specified outputs we should be able to determine the corresponding inputs. With this information, we have the data necessary to calculate the corresponding discriminants. As the discriminants are generated, they should be filtered to ensure that each discriminant consists of either a zero or a single term. In the case that more than one term exists for a given discriminant, we need to ensure that this multi-term SOP formula cannot be represented by an equivalent single term. If it cannot, then the first term is arbitrarily selected and the rest are eliminated.

The procedure described above was implemented in Scheme and can be called using the following format:

```
(MAKE-TABULAR EQUATIONS ARGS)
```

where EQUATIONS represents a Boolean equation (or set of Boolean equations) and ARGS represents the desired outputs. This operation is illustrated in the example shown below:

```
[1]  (make-tabular '("q' j + q k' = s q' t + q r' t'"
                     "0 = r s + r t + s t")
                   '(r s t) )

How Do You Want to Display Your Result?

1.  Raw List Form
2.  Horizontal SOP Form
3.  Vertical SOP Form
4.  Reduced Form

What Choice Do You Want To Select?  4

1 = J'K'R'S'T' + J'Q'S'T' + J K'R'S T' + J Q'R'S T'
    K Q R'S'T + K'Q R'T'
()
```

When used in conjunction with the design module, the algorithm is called using the format

```
(MAKE-TABULAR-SPEC SPEC ARGS)
```

where SPEC represents a previously parsed specification in normal form and ARGS represents the desired outputs.

6.4 Improving the Efficiency of our System

The efficiency of our system refers to its maximal utilization of existing knowledge to achieve the desired results in the quickest time possible. Our strategy for improving the efficiency of the global optimization system was to

- Understand the purpose and operation of each of the fundamental modules,
- Seek ways to improve the efficiency by enhancing existing design techniques,
- Seek ways to improve the efficiency by introducing new design techniques, and
- Improve the overall system efficiency by modifying some of the basic Boolean tools used by the design system.

6.4.1 The Parse Module. The Parse Module can be found in Appendix B. It is called in Scheme using the format

```
(PARSE SYSTEM)
```

where SYSTEM represents a Boolean equation (or set of Boolean equations). The resulting output is a single, list-based Boolean equation in the form $F = 0$. An example of this process is shown below:

```
[1]  (parse '("f = x' + y z"
              "g = x y' + z'"
              "h = x' + y' + z'") )

(((G) (Z)) (G (H)) ((Y) (H)) ((X) (H)) ((Z) (H))
((F) (X)) ((F) (H)) ((F) (G)) ((G) (Y) X) ((G) H X)
((X) G Z) (Y G Z) (F (Z) X) (F H X) (F (Y) X) (F G X)
(F G Z) ((F) Z Y) (Z Y H X))
```

Preliminary tests indicated that the parse module consumes a substantial amount of the overall design time. This justified a more detailed examination of the parsing system. One seemingly interesting question is why do we parse the system into the form $F = 0$ and then take the complement? In other words, why not just parse the system directly into the preferred normal form $F = 1$ that the design system requires. It appears on the surface that we may be able to achieve a speedup by doing this. The answer becomes apparent when we look at the system reduction process.

To reduce a system of equations into a single equation of the form $F = 0$ requires only that we take the sum (OR) of all the individual equations once they are reduced to the form $F = 0$. This was shown earlier in Section 2.3.2. Since the resulting equation remains in an SOP form, no further manipulation is necessary. Efficient complementation routines can take this SOP formula and quickly convert it to the desired normal form. On the other hand, to reduce a system of equations directly to the form $F = 1$ would require that we take the product (AND) of all the individual equations, once they are reduced to the form $F = 1$, as was shown in Section 2.3.2. In this case the resulting single equation is not in a convenient SOP form. To convert it to a SOP form requires a time-intensive Boolean multiplication procedure. It turns out that this multiplication consumes more time than a complementation would. Therefore, we conclude that the method of reducing a system of equations to the form $F = 0$ and then taking the complement is currently the most efficient technique.

Before we examine other approaches aimed at improving the parser, it is worthwhile to understand, in general, its current operation. It was developed using a formal approach. This approach consists of the following basic steps:

1. The string representation of each Boolean formula is converted into equivalent tokenized form.
2. The tokenized list is then transformed into a prefix AND-OR-NOT representation.
3. The AND-OR-NOT representation is then converted to an equivalent list-based SOP form.
4. Steps 1 through 3 are repeated for each Boolean formula in a system, with the resulting list-based SOP formulas being added together.
5. The final result is then reduced to an equivalent *sub-minimal form*[2].

A careful analysis of this system revealed two modifications that could likely improve the overall efficiency. One modification would involve the complete redesign of the parser system. While the formal structure of the original parser adds to its generality and flexibility, it accomplishes this at the expense of the parser's efficiency. If we completely redesigned the parser specifically for the requirements of our particular global design system, a moderate speedup would likely be achieved. This might involve eliminating one or more of the steps above: it might possibly entail going directly from a string-based form to a list-based SOP form. This seemed like the obvious choice until preliminary tests showed us that as much as 90 percent of the parsing time was spent on Step 5.

What makes this observation even more interesting is the fact that after spending a significant amount of computational time reducing our system to a sub-minimal form, we then take its complement. It may not be intuitively obvious, but if a sub-minimal formula is complemented, the result is no longer in a minimal form. We therefore have sacrificed all the time spent reducing this formula and subsequently are forced to minimize the formula once again. It should be noted that the parser was not developed with circuit optimization in mind. It may indeed function very well for a variety of other applications.

To overcome this problem, Step 5 was simply removed from the process when performing a circuit optimization. The formula is then simplified only after the complementation takes place. Since most of the computational time involved in parsing a system

[2]A sub-minimal form is a Blake canonical form with redundant terms removed in any order.

of equations was tied up in Step 5, significant improvements in speed were realized. The only potential drawback is that the formula to be complemented is more complex. Early indications showed us that the small increase in complementation time was far outweighed by a drastic reduction in time spent reducing the system to a sub-minimal form.

6.4.2 The Minimal Determining Subset Module. A copy of the MDS Module can be found in Appendix B. It contains a collection of algorithms aimed at calculating minimal determining subsets for use in our optimization system. It is called in Scheme using the format

```
(MIN-DETERMINING F Z)
```

where F represents the parsed specification in normal form and Z represents the output that the minimal determining subsets describe. An example of this process is described below:

```
[1] (min-determining            ;; find MDSs
      (simplify                   ;; reduce formula
        (complement                 ;; convert to normal form
          (parse "f = x' + y z"       ;; parse system
                 "g = x y' + z'"
                 "h = x' + y' + z'" )))   'z)

((G X) (H X) (X Y Z))
```

The result tells us that for this system of equations, the output z can be created from {G X}, {H X} or {X Y Z}. Naturally z could also be constructed from a superset of any of these sets but not a subset.

Preliminary tests indicated that the MDS Module was excessively slow, typically consuming 50 percent and often up to 90 percent of the total optimization time. This was a surprising discovery that led us to believe there may exist better ways to find minimal determining subsets. With that possibility in mind, improving the speed of the MDS Module became one of the central topics of this research effort.

Earlier we discussed two basic methods for calculating minimal determining subsets: the redundancy elimination technique (Section 4.6.1) and the opposing literals technique (Section 4.6.2). It is our intent to examine these techniques in more detail and propose possible modifications to our optimization system based on our findings. Our ultimate goal is to improve the efficiency of this process to such an extent that it no longer consumes a major portion of the design time.

6.4.2.1 Redundancy Elimination Technique. The redundancy elimination technique was the method used in the original optimization system. As was described in Section 4.6.1, the redundancy elimination technique employed a depth-first search process that involves the successive removal of variables. This technique is illustrated in Table 4.2. It proved to be successful in finding the minimal determining subsets, but unfortunately it takes an inordinate amount of time. If one wished to improve the efficiency of this technique, it is likely that one would need to concentrate on improving the efficiency of its search process. Although this technique might deserve a more detailed analysis, it was not attempted in this research effort. Instead, a variety of alternative approaches were investigated.

6.4.2.2 Opposing Literals Technique. The opposing literals technique was discussed in some detail in Section 4.6.2. Because of its simple step-by-step approach, this process was quickly prototyped in Scheme. Unfortunately, preliminary tests on this prototype were not very encouraging. In fact, Scheme consistently ran out of memory, even when processing rather simple circuits. The problem was traced to Step 4 in described in Section 4.6.2. In this step, a product-of-sums (POS) formula is multiplied out to produce an equivalent SOP formula. As this multiplication is carried out, the number of terms increases at an exponential rate. It does not take too many multiplications before the number of terms exceeds Scheme's internal memory. Fortunately, there are some ways to get around this problem. It is important to note that most of the terms that are generated are redundant and hence can be eliminated without any effect on our resulting formula. It was this observation that led to the development of the following approach.

Multiplication and Absorption Process: To eliminate any redundant terms, we will use a Boolean property called *absorption*[3]. An algorithm that performs absorption has already been developed and can be found in the BORIS Design Tools Module shown in Appendix B. The next problem becomes how to integrate absorption into our multiplication process. We can not wait until the entire multiplication process is complete because Scheme will run out of memory long before that happens. Therefore, we will implement a process that repetitively performs multiplications and absorptions until the entire formula is transformed into a SOP form. To illustrate this idea, let us use an example. To transform the POS formula

$$(s+c)(b+c)(b+d+s)(a+c) \tag{6.1}$$

into an equivalent SOP form, proceed in the following fashion:

$(s+c)(b+c)$	*select multiplicands*
$(bs+cs+bc+c)$	*multiplication*
$(bs+c)$	*absorption*
$(bs+c)(b+d+s)$	*select muliplicands*
$(bs+bds+bs+bc+cd+cs)$	*multiplication*
$(b+bd+cs)$	*absorption*
$(b+bd+cs)(a+c)$	*select multiplicands*
$(ab+abd+acs+bc+bcd+bcs)$	*multiplication*
$(ab+bc+acs)$	*absorption* .

Using thi process, we arrived at the correct solution. It is important to note that the largest number of terms that we deal with, using this process, is six. This compares to 24 terms that the normal multiplication process would encounter.

This multiplication and absorption process was integrated into the opposing literals algorithm. Preliminary tests were quite encouraging. They showed a significant speedup using this method of calculating minimal determining subsets versus the original redun-

[3]The absorption property is shown in (2.19) and (2.20).

dancy elimination technique. This en ouraged us to take a closer look at this method and search for an even more efficient means of transforming a POS formula into an equivalent SOP form. One new approach involved the use of a Boolean Expansion Process.

Boolean Expansion Process: The idea here is to break a long POS formula down into simpler components which can be handled much easier by Scheme. This is accomplished using a techn'que called Boolean Expansion which was introduced in Section 2.1.5. Using this technique, we simply expand the POS formula until no further expansion can take place, at which time the resulting formula is in a POS form. By using this technique, we avoid having to use multiplication at all. We perform absorption only once, after the formula has been reduced to SOP form.

An algorithm that performs this Boolean expansion process was designed and implemented in Scheme. It is important to note that we designed this algorithm around the special features of our opposing literals process. The POS formula that is generated does not contain any literals in a complemented form. With this in mind, the expansion process can be described as follows.

The variable x, with respect to which we expand, is arbitrarily chosen as the first variable in the formula. Let R be the product of all the factors of f involving x, with x set to 0. Let S be the product of all the factors of f not involving x. An expansion for f is

$$f = xS + RS\,. \tag{6.2}$$

It turns out that xS and RS are generally much simpler than the original formula. However, the process does not stop here. Instead, Boolean expansion is applied to S and RS. This process continues in a recursive fashion until no further expansions can take place. This entire process was built into a recursive Scheme procedure that can be found in Appendix B.

To illustrate one step in this expansion procedure, let us use the same example we did earlier (6.1). Using the definitions for x, R and S described above, we derive

$$\begin{aligned} x &= s \\ R &= (c)(b+d) \\ S &= (b+c)(a+c)\,. \end{aligned}$$

Substituting these values into equation (6.2) produces the expansion

$$f = (s)[(b+c)(a+c)] + [(c)(b+d)(b+c)(a+c)]\,. \tag{6.3}$$

We note that the POS formulas on either side of the + are simpler than the original POS formula. Expansion would then proceed with each of these POS formulas. This recursive expansion process continues until f is expressed as a SOP. At that point, absorption is carried out to remove all redundant terms.

Preliminary tests once again were very encouraging. They showed that this expansion technique provides significant improvements over the multiplication/absorption process. Despite this success, we decided to carry this process one step further.

Expansion Process With Intelligent Selection: The selection of an expansion-variable is currently an arbitrary process. We simply choose the first variable to appear in the formula. This choice is seldom the best one. We discovered that if the variable that appeared most often in an expression were chosen first, the expansion process would proceed faster towards a solution. Once again using the same example shown in (6.1), we can define x, R and S as follows:

$$\begin{aligned} x &= c \\ R &= sba \\ S &= (b+d+s) \end{aligned}$$

The value of x was chosen to be c because c appears the most often in our expression, a total of three times. Substituting these values into equation (6.2) we have:

$$f = abcs + (abs)(b + d + s) \tag{6.4}$$

It is easy to see that equation (6.4) is much simpler than equation (6.3) and closer to a solution in SOP form. This illustrates how an intelligent selection of the expansion-variable, can lead us to a quicker solution by reducing the number of expansion steps. However, this is not the only benefit. It also reduces the number of terms that appear in the result before absorption takes place. Since the fewer the terms in an expression, the faster the absorption algorithm can run, we gain an additional speedup.

6.4.3 The Search Module

6.4.3.1 Background. Search plays a critical role in our circuit optimization process. Not only does it affect the speed at which we arrive at a solution, but it also may affect the quality of the solution itself. Unfortunately, to attain an optimal solution we often have to sacrifice the efficiency of our process and conversely to attain an efficient process we often sacrifice the optimality of our solution. Building a practical optimization system almost always involves a compromise between efficiency and optimality.

The number of distinct solutions that will satisfy a given specification is infinite. Searching for optimal solutions in an infinite state space is neither efficient nor practical. Consequently, our first task is to reduce this infinite search space to a practical size. Fortunately we have already done this. Recall that the purpose for introducing minimal determining subsets was to reduce the possible combinations of variables that could be used to define each output. In addition, we ensure that for any combination of variables, a unique circuit representation exists.

While the use of minimal determining subsets substantially reduces the search space, in some circumstances it may also prevent us from obtaining an optimal solution. Unfortunately, that is the price we have to pay for the improved performance of our system. Now that we have effectively reduced the search space, we need to identify the most effective

search algorithm to use.

6.4.3.2 Selecting A Search Process. Search processes are generally divided into two categories: *informed search* and *uninformed search.* With an uninformed search process, the way the state space is searched is determined by the search method and not the information available at each state. Some common examples include *depth first search* and *breadth first search.* On the other hand, informed search uses available information to guide the search through the state space. It is often preferred because, by utilizing available information, one can make an intelligent decision as to where to go next. This often leads one to a quicker and sometimes better solution. Some common examples include *hill-climbing, branch-and-bound search* and A^*. Since we have information available that could be used to guide our search, we need to select an informed search process that best suits our needs.

Hill-climbing proceeds in a depth-first fashion, utilizing the local knowledge surrounding a given state (node). Its main drawback is that it may arrive at a solution that is a local optimum and fail to recognize a better global solution. Without exhaustively proceeding through the entire search space, we have no way of knowing whether or not a given solution is the optimal one. We will therefore consider an alternative approach, using a branch-and-bound search.

Branch-and-bound search maintains a list of partial paths containing the accumulated costs from the start node to the current open nodes. The open node that is selected for evaluation, is always the one with the smallest accumulated cost. Once a node is selected, its children are evaluated and placed on the "open list." The next node to be selected may be one of the children of the current node or a previously evaluated node; whichever represents the least accumulated cost. This search process continues until a solution is obtained. The solution is guaranteed to be a least-cost solution in terms of the costs we have established. However, this may or may not represent a global optimum (i.e. a better global solution may exist outside of our search space).

The branch-and-bound search technique is the method that we chose for our optimization system. It is an effective heuristic search process that proceeds rapidly towards a

least-cost solution. For those familiar with the available search techniques you may wonder why we do not use A* search, which is often guaranteed to return an optimal solution. The reason is that A* search not only relies on information accumulated along the search paths, but also relies on an estimate of the cost remaining to reach a global optimum. Currently, there does not exist any quantifiable means of determining how close we are to an optimal solution. Without such a means, an A* search is not possible.

6.4.3.3 A Branch-and-Bound Search. The search space that we will explore can be visualized as a tree of nodes. The information contained at each node includes the output it defines, an associated minimal determining subset, an associated two-level representation and an associated cost. The evaluation function keeps track of the accumulated costs as we proceed down a path towards a solution. The search begins at the root node, with each successive node defining one of the outputs. Special care is taken to ensure that any given output along a path is defined only in terms of the inputs or other outputs previously defined along the same path. This constraint further reduces the possible search space. Beginning with the root node, each successive node that is explored represents a best possible choice in terms of its accumulated cost; the smaller the accumulated cost the better. Once all of the outputs have been defined, the nodes along this solution path contain all the information necessary to describe our least-cost solution. It may be the case that more than one least-cost solution exists.

6.4.3.4 A Scheme Implementation. The preceding search process was implemented in Scheme and included as part of the original optimization system. The documented source code can be found in Appendix B. It is worthwhile to point out some of the unique features of this search algorithm. It is called in Scheme using the format:

```
(SOLVE QUEUE MDS OUTPUTS MAXCOST)
```

QUEUE maintains a list of partial paths through the state-space. Each partial path has associated with it an accumulated cost. After every step in the search process, these paths are sorted such that the path with the current minimal cost appears first in the queue. MDS maintains a list of the original minimal determining subsets and their associated output,

cost and two-level circuit representation. OUTPUTS maintains a list of the desired outputs identified in the original specification. It turns out that a solution path is achieved if each of the nodes in that path contains one of the outputs specified in OUTPUTS. Finally, MAXCOST is a built-in safety factor that will terminate the search process if the cost exceeds a specified value.

To illustrate this search process consider a circuit specification defined by

```
[1]  (define ckt1 '("f = x' + y z"
                    "g = x y' + z'"
                    "h = x' + y' + z'") )

CKT1
```

We convert this specification to the desired normal standard form using the call procedure

```
[2]  (define parse-ckt1 (simplify (complement (parse ckt1))))

PARSE-CKT1
```

We then find the minimal determining subsets and their associated output, cost and two-level representation using the following procedure:

```
[3]  (define mds (out-arg-lists parse-ckt1 '(f g h)))
Minimal Determining Subsets:
F  ((G X) (H X) (X Y Z))
G  ((F Z) (H X Z) (X Y Z))
H  ((F X) (G X) (X Y Z))

MDS
```

If we take a look at *MDS* we see that it contains the necessary information:

```
[4]  mds
((F 2 (((G)) ((X))) G X) (F 2 (((H)) ((X))) H X)
 (F 4 (((X)) (Y Z)) X Y Z) (G 2 (((Z)) ((F))) F Z)
 (G 4 (((Z)) (X H)) H X Z) (G 4 (((Z)) (X (Y))) X Y Z)
 (H 2 (((F)) ((X))) F X) (H 2 (((X)) (G)) G X)
 (H 3 (((Z)) ((X)) ((Y))) X Y Z))
```

Finally, if we pass the MDS information to SOLVE, we can observe the action of the search process and the resulting solution.

```
[5]  (solve '((0 ())) mds '(f g h) 1000)

(0)
(3 (H 3 X Y Z))
(4 (G 4 X Y Z))
(4 (F 4 X Y Z))
(5 (F 2 H X) (H 3 X Y Z))
(6 (G 4 X Y Z) (H 2 G X))
(6 (F 2 G X) (G 4 X Y Z))
(6 (F 4 X Y Z) (H 2 F X))
(6 (F 4 X Y Z) (G 2 F Z))
(7 (F 4 X Y Z) (H 3 X Y Z))
(7 (G 4 H X Z) (H 3 X Y Z))
(7 (G 4 X Y Z) (H 3 X Y Z))

(7 (F 2 H X) (G 2 F Z) (H 3 X Y Z))
   F = H'+ X'
   G = Z'+ F'
   H = Z'+ X'+ Y'

DONE
```

The solution represents a recursive realization of the original specification. The total accumulated cost of 7 means that this circuit can be implemented with a minimum of 7 gate inputs. This compares with 11 gate inputs for the original, two-level specification.

6.4.3.5 Improving the Search Process. With a better understanding of the search process, we seek a way to improve its efficiency. While a variety of possibilities exist, we will concentrate on only one of these methods. It involves the calculation of cost for each of the minimal determining subsets.

Our concern here is not how the costs are calculated but rather when they are calculated. In our original system, the costs were calculated prior to the start of the search process. A careful examination of this search process reveals an important feature: it is not uncommon to arrive at a solution before all of the nodes in the search-space have been examined. If a particular node is not examined, it is useless and time-consuming to calculate its cost.

We propose that it may be more efficient to calculate the costs of each node as it is examined. There is one problem with this approach: a node may appear more than once in a given search-space. It is very important that we don't calculate the cost for a given node more than once. To overcome this problem, we utilized a procedure called MEMOIZE described in (94). When a node is evaluated, Scheme checks a table. If the cost associated with the given node has already been calculated, Scheme retrieves the necessary information. If it has not been calculated, then the cost procedure is called and the necessary information is produced and stored. The original search algorithm was modified for the purpose of evaluating this approach. The resulting code and documentation can be found in Appendix B.

6.4.4 The BORIS Design Tools Module. An obvious way to improve the efficiency of our global optimization system is to improve the efficiency of some of the fundamental procedures it uses. These procedures are located in the BORIS Design Tools file shown in Appendix B. Examples include Boolean addition, mu'tiplication, division, complementation, absorption, elimination, reduction, simplification and numerous other procedures. Since our system makes extensive use of these procedures at various stages throughout the optimization process, any speedup that can be achieved with these procedures will be a speedup of the optimization system as a whole. A great deal of progress has been achieved in this area by Army Captain James Kainec at the Air Force Institute

of Technology. A decision was made not to introduce any of these improved algorithms in our optimization system. It was felt that this would blur the distinction as to which modified processes were responsible for improving the efficiency. Once our results are well documented, these upgraded procedures will be incorporated into the BORIS Optimization System and will likely improve overall efficiency even further.

6.5 Finding an Optimal Solution

So far, most of the emphasis of our research and development has focused on improving the speed of our global optimization system. We would like to divert our focus slightly and address the problem of obtaining an optimal solution. While our technique of using minimal determining subsets to reduce the search space is effective, there are cases when a solution that represents a global optimum is eliminated by this process. With this in mind, we investigated ways to modify our existing system so that it would discover an optimal solution.

To uncover an optimal solution we need to find a way to expand the search space. One idea that was tested involved extending the list of minimal determining subsets to include some output-augmented subsets. These were minimal determining subsets that contained an additional output, other than the one that the MDS describes. Unfortunately, by expanding the search space we are likely to reduce the efficiency of our search. However, we would ultimately like the user to be able to decide whether or not the additional search time is worth the possibility of finding a better solution.

An auxiliary procedure that performs the preceding expansion process was developed in Scheme. It has the form

```
(MDS-EXPAND MDS Z OUTPUTS)
```

where `MDS` is a list of minimal determining subsets for a given output `Z` and a `OUTPUTS` is a list of the specified outputs. It returns an expanded list including the original minimal determining subsets along with the new modified ones. To illustrate how it functions, an example is shown below:

```
[1]  (mds-expand '((X1 X2 X3) (X2 X3 Z1)) 'Z3 '(Z1 Z2 Z3))

((X1 X2 X3 Z1) (X1 X2 X3 Z2) (X1 X2 X3) (X2 X3 Z1 Z2)
 (X2 X3 Z1))
```

By incorporating this process into the procedure that calculates the minimal determining subsets, we can effectively modify the optimization process. This new process is guaranteed to have a better chance at finding an optimal solution by virtue of its expanded search space. However, the decision on whether or not to use it depends on the speed at which we desire a result.

6.6 Summary

In this chapter we have scratched the surface of the myriad possibilities that exist for improving our global optimization system. We have analyzed the performance of various parts of the system and recommended viable solutions. In many cases these recommendations were researched, designed and then implemented in Scheme. It was not our intent in this chapter for all of the intricate workings of the design system to be discussed in detail; the complete listing of the source code is included in Appendix B and the software is available for those that are interested. The results of our efforts and the subsequent testing are summarized in the following chapter.

VII. Summary of Results

7.1 Introduction

The various modifications and upgrades to the global optimization system were thoroughly tested to verify both their efficiency and effectiveness. The computer that was used for this testing was an IBM-AT compatible with an 80286 microprocessor. To accomplish this task, a variety of sample circuit specifications were placed into a single file[1] and loaded into the BORIS optimization system. Each specification contains a particular combination of inputs, outputs, terms and operators, with some of the specifications known to be non-tabular. The specifications were carefully chosen to test our system under a wide range of conditions.

In this chapter we will highlight the results of this testing. Some of the key topics that will be addressed include:

- a typical optimization session,
- the performance of the tabular design module,
- improvements to the optimization system efficiency,
- the results of optimization, and
- other noteworthy observations.

A more detailed accounting of the results of each test can be found in Appendix A along with a brief description.

7.2 A Typical Optimization Session

We begin by loading Scheme. A special batch file that is recognized by Scheme (SCHEME.INI) will automatically load the necessary optimization files when Scheme is called. SCHEME.INI must be located in the same directory as the Scheme code in order for this to work properly. We begin the loading process by calling Scheme at the DOS prompt.

[1]The file is data.s and is shown in Appendix B

```
C: pcs
```

This initiates the loading of Scheme and subsequently all of the necessary optimization files. It will appear as follows:

```
 PC Scheme 3.03 07 June 88
(C) Copyright 1988 by Texas Instruments
          All Rights Reserved
TABULAR loaded
PARSE   loaded
MDS     loaded
COST    loaded
SEARCH  loaded
TOOLS   loaded
DATA    loaded
DESIGN  loaded

[PCS-DEBUG-MODE is OFF]
[1]
```

Once the system has been loaded and the Scheme prompt [1] appears, the optimization process can begin. The optimization of a specific circuit is initiated as follows

```
(DESIGN CIRCUIT OUTPUTS)
```

where CIRCUIT represents a circuit specification and OUTPUTS represents the designated outputs of the circuit. To illustrate the use of this system, we will run an example. An attempt will be made to optimize CKT2[2] whose outputs are F, G and H. The results are shown below:

[2]Defined in the data.s file shown in Appendix B

```
[1] ckt2
("f = x' + y z" "g = x y' + z'" "h = x' + y' + z'")

[2] (design ckt2 '(f g h))

* Parsing Equations and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
F'G H X Y'
F'G H X Z'
F G'H X'Z
F G H X'Z'
F G'H'X Y Z

Minimal Determining Subsets:
F  ((G X) (H X) (X Y Z))
G  ((F Z) (H X Z) (X Y Z))
H  ((F X) (G X) (X Y Z))

(0)
(3 (H 3 X Y Z))
(4 (G 4 X Y Z))
(4 (F 4 X Y Z))
(5 (F 2 H X) (H 3 X Y Z))
(6 (G 4 X Y Z) (H 2 G X))
(6 (F 2 G X) (G 4 X Y Z))
(6 (F 4 X Y Z) (H 2 F X))
(6 (F 4 X Y Z) (G 2 F Z))
(7 (F 4 X Y Z) (H 3 X Y Z))
(7 (G 4 H X Z) (H 3 X Y Z))
(7 (G 4 X Y Z) (H 3 X Y Z))

(7 (F 2 H X) (G 2 F Z) (H 3 X Y Z))
   F = H'+ X'
   G = F'+ Z'
   H = Y'+ X'+ Z'

DONE
```

We see that this recursive optimization process begins by parsing the system of equations into one equation in normal form. This equation is checked to see if it is tabular. In this example it passed the test. The terms of the resulting equation are then displayed

in a vertical fashion. Next, the minimal determining subsets for each output are displayed. The information concerning the cost for each minimal determining set has already been calculated at this point. Finally the branch-and-bound search process begins and proceeds until a solution is obtained. The final solution is a recursive realization of the original specification with a cost that has been reduced from 11 to 7.

7.3 The Performance of the Tabular Design Module

The tabular verification and design module functioned as expected throughout the testing process. It successfully identified all of the circuit specifications known to be non-tabular and passed all of those that were not. It was an efficient process that never consumed more than one to two percent of the total design time. When a non-tabular specification was encountered, it transformed the specification into an acceptable tabular form. Two examples of optimizing a non-tabular specification (nontab1 and nontab2) can be found in Appendix A.

7.4 Improvements to the Optimization System Efficiency

7.4.1 Preliminary Testing. Prior to undertaking any new modifications, the original BORIS optimization system was thoroughly analyzed. Its performance was divided into the three distinct parts mentioned in Section 6.2: parsing and reduction of a system of Boolean equations, calculating the minimal determining subsets and searching for the best solution.

1. Parsing and reduction of a system of Boolean equations.
2. Calculation of the minimal determining subsets and their associated costs.
3. Searching for an optimal or near-optimal solution.

Using a variety of sample specifications, the total run-times of each of these parts was measured. The results have been compiled and can be found in Table 7.1. While this sampling represents only a fraction of the possible types of specifications that may be encountered, it is large enough to draw some general conclusions. It is obvious that as the number of

Name	Inputs/ Outputs	Parse (sec)	MDS (sec)	Search (sec)	Total (sec)
ckt1	2/2	1.25	1.34	0.64	3.23
ckt2	3/3	4.70	9.42	1.90	16.02
ckt3	3/3	7.83	12.12	1.01	20.96
ckt4	4/4	11.14	65.24	4.42	80.80
ckt5	5/5	90.15	808.15	20.99	919.29
wsu-ckt	4/3	10.97	20.46	2.07	33.50
example	4/3	24.82	22.44	3.12	50.38
sample	3/3	28.14	17.05	3.36	48.55
ex-951	3/3	6.84	11.96	2.51	21.31
bcdto3	4/4	78.53	104.13	25.27	207.87

Table 7.1. Efficiency of Original BORIS Optimization System

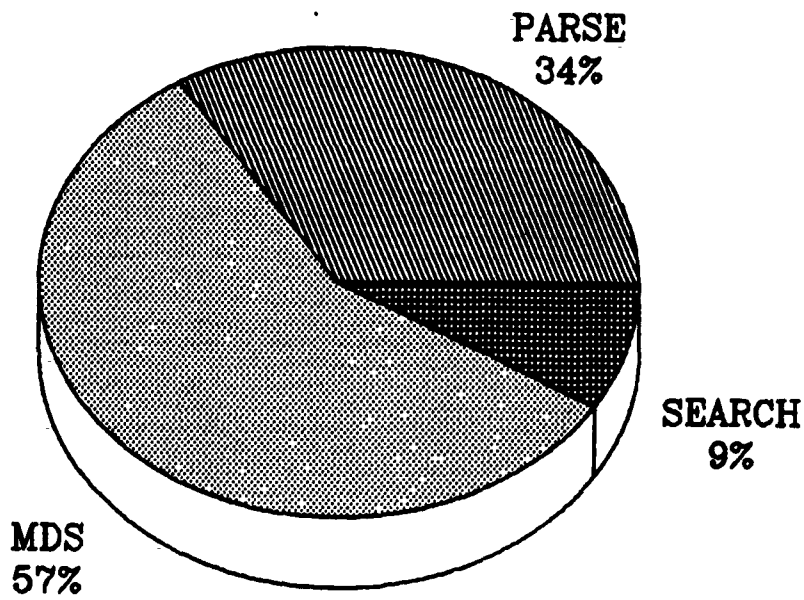


Figure 7.1. The Original Run-Time Distribution

Name	Original Parse (sec)	Updated Parse (sec)	Run-Time Reduction (%)
ckt1	1.25	0.50	60
ckt2	4.70	1.90	60
ckt3	7.83	3.12	60
ckt4	11.14	4.43	60
ckt5	90.15	10.29	89
wsu-ckt	10.97	5.12	53
example	24.82	3.37	86
sample	28.14	14.30	51
ex-951	6.84	2.34	34
bcdto3	78.53	8.27	89
Average Improvement - 64			

Table 7.2. Speed of Original Parser versus Updated Parser

inputs and outputs increases, the run-times increase correspondingly, at what appears to be an exponential rate. This emphasizes the system's inherent problems with computational complexity. Another important observation is the relative proportion of run-time spent by each part of the system. To illustrate this, the pie chart shown in Figure 7.1 was constructed. It represents the amount of run-time spent on each part of the optimization system when averaged out over all of the test runs shown in Table 7.1. It shows us that the bulk of the processing time was consumed in calculating the minimal determining subsets and associated costs (57%). This was followed by the parsing system (34%) and finally the search process (9%). As we mentioned in Chapter 6, these results contributed significantly to the direction our research would take. Our primary emphasis was placed on improving the algorithms associated with parsing and finding minimal determining subsets.

7.4.2 The Modified Parsing System. The interesting discovery concerining the parser is that the parser itself is rather efficient. It is the process of trying to reduce the parsed equation to its simplest form that consumes a majority of the time. The only modification that was made was to postpone any simplification attempts until after the complement was taken. After the equation was in normal form, we applied the algorithm SIMPLIFY which quickly reduces the equation to an adequately minimized, though not necessarily optimal, form. The results of this upgrade are illustrated in Table 7.2. This

Name	MDS1 (sec)	MDS2 (sec)	MDS3 (sec)	MDS4 (sec)
ckt1	1.34	0.73	0.65	0.58
ckt2	9.42	9.06	8.79	4.75
ckt3	12.12	8.28	7.76	5.66
ckt4	76.73	36.26	33.23	25.18
ckt5	808.15	589.13	513.27	121.79
wsu-ckt	20.46	13.41	13.01	10.29
example	22.44	15.86	14.48	14.31
sample	17.05	14.03	12.73	11.89
ex-951	11.96	5.91	5.54	5.06
bcdto3	104.13	56.64	44.44	44.27

Table 7.3. A Comparison of MDS Algorithm Run-Times

data indicates an average 64 percent reduction in the run-time of the parser system. This translates into a significant improvement in the speed of the optimization system in general.

7.4.3 Comparing the MDS Algorithms. Our goal was to find the most efficient technique for calculating minimal determining subsets. A variety of approaches were investigated, with four of them actually prototyped in Scheme. All of these techniques produced identical results; however, their run-times varied quite dramatica . four algorithms were tested on the sample specifications and the results were c' . . n Table 7.3. The first approach (MDS1) uses the Redundancy Elimination Techn, that was part of the original design system. The second approach (MDS2) uses the Opposing Literals Technique with a multiplication/absorption process. The third approach (MDS3) uses the Opposing Literals Technique with a expansion process. The fourth and final approach (MDS4) uses the Opposing Literals Technique with an expansion process that includes intelligent selection.

It is obvious from the results that the fourth approach (MDS4) is by far the fastest. On the average, it was able to calculate the minimal determining subsets over twice as fast and in some cases up to six times as fast as the original approach (MDS1). The speedup using MDS4 seemed become more pronounced as the number of inputs and outputs in the specifications increased. This is a very encouraging feature of this approach. These results led to the integration of the MDS4 module into the final version of our optimization

system.

7.4.4 Evaluating the Modified Search Algorithm. The search algorithm was modified such that the cost of each node in the search tree was calculated only when it was examined. If the solution was found before a given node was examined, the cost for that node would never have to be calculated. This modified search algorithm was analyzed using our set of sample specifications. The resulting analysis indicated that no significant speedup was achieved. There were a few examples that were optimized slightly faster using this approach, but for most them the time was about the same or a little slower. To understand the reasons for these results we need to understand a little more about the process itself.

The process does introduce some additional overhead. Every time a node is examined, we have to check a table to see if the cost has been previously calculated. If it has not, then we need to calculate the cost at that point. To calculate the cost for a given output, we need to know the range of that output. The ranges for each of the outputs are calculated at the beginning and stored in a table. Thus every time a cost is calculated, the proper range needs to be selected from the table. This also introduces additional overhead.

Another reason the results are not what we might expect is that, given the relatively small size of our sample specifications, almost all of the nodes were examined any way before a solution was obtained. Thus, in these examples, the advantages of calculating the costs later was overridden by the additional overhead that was introduced. It is only when the circuit specifications contain a large number of inputs and outputs that this technique has a chance at improving the speed and efficiency of the process. For a large specification, many cost calculations may be avoided and the overhead would be amortized over a larger search space.

7.4.5 Summary of Efficiency Upgrades. The improved parsing and MDS algorithms have significantly enhanced the overall performance of the circuit optimization system. Table 7.4 compares the total run-times of our original optimization system with our upgraded one. As you can see, using the upgraded system resulted in a significant im-

Name	Original System (sec)	Upgraded System (sec)	Run-Time Reduction (%)
ckt1	3.23	2.00	38
ckt2	16.02	9.51	41
ckt3	20.96	10.84	48
ckt4	80.80	34.02	58
ckt5	919.29	157.23	83
wsu-ckt	33.50	17.94	46
example	50.38	22.40	56
sample	48.55	31.67	35
ex-951	21.31	11.85	44
bcdto3	207.87	85.19	59
Average Improvement - 51			

Table 7.4. Speed of Original System versus Upgraded System

provement in the run-time for every sample specification tested. This reduction averaged about 51 percent with up to an 83 percent reduction in the case of the ckt5 specification.

7.5 The Results of Optimization

While our focus has been on improving the speed and efficiency of the BORIS optimization system, it is worthwhile to point out the cost reductions that this system was able to achieve. The gate-input costs of the specified circuit before and after optimization are shown in Table 7.5. We can see from these results, that 50 percent reductions in the gate-input cost are quite possible.

7.6 Improving the Optimization Results

An alternative optimization technique that expands the potential search space was evaluated. The procedure MDS-EXPAND was incorporated into the existing optimization system. It expands the collection of minimal determining subsets to include some non-minimal ones. Using this special technique, described in Chapter 6, we hoped to achieve a solution that is closer to optimal. Using as a test case the specification called "sample",

Name	Original Cost	Optimized Cost	Reduction (%)
ckt1	4	3	25
ckt2	11	7	36
ckt3	17	10	41
ckt4	11	7	36
ckt5	15	13	13
wsu-ckt	21	10	52
example	33	27	18
sample	44	22	50
ex-951	16	11	31
bcdto3	24	21	12
Average Reduction - 31			

Table 7.5. Gate-Input Cost Before and After Optimization

the original BORIS design system achieved the result

$$\begin{aligned} z_1 &= x_1x_3' + x_1x_2' + x_1'x_2x_3 \\ z_2 &= z_1z_3 \\ z_2 &= x_3z_1 + x_2z_1 + x_2'x_3'z_1' \end{aligned}$$

at a cost of 22. By using our modified technique to expand the search space, we were able to obtain the result

$$\begin{aligned} z_1 &= z_2 + x_1x_3' \\ z_2 &= x_1x_2x_3' + x_1'x_2x_3 + x_1x_2'x_3 \\ z_3 &= z_2 + x_2'x_3'z_1' \end{aligned}$$

at a cost of 21. In addition, eleven other circuits that meet the original specification at a cost of 21 were uncovered. Unfortunately, the system took over twice as long to arrive at this solution. Thus we have showed that this technique can effectively uncover better solutions at the expense of the system's efficiency. The listing of this trial run can be found along with the other results in Appendix A.

7.7 Other Noteworthy Observations

As often happens in research, a careless mistake led to an interesting discovery. While testing the BORIS optimization system, a parsed specification in the form $F' = 1$ was accidentally passed on to the system instead of a specification in the form $F = 1$. In other words, the specification fed to the system was the obverse of what it should be. This led to the following interesting results:

- The minimal determining subsets were identical to what they should be.
- The formula describing each output in the solution was simply the complement of what it should be.
- The search involved less than half the number of steps and took less than half the time.

Subsequent testing found that designing with the *obverse specification* (specification of the form $F' = 1$) can result in a search that is either longer or shorter than the original. The reason that the search changes when designing with the obverse specification is that the costs assigned to each minimal determining subset are different. This discovery opens the door to a variety of new possibilities which could be aimed at improving the efficiency of the search process. We could use a parallel system to concurrently optimize the original specification and the obverse specification; the optimization process would terminate with whichever process completed first. The results of this testing are listed in Appendix A.

7.8 Summary

We have investigated just a few of numerous possible upgrades to the BORIS optimization system. These upgrades have resulted in dramatic improvements in efficiency. Because we have been successful in reducing the time consumed in parsing the specification and calculating the minimal determining subsets, a larger percentage of the total run-time is now consumed by the search process. This is illustrated in the pie chart shown in Figure 7.2. Since the search process now consumes a significant portion of the overall run-time, future efforts can concentrate on improving it.

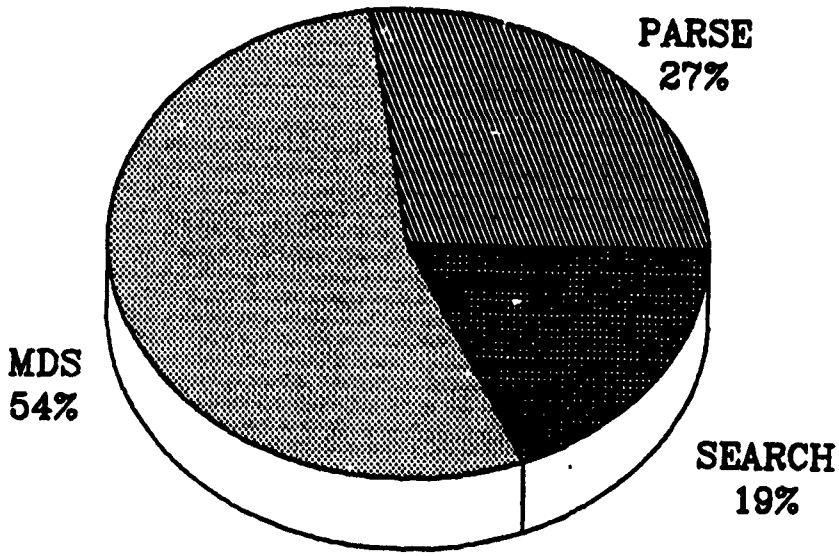


Figure 7.2. The Run-Time Distribution For Our Upgraded System

While significant improvements have been made to the optimization process, practical circuit specifications with 20 to 30 or more inputs/outputs are still beyond our system's capacity. The system does have an inherent problem with complexity. Consequently it takes an inordinate amount of time to optimize large specifications. Scheme typically runs out of internal memory or stack space long before a solution is obtained. Despite these problems, the system can effectively handle circuits that are larger than ever before. It definately is a viable approach to circuit optimization that holds a great deal of promise for the future.

VIII. Conclusions and Recommendations

8.1 Summary

A great deal of emphasis in this research effort was placed on establishing a foundation of knowledge on the topic of logic optimization techniques and principles. A brief historical perspective on the development of optimization systems was presented along with an analysis of the current, state-of-the-art techniques. Given this background, we were able to identify the basic problem with current optimization systems: their inability to efficiently achieve near-optimal, multi-level solutions. This typically results from their failure to take advantage of don't-care conditions and redundancies that exist in a circuit specification. Global approaches aimed at solving these problems have been around for a long time. However most of these approaches incorporate algebraic techniques that produce faster, though seldom optimal solutions. A better approach is the use of Boolean reasoning techniques. However, this approach has been largely ignored because of its innate complexity and a lack of understanding that such algorithms exist.

New technologies and requirements are forcing designers to pack more and more logic onto smaller chips. Consequently, global optimization systems are beginning to receive considerable attention. Some multi-level optimization systems are beginning to incorporate some Boolean reasoning techniques along with their algebraic ones. Better techniques, coupled with faster computers, have begun to make global optimization a profitable venture. However, a significant amount of work still needs to be done to make current global techniques more efficient and effective. The need for better optimizations systems will become increasingly important as the application-specific integrated circuit market (ASIC) continues to grow (18).

The recursive optimization system presented in this thesis was developed to investigate a new approach to global optimization. It is a relatively simple system that overcomes some of the complexity and inefficiencies that are associated with Boolean-based approaches. To take advantage of redundancies and don't-care conditions, the equations that specify the circuit were reduced to a single equation. A dependency analysis on the variables was performed to reduce the potential search space and improve the system ef-

ficiency. A branch-and-bound search was then used to find an optimal solution in the search space, based on the given costs. For specifications with up to 10 variables, the recursive optimization system was effectiv in finding a near-optimal solution. However, the efficiency of the system still lagged behind expectations and consequently became one of the main focal points of this research effort.

8.2 Specific Accomplishments

8.2.1 The System Efficiency Was Improved. When the recursive optimization system was analyzed, it was found that the bulk of the run-time was consumed parsing the equations and calculating the minimal determining subsets. Subsequent modifications to these processes achieved significant improvements. The run-time consumed by the parsing process was reduced, on the average, by 64 percent. This was achieved by modifying the way the parser simplified an expression and at what point it was simplified. By modifying the process that calculates the minimal determining subsets to an opposed literals approach, the resulting run-times were reduced by an average of 54 percent. This new approach incorporated an expansion process that intelligently selected the variable to expand on. The integration of these upgraded modules into the original optimization system resulted in run-times that were reduced by up to 83 percent.

8.2.2 A Tabular Design Filter Was Constructed. A tabular design filter was successfully integrated into the system. It provided an effective means of ensuring that non-tabular specifications are not passed on to the optimization system. Those non-tabular specifications that were encountered were transformed into acceptable tabular forms. The tabular module gives the designer some freedom to describe a system's desired characteristics in the most general terms. It eliminates the need for one to ensure that the system he describes is tabular and can be represented by a truth table.

8.2.3 Further Optimization Was Achieved. The recursive optimization system consistantly achieved significant cost reductions when compared to a non-recursive, two-level implementation. Despite this, an attempt was made to improve the original optimization system even further. It involved expanding the search space by introducing

output-augmented subsets. Although the process of injecting additional outputs into the minimal determining subsets was rather primitive, it achieved the desired results. In one example, the gate-input cost of an optimized sample circuit was reduced from 22 to 2_. Because of the reduced efficiency of this process, it was not integrated into the upgraded optimization system. It does, however, show that techniques do exist that can improve the effectiveness of the recursive optimization process by expanding the search space.

8.3 Recommendations

As a result of this research effort a number of interesting questions were raised. Some ideas that may warrent further investigation include the following:

Improved Search Techniques Search plays an extremely important role in the recursive optimization process. The current search technique could be improved through the use of "open" and "closed" lists instead of the current queue-based approach. In addition, alternative search techniques should be explored. This research could even evaluate the use of non-deterministic search processes such as the use of *genetic algorithms* (47). Genetic algorithms are an adaptive search process that has achieved noteworthy success when applied some difficult, combinatorial, NP-complete problems.

Calculating MDS Cost Currently the costs of minimal determining subsets are calculated by finding their optimal two-level representation and then counting the gate-inputs. This is a very time-consuming process. Alternative approaches may make use the size of the minimal determining subsets themselves, or incorporate other factors.

Designing With The Obverse Specification We discovered that a system could be designed using the obverse specification instead of the specification itself. Further research could explore ways to utilize this finding to improve the efficiency of the optimization process.

Improve Tabular Design Filter Currently, when a non-tabular specification is encountered, it is transformed into an arbitrary tabular form. Most non-tabular specifications can be decomposed into a collection of tabular specifications, each sufficient to describe the desired behavioral characteristics of the original system. One could investigate

heuristics for selecting, from this collection of tabular specifications, the one that would result in a relatively good solution.

Removing Assumptions By removing some of our original assumptions, one could investigate a variety of factors; these factors might include controlling the allowable time-delay introduced by the optimization process, handling sequential circuits and controlling fan-in limitations.

Optimize For A Target Technology The recursive optimization process represents only a portion of a complete design procedure. It begins with a specification that is represented by a system of Boolean equations in SOP form. It produces an optimized system of equations that are also in SOP form. Today, very few circuits are actually implemented in AND-OR logic. Consequently, our result must be transformed into the applicable target technology. Unfortunately this transformation usually results in a system that is no longer optimal. Is there some way, using global techniques, to optimize a system of equations directly into a target technology? This would be an interesting research project.

Develop A VHDL To Scheme Parser This would provide an effective front-end to the recursive optimization system and facilitate tests using realistic specifications. It should be capable of accepting a specification defined in VHDL and transforming it into a system of list-based Boolean equations that can be interpreted by Scheme.

Improve User Interface BORIS is more than just a recursive optimization system; it is a library of Boolean reasoning tools. It includes everything from complementation, absorption, elimination and Boolean arithmetic procedures to algorithms that find the Blake canonical form or perform a variety of other simplification tasks. With the proper user interface, it could become a valuable learning or design tool for anyone working with Boolean algebra or involved in digital design.

8.4 Conclusion

More important than any of the improvements we achieved was the knowledge that we gained. One of the primary objectives of our effort was to lay the groundwork for contin-

ued research in this area. The results that were obtained using this recursive optimization approach should stimulate some interest. Even though this system is currently not capable of handling specifications with a large number of variables, this does not present an insurmountable obstacle. Nothing we have seen seems to indicate that further, even more dramatic, improvements in speed and efficiency cannot be obtained. No claim is made that this approach is the answer to all optimization problems. It is, however, a step in the right direction. It is a viable technique for the optimization of digital circuits.

Appendix A. Selected Listings of Results

A.1 Recursive Optimization of CKT1

The specification defined by ckt1 is a simple two-input, two-ouput system. The results of optimization are listed below:

```
[2] ckt1

("f = a' + b'" "g = a b")

[3] (design ckt1 '(f g))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
B'F G'
A'F G'
A B F'G

Minimal Determining Subsets:
F  ((G) (A B))
G  ((F) (A B))

(0)
(2 (F 2 A B))
(2 (G 2 A B))

(3 (F 1 G) (G 2 A B))
   F = G'
   G = A B

(3 (F 2 A B) (G 1 F))
DONE
```

A.2 Recursive Optimization of CKT2

The specification defined by CKT2 is a simple three-input, three-output system. The results of optimization are listed below:

```
[4] ckt2
("f = x' + y z" "g = x y' + z'" "h = x' + y' + z'")
[5] (design ckt2 '(f g h))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
F G'H X'Z
F G'H'X Y Z
F'G H X Y'
F'G H X Z'
F G H X'Z'

Minimal Determining Subsets:
F  ((G X) (H X) (X Y Z))
G  ((F Z) (H X Z) (X Y Z))
H  ((F X) (G X) (X Y Z))

(0)
(3 (H 3 X Y Z))
(4 (G 4 X Y Z))
(4 (F 4 X Y Z))
(5 (F 2 H X) (H 3 X Y Z))
(6 (G 4 X Y Z) (H 2 G X))
(6 (F 2 G X) (G 4 X Y Z))
(6 (F 4 X Y Z) (H 2 F X))
(6 (F 4 X Y Z) (G 2 F Z))
(7 (F 4 X Y Z) (H 3 X Y Z))
(7 (G 4 H X Z) (H 3 X Y Z))
(7 (G 4 X Y Z) (H 3 X Y Z))

(7 (F 2 H X) (G 2 F Z) (H 3 X Y Z))
   F = H'+ X'
   G = Z'+ F'
   H = Z'+ X'+ Y'
DONE
```

A.3 Recursive Optimization of CKT3

The specification defined by CKT3 is a another three-input, three-output system. The results of optimization are listed below:

```
[6] ckt3

("z1 = a b' + a' b + b c" "z2 = a' b' + a b" "z3 = b' + c'")

[7] (design ckt3 '(z1 z2 z3))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
A'B C Z1 Z2'Z3'
A B C Z1 Z2 Z3'
A B C'Z1'Z2 Z3
A'B'Z1'Z2 Z3
A'B C'Z1 Z2'Z3
A B'Z1 Z2'Z3

Minimal Determining Subsets:
Z1  ((Z2 Z3) (A B C) (A B Z3) (A C Z2) (B C Z2))
Z2  ((A B) (A Z1 Z3))
Z3  ((B C) (A C Z2))

(0)
(2 (Z3 2 B C))
(6 (Z2 6 A B))
(8 (Z2 6 A B) (Z3 2 B C))
(9 (Z1 7 A B Z3) (Z3 2 B C))
(9 (Z1 9 A B C))
(10 (Z1 4 B C Z2) (Z2 6 A B))
(10 (Z1 4 A C Z2) (Z2 6 A B))

(10 (Z1 2 Z2 Z3) (Z2 6 A B) (Z3 2 B C))
   Z1 = Z3'+ Z2'
   Z2 = A B + A'B'
   Z3 = C'+ B'

DONE
```

A.4 Recursive Optimization of CKT4

The specification defined by CKT4 is a four-input, four-output system. The results of optimization are listed below:

```
[8] ckt4

("w = a p + q" "x = a p" "y = p + a' r" "z = q")

[9] (design ckt4 '(w x y z))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
P'Q'R'W'X'Y'Z'
A P'Q'W'X'Y'Z'
A'P Q'W'X'Y Z'
A'Q'R W'X'Y Z'
P'Q R'W X'Y'Z
A P'Q W X'Y'Z
A'P Q W X'Y Z
A'Q R W X'Y Z
A P Q'W X Y Z'
A P Q W X Y Z

Minimal Determining Subsets:
W  ((Q X) (X Z) (A P Q) (A P Z) (A Q Y) (A Y Z))
X  ((A P) (A Y))
Y  ((A P R))
Z  ((Q))

(0)
(1 (Z 1 Q))
(2 (X 2 A P))
(3 (X 2 A P) (Z 1 Q))
(4 (Y 4 A P R))
(4 (W 4 A P Q))
(4 (W 2 Q X) (X 2 A P))
(5 (Y 4 A P R) (Z 1 Q))
(5 (W 4 A P Q) (Z 1 Q))
(5 (W 4 A P Z) (Z 1 Q))
(5 (W 2 X Z) (X 2 A P) (Z 1 Q))
```

```
(5 (W 2 Q X) (X 2 A P) (Z 1 Q))
(6 (X 2 A P) (Y 4 A P R))
(6 (W 4 A P Q) (X 2 A P))
(6 (X 2 A Y) (Y 4 A P R))
(7 (X 2 A P) (Y 4 A P R) (Z 1 Q))
(7 (W 4 A P Z) (X 2 A P) (Z 1 Q))
(7 (W 4 A P Q) (X 2 A P) (Z 1 Q))
(7 (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(8 (W 2 Q X) (X 2 A P) (Y 4 A P R))
(8 (W 4 A Q Y) (Y 4 A P R))
(8 (W 4 A P Q) (Y 4 A P R))
(8 (W 2 Q X) (X 2 A Y) (Y 4 A P R))

(9 (W 2 Q X) (X 2 A P) (Y 4 A P R) (Z 1 Q))
   W = Q + X
   X = A P
   Y = P + A'R
   Z = Q

(9 (W 2 X Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(9 (W 2 X Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(9 (W 2 Q X) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
DONE
```

A.5 Recursive Optimization of CKT5

The specification defined by CKT5 is a five-input, five-output system. With ten variables, this circuit thoroughly exercises our program. The results of this optimization are shown below:

```
[10] ckt5

("v = q' r + t" "w = a p + q" "x = a p" "y = p + a' r" "z = q")

[11] (design ckt5 '(v w x y z))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
P'Q'R'T'V'W'X'Y'Z'
P'Q'R'T V W'X'Y'Z'
A P'Q'T V W'X'Y'Z'
A P'Q'R V W'X'Y'Z'
A'P'Q'R V W'X'Y Z'
A'P Q'R'T'V'W'X'Y Z'
A'Q'R T'V W'X'Y Z'
A'P Q'T V W'X'Y Z'
P'Q R'T'V'W X'Y'Z
A P'Q T'V'W X'Y'Z
P'Q R'T V W X'Y'Z
A P'Q T V W X'Y'Z
A'P Q T'V'W X'Y Z
A'Q R T'V'W X'Y Z
A'P Q T V W X'Y Z
A'Q R T V W X'Y Z
A P Q'R'T'V'W X Y Z'
A P Q'T V W X Y Z'
A P Q'R V W X Y Z'
A P Q T'V'W X Y Z
A P Q T V W X Y Z
```

```
Minimal Determining Subsets:
V  ((Q R T) (R T Z))
W  ((Q X) (X Z) (A P Q) (A P Z) (A Q Y) (A Y Z))
X  ((A P) (A Y))
Y  ((A P R))
Z  ((Q))

(0)
(1 (Z 1 Q))
(2 (X 2 A P))
(3 (X 2 A P) (Z 1 Q))
(4 (Y 4 A P R))
(4 (V 4 Q R T))
(4 (W 4 A P Q))
(4 (W 2 Q X) (X 2 A P))
(5 (Y 4 A P R) (Z 1 Q))
(5 (V 4 Q R T) (Z 1 Q))
(5 (V 4 R T Z) (Z 1 Q))
(5 (W 4 A P Q) (Z 1 Q))
(5 (W 4 A P Z) (Z 1 Q))
(5 (W 2 X Z) (X 2 A P) (Z 1 Q))
(5 (W 2 Q X) (X 2 A P) (Z 1 Q))
(6 (X 2 A P) (Y 4 A P R))
(6 (W 4 A P Q) (X 2 A P))
(6 (V 4 Q R T) (X 2 A P))
(6 (X 2 A Y) (Y 4 A P R))
(7 (X 2 A P) (Y 4 A P R) (Z 1 Q))
(7 (W 4 A P Z) (X 2 A P) (Z 1 Q))
(7 (W 4 A P Q) (X 2 A P) (Z 1 Q))
(7 (V 4 Q R T) (X 2 A P) (Z 1 Q))
(7 (V 4 R T Z) (X 2 A P) (Z 1 Q))
(7 (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(8 (V 4 Q R T) (W 2 Q X) (X 2 A P))
(8 (W 2 Q X) (X 2 A P) (Y 4 A P R))
(8 (W 4 A Q Y) (Y 4 A P R))
(8 (W 4 A P Q) (Y 4 A P R))
(8 (V 4 Q R T) (Y 4 A P R))
(8 (V 4 Q R T) (W 4 A P Q))
(8 (W 2 Q X) (X 2 A Y) (Y 4 A P R))
(9 (V 4 Q R T) (W 2 Q X) (X 2 A P) (Z 1 Q))
(9 (V 4 R T Z) (W 2 Q X) (X 2 A P) (Z 1 Q))
(9 (W 2 Q X) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(9 (V 4 Q R T) (W 2 X Z) (X 2 A P) (Z 1 Q))
(9 (V 4 R T Z) (W 2 X Z) (X 2 A P) (Z 1 Q))
```

```
(9 (W 2 X Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(9 (V 4 R T Z) (W 4 A P Q) (Z 1 Q))
(9 (V 4 R T Z) (W 4 A P Z) (Z 1 Q))
(9 (W 4 A Y Z) (Y 4 A P R) (Z 1 Q))
(9 (W 4 A Q Y) (Y 4 A P R) (Z 1 Q))
(9 (W 4 A P Z) (Y 4 A P R) (Z 1 Q))
(9 (W 4 A P Q) (Y 4 A P R) (Z 1 Q))
(9 (V 4 R T Z) (Y 4 A P R) (Z 1 Q))
(9 (V 4 Q R T) (Y 4 A P R) (Z 1 Q))
(9 (V 4 Q R T) (W 4 A P Q) (Z 1 Q))
(9 (V 4 Q R T) (W 4 A P Z) (Z 1 Q))
(9 (W 2 X Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(9 (W 2 Q X) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(10 (V 4 Q R T) (X 2 A Y) (Y 4 A P R))
(10 (W 4 A P Q) (X 2 A Y) (Y 4 A P R))
(10 (W 4 A Q Y) (X 2 A Y) (Y 4 A P R))
(10 (V 4 Q R T) (W 4 A P Q) (X 2 A P))
(10 (V 4 Q R T) (X 2 A P) (Y 4 A P R))
(10 (W 4 A P Q) (X 2 A P) (Y 4 A P R))
(10 (W 4 A Q Y) (X 2 A P) (Y 4 A P R))
(11 (V 4 R T Z) (W 4 A P Z) (X 2 A P) (Z 1 Q))
(11 (V 4 Q R T) (W 4 A P Z) (X 2 A P) (Z 1 Q))
(11 (V 4 Q R T) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (V 4 R T Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A P Q) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A P Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A Q Y) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A Y Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(11 (V 4 R T Z) (W 4 A P Q) (X 2 A P) (Z 1 Q))
(11 (V 4 Q R T) (W 4 A P Q) (X 2 A P) (Z 1 Q))
(11 (W 4 A Y Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A Q Y) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A P Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(11 (W 4 A P Q) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(11 (V 4 Q R T) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(11 (V 4 R T Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(12 (V 4 Q R T) (W 2 Q X) (X 2 A P) (Y 4 A P R))
(12 (V 4 Q R T) (W 4 A Q Y) (Y 4 A P R))
(12 (V 4 Q R T) (W 4 A P Q) (Y 4 A P R))
(12 (V 4 Q R T) (W 2 Q X) (X 2 A Y) (Y 4 A P R))
```

```
(13 (V 4 Q R T) (W 2 Q X) (X 2 A P) (Y 4 A P R) (Z 1 Q))
   V = T + Q'R
   W = Q + X
   X = A P
   Y = P + A'R
   Z = Q

(13 (V 4 R T Z) (W 2 Q X) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(13 (V 4 Q R T) (W 2 X Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(13 (V 4 R T Z) (W 2 X Z) (X 2 A P) (Y 4 A P R) (Z 1 Q))
(13 (V 4 R T Z) (W 2 X Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(13 (V 4 Q R T) (W 2 X Z) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(13 (V 4 R T Z) (W 2 Q X) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
(13 (V 4 Q R T) (W 2 Q X) (X 2 A Y) (Y 4 A P R) (Z 1 Q))
DONE
```

A.6 Recursive Optimization of WSU-CKT

The specification defined by WSU-CKT is a three-input, four-output system. WSU-CKT was one of the examples used in the original optimization system. The results of optimization are shown below:

```
[15] wsu-ckt
("z1 = a b + a c' + b d' + c' d'" "z2 = b' c + a' c d"
 "z3 = b' c")
[16] (design wsu-ckt '(z1 z2 z3))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
A'C'D Z1'Z2'Z3'
A B Z1 Z2'Z3'
A C'Z1 Z2'Z3'
C'D'Z1 Z2'Z3'
B D'Z1 Z2'Z3'
B'C Z1'Z2 Z3
A'B C D Z1'Z2 Z3'

Minimal Determining Subsets:
Z1  ((A D Z2) (A D Z3) (A B C D))
Z2  ((C Z1) (A B C D) (A C D Z3))
Z3  ((B C) (B Z2))

(0)
(2 (Z3 2 B C))
(7 (Z2 5 A C D Z3) (Z3 2 B C))
(7 (Z2 7 A B C D))
(8 (Z1 6 A D Z3) (Z3 2 B C))
(9 (Z2 7 A B C D) (Z3 2 B C))
(9 (Z2 7 A B C D) (Z3 2 B Z2))

(10 (Z1 6 A D Z3) (Z2 2 C Z1) (Z3 2 B C))
   Z1 = D'Z3'+ A Z3'
   Z2 = C Z1'
   Z3 = B'C

DONE
```

A.7 Recursive Optimization of EXAMPLE

The specification defined by EXAMPLE is a three-input, three-output system. It was used in Example 7.7.1 of (22). It is unique in that the output u is already defined in terms of s, another output. Even more interesting is the fact that in the final result, u depends only on the inputs and d and s depend on u. The optimization of EXAMPLE is shown below:

```
[17] example

("d = a b + a c + b c" "s = a ! b ! c" "u = a b s' + a' b' s")

[18] (design example '(d s u))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
A'B'C'D'S'U'
A'B C'D'S U'
A B'C'D'S U'
A B'C D S'U'
A'B C D S'U'
A B C D S U'
A B C'D S'U
A'B'C D'S U

Minimal Determining Subsets:
D  ((C U) (A B C) (A B S) (A C S) (B C S))
S  ((A B C) (A B D U))
U  ((C D) (A B C) (A B S) (A C S) (B C S))

(0)
(8 (U 8 A B C))
(9 (D 9 A B C))
(14 (D 6 C U) (U 8 A B C))
(15 (D 9 A B C) (U 6 C D))
(16 (S 16 A B C))
(17 (D 9 A B C) (U 8 A B C))
(24 (S 16 A B C) (U 8 A B C))
(24 (S 16 A B C) (U 8 B C S))
```

```
(24 (S 16 A B C) (U 8 A C S))
(24 (S 16 A B C) (U 8 A B S))
(25 (D 9 B C S) (S 16 A B C))
(25 (D 9 A C S) (S 16 A B C))
(25 (D 9 A B S) (S 16 A B C))
(25 (D 9 A B C) (S 16 A B C))

(27 (D 6 C U) (S 13 A B D U) (U 8 A B C))
   D = C'U + C U'
   S = A D'+ B D'+ B'U + A B U'
   U = A B C'+ A'B'C

DONE
```

A.8 Recursive Optimization of SAMPLE

The specification defined by SAMPLE is a three-input, three-output system. It was used in example 9.6.3 of (22). It contains a larger number of terms in the specification than the previous examples. The results of optimization are shown below:

```
[19] sample
("z1 = x1' x2 x3 + x1 x2' x3' + x1 x2' x3 + x1 x2 x3'"
 "z2 = x1' x2 x3 + x1 x2' x3 + x1 x2 x3'"
 "z3 = x1' x2' x3' + x1' x2 x3 + x1 x2' x3 + x1 x2 x3'")
[20] (design sample '(z1 z2 z3))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
X1'X2'X3'Z1'Z2'Z3
X1 X2'X3'Z1 Z2'Z3'
X1'X2 X3'Z1'Z2'Z3'
X1'X2'X3 Z1'Z2'Z3'
X1 X2 X3 Z1'Z2'Z3'
X1 X2 X3'Z1 Z2 Z3
X1'X2 X3 Z1 Z2 Z3
X1 X2'X3 Z1 Z2 Z3

Minimal Determining Subsets:
Z1  ((X1 X2 X3) (X1 X2 Z2) (X1 X2 Z3) (X1 X3 Z2) (X1 X3 Z3)
     (X2 X3 Z3))
Z2  ((Z1 Z3) (X1 X2 X3) (X1 X2 Z3) (X1 X3 Z3) (X2 X3 Z1)
     (X2 X3 Z3))
Z3  ((X1 X2 X3) (X2 X3 Z1))

(0)
(10 (Z1 10 X1 X2 X3))
(12 (Z2 12 X1 X2 X3))
(16 (Z1 4 X1 X2 Z2) (Z2 12 X1 X2 X3))
(16 (Z1 4 X1 X3 Z2) (Z2 12 X1 X2 X3))
(16 (Z3 16 X1 X2 X3))
(16 (Z1 10 X1 X2 X3) (Z2 6 X2 X3 Z1))
(20 (Z1 10 X1 X2 X3) (Z3 10 X2 X3 Z1))
```

```
(22 (Z1 10 X1 X2 X3) (Z2 2 Z1 Z3) (Z3 10 X2 X3 Z1))
   Z1 = X1 X3'+ X1 X2'+ X1'X2 X3
   Z2 = Z1 Z3
   Z3 = X3 Z1 + X2 Z1 + X2'X3'Z1'

DONE
```

A.9 Recursive Optimization of EX-951

The specification defined by EX-951 is a three-input, three-output system. It was used in Example 9.5.1 of (22). The result of optimization is shown below:

```
[21] ex-951

("z1 = x1 + x2' x3' + x2 x3" "z2 = x1' x2 + x1' x3"
 "z3 = x1' x2 x3")

[22] (design ex-951 '(z1 z2 z3))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
X1 Z1 Z2'Z3'
X2'X3'Z1 Z2'Z3'
X1'X2 X3'Z1'Z2 Z3'
X1'X2'X3 Z1'Z2 Z3'
X1'X2 X3 Z1 Z2 Z3

Minimal Determining Subsets:
Z1  ((Z2 Z3) (X1 X2 X3) (X2 X3 Z2))
Z2  ((Z1 Z3) (X1 X2 X3) (X1 X2 Z1) (X1 X3 Z1))
Z3  ((Z1 Z2) (X1 X2 X3) (X1 X2 Z1) (X1 X3 Z1) (X2 X3 Z2))

(0)
(3 (Z3 3 X1 X2 X3))
(6 (Z2 6 X1 X2 X3))
(7 (Z1 7 X1 X2 X3))
(9 (Z2 6 X1 X2 X3) (Z3 3 X1 X2 X3))
(9 (Z2 6 X1 X2 X3) (Z3 3 X2 X3 Z2))
(10 (Z1 4 X2 X3 Z2) (Z2 6 X1 X2 X3))
(10 (Z1 7 X1 X2 X3) (Z3 3 X1 X2 X3))
(10 (Z1 7 X1 X2 X3) (Z3 3 X1 X3 Z1))
(10 (Z1 7 X1 X2 X3) (Z3 3 X1 X2 Z1))
(11 (Z1 7 X1 X2 X3) (Z2 4 X1 X3 Z1))
(11 (Z1 7 X1 X2 X3) (Z2 4 X1 X2 Z1))
```

```
(11 (Z1 2 Z2 Z3) (Z2 6 X1 X2 X3) (Z3 3 X1 X2 X3))
   Z1 = Z2'+ Z3
   Z2 = X1'X3 + X1'X2
   Z3 = X1'X2 X3

(11 (Z1 2 Z2 Z3) (Z2 6 X1 X2 X3) (Z3 3 X2 X3 Z2))
DONE
```

A.10 Recursive Optimization of BCDTO3

The specification defined by BCDTO3 is a four-input, four-output system. It describes a BCD to Excess-3 Code convertion system from (73). A, B, C and D represent the BCD inputs and w, x, y and z represent the Excess-3 outputs. The result of optimization is shown below:

```
[23] bcdto3

("w = A + B C + B D" "x = B' C + B' D + B C' D'"
 "y = C D + B' D + B C' D'" "z = D'")

[24] (design bcdto3 '(w x y z))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  PASSED!

Function:
A'B'D W'X Y Z'
B C'D W X'Y'Z'
B C D W X'Y Z'
A B'D W X Y Z'
A'B'C'D'W'X'Y'Z
A'B C'D'W'X Y Z
A'B'C D'W'X Y'Z
A B'C'D'W X'Y'Z
B C D'W X'Y'Z
A B'C D'W X Y'Z
A B C'D'W X Y Z

Minimal Determining Subsets:
W  ((A B X) (A B C D) (A B C Y) (A B C Z) (A B D Y)
    (A B Y Z) (A C D X) (A C X Z))
X  ((B C D) (B C Y) (B C Z))
Y  ((B C D) (B C Z) (C D X) (C X Z))
Z  ((D) (B C Y))

(0)
(1 (Z 1 D))
(7 (W 7 A B C D))
(8 (W 7 A B C Z) (Z 1 D))
(8 (W 7 A B C D) (Z 1 D))
```

```
(10 (Y 10 B C D))
(10 (X 10 B C D))
(11 (Y 10 B C Z) (Z 1 D))
(11 (Y 10 B C D) (Z 1 D))
(11 (X 10 B C Z) (Z 1 D))
(11 (X 10 B C D) (Z 1 D))
(14 (W 4 A B X) (X 10 B C D))
(15 (W 4 A B X) (X 10 B C Z) (Z 1 D))
(15 (W 4 A B X) (X 10 B C D) (Z 1 D))
(16 (X 10 B C D) (Y 6 C D X))
(16 (X 6 B C Y) (Y 10 B C D))
(17 (X 6 B C Y) (Y 10 B C Z) (Z 1 D))
(17 (W 7 A B D Y) (Y 10 B C D))
(17 (W 7 A B C Y) (Y 10 B C D))
(17 (W 7 A B C D) (Y 10 B C D))
(17 (W 7 A B C D) (X 10 B C D))
(17 (W 7 A C D X) (X 10 B C D))
(17 (X 6 B C Y) (Y 10 B C D) (Z 1 D))
(17 (X 10 B C Z) (Y 6 C X Z) (Z 1 D))
(17 (X 10 B C Z) (Y 6 C D X) (Z 1 D))
(17 (X 10 B C D) (Y 6 C X Z) (Z 1 D))
(17 (X 10 B C D) (Y 6 C D X) (Z 1 D))
(18 (W 7 A C X Z) (X 10 B C D) (Z 1 D))
(18 (W 7 A C D X) (X 10 B C D) (Z 1 D))
(18 (W 7 A B Y Z) (Y 10 B C D) (Z 1 D))
(18 (W 7 A B D Y) (Y 10 B C D) (Z 1 D))
(18 (W 7 A B C Y) (Y 10 B C D) (Z 1 D))
(18 (W 7 A B C D) (Y 10 B C Z) (Z 1 D))
(18 (W 7 A B C D) (Y 10 B C D) (Z 1 D))
(18 (W 7 A B C D) (X 10 B C Z) (Z 1 D))
(18 (W 7 A B C D) (X 10 B C D) (Z 1 D))
(18 (W 7 A B C Z) (X 10 B C D) (Z 1 D))
(18 (W 7 A B C Z) (X 10 B C Z) (Z 1 D))
(18 (W 7 A B C Z) (Y 10 B C D) (Z 1 D))
(18 (W 7 A B C Z) (Y 10 B C Z) (Z 1 D))
(18 (W 7 A B C Y) (Y 10 B C Z) (Z 1 D))
(18 (W 7 A B D Y) (Y 10 B C Z) (Z 1 D))
(18 (W 7 A B Y Z) (Y 10 B C Z) (Z 1 D))
(18 (W 7 A C D X) (X 10 B C Z) (Z 1 D))
(18 (W 7 A C X Z) (X 10 B C Z) (Z 1 D))
(20 (W 4 A B X) (X 10 B C D) (Y 6 C D X))
(20 (X 10 B C D) (Y 10 B C D))
(20 (Y 10 B C D) (Z 10 B C Y))
(20 (W 4 A B X) (X 6 B C Y) (Y 10 B C D))
```

```
(21 (W 4 A B X) (X 10 B C D) (Y 6 C X Z) (Z 1 D))
   W = A + B X'
   X = B'C + B'D + B C'D'
   Y = C'X + C Z'
   Z = D'

(21 (W 4 A B X) (X 10 B C D) (Y 6 C D X) (Z 1 D))
(21 (W 4 A B X) (X 10 B C Z) (Y 6 C X Z) (Z 1 D))
(21 (W 4 A B X) (X 10 B C Z) (Y 6 C D X) (Z 1 D))
(21 (W 4 A B X) (X 6 B C Y) (Y 10 B C Z) (Z 1 D))
(21 (W 4 A B X) (X 6 B C Y) (Y 10 B C D) (Z 1 D))
DONE
```

A.11 Optimization of NONTAB1 — a Non-Tabular Spec.

This specification tests our system's ability to identify a non-tabular specification, convert it to a tabular form and complete the optimization process. It uses Example 9.3.1 from (22) that is known to be non-tabular for R, S and T when evaluated in terms of J, K and Q. The result is shown below:

```
[4] nontab1

("q' j + q k' = s + q' t + q r' t'" "0 = r s + r t + s t")

[5] (design nontab1 '(r s t))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification Is Tabular:  FAILED!

* Converting To A Tabular Form.

Function:
J'Q'S'T'
J Q'R'S'T
K'Q R'T'
J K R'S'T
K Q R'S'T

Minimal Determining Subsets:
R  ((J Q) (Q T))
S  ((K Q) (Q T))
T  ((J K Q))

(0)
(0 (R 0 J Q))
(0 (R 0 J Q) (S 0 K Q))
(0 (S 0 K Q))
(0 (R 0 J Q) (S 0 K Q))
(6 (S 0 K Q) (T 6 J K Q))
```

```
(6 (R 0 Q T) (S 0 K Q) (T 6 J K Q))
   R = 0
   S = 0
   T = K Q + J Q'

(6 (R 0 Q T) (S 0 K Q) (T 6 J K Q))
(6 (R 0 Q T) (S 0 Q T) (T 6 J K Q))
(6 (R 0 Q T) (S 0 K Q) (T 6 J K Q))
(6 (R 0 J Q) (S 0 Q T) (T 6 J K Q))
(6 (R 0 Q T) (S 0 Q T) (T 6 J K Q))
(6 (R 0 J Q) (S 0 Q T) (T 6 J K Q))
(6 (R 0 J Q) (S 0 K Q) (T 6 J K Q))

DONE
```

A.12 Optimization of CKT2 Using Its Obverse Specification

This test runs the obverse specification ($F' = 1$) of CKT2 through the optimization system. We can compare this result with the standard optimization results shown earlier. We note that by designing with the obverse specification of a circuit, the resulting Boolean formulas are merely the complement of what they should be. One can also observe that this method involved 6 steps in the search process versus 12 for the original system, a significant reduction in search time.

```
[9] ckt2
("f = x' + y z" "g = x y' + z'" "h = x' + y' + z'")
[11] (comp-design ckt2 '(f g h))

Function:
G'Z'
G H'
H'Y'
H'X'
F'X'
F'G'
G X'Z
G Y Z
F H X

Minimal Determining Subsets:
F  ((G X) (H X) (X Y Z))
G  ((F Z) (H X Z) (X Y Z))
H  ((F X) (G X) (X Y Z))

(0)
(3 (H 3 X Y Z))
(5 (F 2 H X) (H 3 X Y Z))
(6 (G 6 X Y Z))
(6 (F 6 X Y Z))
(7 (G 4 H X Z) (H 3 X Y Z))

(7 (F 2 H X) (G 2 F Z) (H 3 X Y Z))
   F = H X
   G = F Z
   H = X Y Z
DONE
```

A.13 Optimization of EX-951 Using Its Obverse Specification

This is another example of optimizing a cicuit using its obverse specification. As before, the number of steps in the search process was reduced. We should point out that this is not always the case; the number of steps in the search process increased in some examples.

```
[12]  ex-951
("z1 = x1 + x2' x3' + x2 x3" "z2 = x1' x2 + x1' x3" "z3 = x1' x2 x3")
[13] (comp-design ex-951 '(z1 z2 z3))

Function:
Z2'Z3
X3'Z3
X2'Z3
X1 Z2
Z1'Z2'
X1'X3 Z2'
X1'X2 Z2'
X2'X3'Z2
Z1 Z2 Z3'
X2 X3 Z1'

Minimal Determining Subsets:
Z1  ((Z2 Z3) (X1 X2 X3) (X2 X3 Z2))
Z2  ((Z1 Z3) (X1 X2 X3) (X1 X2 Z1) (X1 X3 Z1))
Z3  ((Z1 Z2) (X1 X2 X3) (X1 X2 Z1) (X1 X3 Z1) (X2 X3 Z2))

(0)
(3 (Z3 3 X1 X2 X3))
(4 (Z2 4 X1 X2 X3))
(7 (Z2 4 X1 X2 X3) (Z3 3 X2 X3 Z2))
(7 (Z2 4 X1 X2 X3) (Z3 3 X1 X2 X3))
(8 (Z1 8 X1 X2 X3))

(9 (Z1 2 Z2 Z3) (Z2 4 X1 X2 X3) (Z3 3 X2 X3 Z2))
   Z1 = Z2 Z3'
   Z2 = X1 + X2'X3'
   Z3 = Z2'+ X3'+ X2'

(9 (Z1 2 Z2 Z3) (Z2 4 X1 X2 X3) (Z3 3 X1 X2 X3))
DONE
```

A.14 Non-MDS Optimization of SAMPLE

This example tests a procedure that modifies the list of minimal determining subsets. By introducing non-minimal subsets into this list we were able to obtain a further optimization of the SAMPLE circuit, from a cost of 22 to a cost of 21. The results of this modified optimization process are shown below:

```
[17] sample

("z1 = x1' x2 x3 + x1 x2' x3' + x1 x2' x3 + x1 x2 x3'"
"z2 = x1' x2 x3 + x1 x2' x3 + x1 x2 x3'"
"z3 = x1' x2' x3' + x1' x2 x3 + x1 x2' x3 + x1 x2 x3'")

[18] (non-mds-design sample '(z1 z2 z3))

* Parsing Specification and Reducing to Normal Form

* Checking To See If Specification is Tabular:  PASSED!

Function:
X1'X2'X3'Z1'Z2'Z3
X1'X2 X3'Z1'Z2'Z3'
X1'X2'X3 Z1'Z2'Z3'
X1 X2 X3 Z1'Z2'Z3'
X1 X2'X3'Z1 Z2'Z3'
X1 X2 X3'Z1 Z2 Z3
X1'X2 X3 Z1 Z2 Z3
X1 X2'X3 Z1 Z2 Z3

Minimal Determining Subsets:
Z1   ((X1 X2 X3) (X1 X2 Z2) (X1 X2 Z3) (X1 X3 Z2) (X1 X3 Z3)
      (X2 X3 Z3))
Z2   ((Z1 Z3) (X1 X2 X3) (X1 X2 Z3) (X1 X3 Z3) (X2 X3 Z1) (X2 X3 Z3))
Z3   ((X1 X2 X3) (X2 X3 Z1))

(0)
(10 (Z1 10 X1 X2 X3))
(12 (Z2 12 X1 X2 X3))
(16 (Z1 4 X1 X2 X3 Z2) (Z2 12 X1 X2 X3))
(16 (Z1 4 X1 X2 Z2) (Z2 12 X1 X2 X3))
(16 (Z1 4 X1 X3 Z2) (Z2 12 X1 X2 X3))
(16 (Z3 16 X1 X2 X3))
(16 (Z1 10 X1 X2 X3) (Z2 6 X2 X3 Z1))
```

```
(16 (Z1 10 X1 X2 X3) (Z2 6 X1 X2 X3 Z1))
(17 (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
(20 (Z1 10 X1 X2 X3) (Z3 10 X2 X3 Z1))
(20 (Z1 10 X1 X2 X3) (Z3 10 X1 X2 X3 Z1))

(21 (Z1 4 X1 X2 X3 Z2) (Z2 12 X1 X2 X3) (Z3 5 X2 X3 Z1 Z2))
   Z1 = Z2 + X1 X3'
   Z2 = X1 X2 X3'+ X1'X2 X3 + X1 X2'X3
   Z3 = Z2 + X2'X3'Z1'

(21 (Z1 4 X1 X2 X3 Z2) (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 4 X1 X2 Z2) (Z2 12 X1 X2 X3) (Z3 5 X2 X3 Z1 Z2))
(21 (Z1 4 X1 X2 Z2) (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 4 X1 X3 Z2) (Z2 12 X1 X2 X3) (Z3 5 X2 X3 Z1 Z2))
(21 (Z1 4 X1 X3 Z2) (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 10 X1 X2 X3) (Z2 6 X2 X3 Z1) (Z3 5 X2 X3 Z1 Z2))
(21 (Z1 10 X1 X2 X3) (Z2 6 X2 X3 Z1) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 10 X1 X2 X3) (Z2 6 X1 X2 X3 Z1) (Z3 5 X2 X3 Z1 Z2))
(21 (Z1 10 X1 X2 X3) (Z2 6 X1 X2 X3 Z1) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 4 X1 X3 Z2 Z3) (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
(21 (Z1 4 X1 X2 Z2 Z3) (Z2 12 X1 X2 X3) (Z3 5 X1 X2 X3 Z2))
DONE
```

Appendix B. **BORIS Recursive Optimization System Software**

This appendix contains the fully documented source code for the BORIS Design (Optimization) System. The system is composed of a variety of procedures found in eight distinct files. Two additional files which are available provide slightly modified versions of the recursive optimization system. Each file has a header which contains important information about the file itself and general information about the procedures in that file. Each procedure is described in detail and often includes examples.

The following is a short description of each of these files:

- **design.s:** Contains the main design procedures for performing a recursive optimization of digital circuits.
- **parse.s:** Includes procedures that reduce a given specification into a more convenient list-based form.
- **tabular.s:** Its procedures check to see if a given specification is tabular. If it is not, it converts it to a tabular form.
- **mds.s** Contains a variety of procedures that can be used to find the minimal determining subsets.
- **cost.s** Contains a variety of procedures used to determine the cost of a given minimal determining subset.
- **search.s** Includes procedures that utilize a branch-and-bound search technique to find the least cost, recursive realization of a given circuit.
- **data.s** Contains some predefined circuit specifications.
- **tools.s** A collection of procedures used to process Boolean sum-of-product formulas.
- **new_dsgn.s** Contains procedures that modify when the SOP formulas and associated costs for a given MDS are calculated.
- **non_mds.s** Contains procedures that create non-minimal determing subsets in addition to the MDSs.

B.1 DESIGN.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;                 D E S I G N   M O D U L E                   ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; This file contains the main design procedure used to        ;;
;; recursively optimize combinational logic circuits. It       ;;
;; accepts as its inputs a specification consisting of a       ;;
;; system of Boolean equations and a list of specified         ;;
;; outputs. The optimization procedure selectively calls the   ;;
;; appropriate subroutines which:                              ;;
;;                                                             ;;
;;   - parse the system of Boolean equations,                  ;;
;;   - ensure the specification is valid,                      ;;
;;   - find the minimal determining subsets for each output,   ;;
;;   - assign costs to each of these subsets,                  ;;
;;   - perform a branch-and-bound search for an optimal        ;;
;;       solution, and                                         ;;
;;   - display the final results.                              ;;
;;                                                             ;;
;; The optimization algorithm can be altered to:               ;;
;;                                                             ;;
;;    -  select a particular method for calculating minimal    ;;
;;       determining subsets,                                  ;;
;;    -  select a particular method for determining the cost of ;;
;;       a minimal determining subset,                         ;;
;;    -  provide an enhanced output of information concerning  ;;
;;       the minimal determing subsets and their associated    ;;
;;       cost and  Finally a  routine is                       ;;
;;    -  enables one to design using the complement of         ;;
;;       the specification.                                    ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;; FILE NAME:          DESIGN.S or DESIGN.FSL                        ;;
;;                                                                   ;;
;; DESCRIPTION:        Recursive Circuit Optimization System         ;;
;;                                                                   ;;
;; AUTHOR:             Frank M. Brown with some modifications by     ;;
;;                     Eric J. Knutson                               ;;
;;                                                                   ;;
;; DATE:               1 NOV 90                                      ;;
;;                                                                   ;;
;; AUXILIARY FILES:    From the BORIS System Software                ;;
;;                                                                   ;;
;;                     TABULAR.S       SEARCH.S                      ;;
;;                     PARSE.S         TOOLS.S                       ;;
;;                     MDS.S           DATA.S                        ;;
;;                     COST.S                                        ;;
;;                                                                   ;;
;; GETTING STARTED:    To get started, load design.fsl and all       ;;
;;                     auxiliary files at the PC Scheme System       ;;
;;                     prompt.  Then follow the instructions         ;;
;;                     and/or examples provided with each of the     ;;
;;                     algorithms found below.                       ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; DESIGN ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;; The optimization of a specific circuit is initiated by an         ;;
;; input of the form (DESIGN CIRCUIT OUTPUTS) where  CIRCUIT         ;;
;; represents the circuit specification and OUTPUTS represents       ;;
;; the designated cicuit outputs.  An example is shown below:        ;;
;;                                                                   ;;
;; [1] (design '("f = x' + y z"                                      ;;
;;               "g = x y' + z'"                                     ;;
;;               "h = x' + y'")    '(f g h) )                        ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (design circuit outputs)
  (define (design-fcn f outputs)
    (newline) (princ "Function:") (newline)
    (list-terms f)
    (let ( (mds (out-mds-lists f outputs)) )
         (solve '((0 ())) mds outputs 1000) ))
  (newline)
  (princ "* Parsing Specification and Reducing to Normal Form")
  (newline) (newline)
    (let ( (spec (simplify (complement (parse-design circuit)))))
      (princ "* Checking To See If Specification Is Tabular:  ")
      (if (tabular-spec? spec outputs)
          (begin
            (princ "PASSED!") (newline)
            (design-fcn spec outputs) )
          (begin
            (princ "FAILED!") (newline) (newline)
            (princ "* Converting To A Tabular Form.")
            (newline)
            (design-fcn (make-tabular-spec spec outputs)
                        outputs) ))))

;;;;;;;;;;;;;;;;;;;;;;;; TABULAR-SPEC? ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; The function (TABULAR-SPEC? SPEC ARGS) accepts a parsed        ;;
;; specification SPEC and a list of specified outputs ARGS.       ;;
;; It passes this information on to TABULAR-AUX? to determine     ;;
;; whether or not the specification is tabular.  It returns      ;;
;; #T (true) if the system is tabular and '() (false) if the      ;;
;; system is non-tabular.                                         ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (tabular-spec? spec args)
  (tabular-aux? spec (other-args spec args)) )
```

```
;;;;;;;;;;;;;;;;;;;;;; MAKE-TABULAR-SPEC ;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; The function (MAKE-TABULAR-SPEC SPEC ARGS) accepts a parsed   ;;
;; specification SPEC and a list of specified outputs ARGS.      ;;
;; Using the function (OTHER-ARGS SPEC ARGS), we can determine   ;;
;; the inputs to the system (all of those arguements in the      ;;
;; specification that aren't outputs).  Given the specifica-     ;;
;; tion and the inputs, we call the function DISCRIMINANTS.      ;;
;; It returns a complete tabular listing of the original         ;;
;; specification.                                                ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-tabular-spec spec args)
  (simplify (discriminants spec (other-args spec args) '())) )

;;;;;;;;;;;;;;;;;;;;;;;; OUT-MDS-LISTS ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Given a parsed specification in normal form (F = 1),          ;;
;; (OUT-MDS-LISTS F OUTPUTS) finds the minimal determining       ;;
;; subsets, and associated cost, for each of the ouputs in       ;;
;; OUTPUTS.  An example is shown below:                          ;;
;;                                                               ;;
;;  [1] (out-mds-lists '(((f) b a g) (f (a) (g)) ((b) f (g)))    ;;
;;                     '(f g) )                                  ;;
;;                                                               ;;
;;   ( (F 1 (((G))) G) (F 2 (((B)) ((A))) A B) (G 1 (((F))) F)   ;;
;;     (G 2 ((A B)) A B) )                                       ;;
;;                                                               ;;
;; The format of the output is                                   ;;
;;                                                               ;;
;;   ( (OUTPUT COST FORM MDS) (OUTPUT COST FORM MDS) ... )       ;;
;;                                                               ;;
;; where OUTPUT represents an argument found in OUTPUTS, MDS     ;;
;; represents the arguments of one of OUTPUT's minimal           ;;
;; determining subsets, FORM represents a minimum SOP formula    ;;
;; that produces OUTPUT using the arguments found in MDS, and    ;;
;; COST represents the gate-input cost associated with FORM.     ;;
```

```
;; This function will only display the MDSs corresponding to a ;;
;; given output.  To display all of the information concerning ;;
;; MDSs, including their associated cost and SOP formulas,     ;;
;; place a semi-colon in front of the OPT1 lines below and     ;;
;; remove the semi-colon from in front of the OPT2 line.       ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (out-mds-lists f outputs)
  (define (out-mds-lists-aux f outputs)
    (cond ( (null? outputs) nil)
          ( else
            (let* ( (z          (car outputs))
                    (mds-lists  (min-determining f z))
                    (mds-lists* (attach-costs f z mds-lists)) )
                  (princ z) (princ "  ")                ;; OPT1
                  (princ mds-lists) (newline)           ;; OPT1
                ; (display-cost mds-lists* z) (newline) ;; OPT2
                  (append
                    (splice z mds-lists*)
                    (out-mds-lists-aux f (cdr outputs)) )))))
  (princ "Minimal Determining Subsets:") (newline)
  (out-mds-lists-aux f outputs) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; ATTACH-COSTS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;; ATTACH-COSTS is called by the OUT-MDS-LISTS function.  It          ;;
;; requires a specification F in normal form, an output Z and         ;;
;; a complete list MDS-LISTS of minimal determining subsets           ;;
;; for the output Z.  It returns a list of information for a          ;;
;; given output in the form:                                          ;;
;;                                                                    ;;
;;   ( (COST FORM MDS) (COST FORM MDS) (COST FORM MDS) .... )         ;;
;;                                                                    ;;
;; It calculates the range of possible functions that can be          ;;
;; used to express a given output.  Using this range, it finds        ;;
;; FORM, the minimal SOP formula that produces the output in          ;;
;; terms of the arguments in the MDS.  From this minimal, SOP         ;;
;; formula, the COST is generated using a pre-selected cost           ;;
;; function.  The cost is currently determined by the number          ;;
;; of gate inputs, however, other options are described in the        ;;
;; COST.S file.  These options can be selected by removing the        ;;
;; appropriate comment marks (;;) from in front of the desired        ;;
```

```
;; option below.  The following example illustrates the use of ;;
;; this function:                                               ;;
;;                                                              ;;
;;    [1] (attach-costs '(((b) (g) f) ((a) (g) f) (b a (f) g)) ;;
;;                      'f                                      ;;
;;                      '((g) (a b)) )                          ;;
;;                                                              ;;
;;    ((1 (((G))) G) (2 (((B)) ((A))) A B))                     ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (attach-costs f z mds-lists)
  (define (costs-to-range range mds-lists)
    (define (attach-one-cost range mds)
      (let* ( (new-range   (project-range range mds))
              (min-formula (submin-interval new-range))
              (new-cost    (gate-input-cost min-formula)) )
          ;;  (new-cost    (gate-input-cost1 min-formula)) )
          ;;  (new-cost    (gate-cost min-formula)) )
            (cons new-cost
                  (cons min-formula
                        mds ))))
    (if (null? mds-lists)
        '()
        (cons (attach-one-cost range (car mds-lists))
              (costs-to-range range (cdr mds-lists)) )))
  (costs-to-range (range f z) mds-lists) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; SPLICE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; SPLICE is an auxillary procedure called by OUT-MDS-LISTS.     ;;
;; It accepts as an input a list of the form:                    ;;
;;                                                               ;;
;;   ( (COST FORM MDS) (COST FORM MDS) ... )                     ;;
;;                                                               ;;
;; and an OUTPUT.  It returns a list of the form:                ;;
;;                                                               ;;
;;   ( (OUTPUT COST FORM MDS) (OUTPUT COST FORM MDS) ... )       ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (splice x lists)
  (cond ( (null? lists) nil)
        ( else
          (cons (cons x (car lists))
                (splice x (cdr lists)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; DISPLAY-COST ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; The auxillary procedure (DISPLAY-COST ARG-LIST Z) displays     ;;
;; a list ARG-LIST that contains information about the mds,      ;;
;; the optimal two-level implementation using the mds, and the   ;;
;; associated cost. An example is shown below:                   ;;
;;                                                               ;;
;;  [1] (display-cost '((2 ((X G)) G X) (2 ((H X)) H X)          ;;
;;                      (6 (((Y) X) ((Z) X)) X Y Z))    'F)      ;;
;;  [2] F = G X                                                  ;;
;;           cost is 2                                           ;;
;;           mds is (G X)                                        ;;
;;      F = H X                                                  ;;
;;           cost is 2                                           ;;
;;           mds is (H X)                                        ;;
;;      F = X Y' + X Z'                                          ;;
;;           cost is 6                                           ;;
;;           mds is (X Y Z)                                      ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (display-cost arg-list z)
  (define (show-info fcn)
    (define (show-info-aux fcn)
      (define (write-term1 term)
        (cond ( (null? term)
                (princ "") )
              ( (atom? (car term))
                (princ (car term)) (princ " ")
                (write-term1 (cdr term)) )
              (else
                (princ (car (car term))) (princ "'")
                (write-term1 (cdr term)) )))
      (cond ( (null? fcn)
              (princ '()) )
            ( (null? (cdr fcn))
              (write-term1 (sort-term (car fcn)))
              (princ "") )
            (else
              (write-term1 (sort-term (car fcn)))
              (princ "+ ")
              (show-info-aux (cdr fcn)) )))
    (cond ( (member nil fcn)
            (princ "1") (newline) )
          ( (null? fcn)
            (princ "0") (newline) )
          (else
            (show-info-aux fcn) )))
  (cond ( (null? arg-list) (princ ""))
        (else
          (let ( (cost  (caar  arg-list))
                 (entry (cadar arg-list))
                 (mds   (cddar arg-list)) )
            (princ z) (princ " = ")
            (show-info entry) (newline)
            (princ "     cost is ") (princ cost) (newline)
            (princ "     mds  is ") (princ mds) (newline)
            (display-cost (cdr arg-list) z) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; COMP-DESIGN ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; COMP-DESIGN is identical to DESIGN with the exception that    ;;
;; the specification is processed in the form (F' = 1) instead   ;;
;; of the normal form (F = 1).  It should be emphasized that     ;;
;; COMP-DESIGN will only work if MDS1 is used to calculate the   ;;
;; minimal determining subsets.  This is because of the way      ;;
;; the interval, that bounds the output function, is defined.    ;;
;; By designing with the obverse specification, the costs are    ;;
;; assigned differently and hence a reduction in the length of   ;;
;; the search process is possible. The result of this function   ;;
;; is a system of Boolean formulas, defining each output, that   ;;
;; are the obverse of what they should be.  By passing           ;;
;; these outputs through a simple invertor, the desired          ;;
;; behavioral characteristics can be achieved.                   ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (comp-design circuit outputs)
  (define (comp-aux f outputs)
    (newline) (princ "Function:") (newline)
    (list-terms f)
    (let ( (mds (out-mds-lists f outputs)) )
      (solve '((0 ())) mds outputs 1000) ))
  (comp-aux (parse circuit) outputs) )
```

B.2 PARSE.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;;                  P A R S E   M O D U L E                     ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; This module contains algorithms that use a formal approach  ;;
;; to parsing a string or list of strings. Each of the strings ;;
;; represnets either a Boolean formula or an equation whose    ;;
;; two members are Boolean formulas.  The approach consists of ;;
;; the following basic steps:                                  ;;
;;                                                              ;;
;;    1)  The string representation of each Boolean formula is ;;
;;        converted into an equivalent tokenized form.          ;;
;;    2)  The tokenized list is then transformed into a prefix ;;
;;        AND-OR-NOT representation.                            ;;
;;    3)  The AND-OR-NOT representation is then converted to   ;;
;;        an equivalent list-based SOP form.                    ;;
;;    4)  Steps 1 through 3 are repeated for each Boolean      ;;
;;        formula in a system, with the resulting list-based   ;;
;;        SOP formulas being added together.                    ;;
;;    5)  The final result is then reduced to an equivalent    ;;
;;        sub-minimal form.                                     ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; FILE NAME:        PARSE.S or PARSE.FSL                       ;;
;;                                                              ;;
;; DESCRIPTION:      Boolean to List-Based Parser               ;;
;;                                                              ;;
;; AUTHOR:           Frank M. Brown with minor modifications    ;;
;;                   by Eric J. Knutson                         ;;
;;                                                              ;;
;; DATE:             2 NOV 90                                   ;;
```

```
;; AUXILIARY FILES:  From the BORIS System Software           ;;
;;                                                             ;;
;;                   TOOLS.FSL                                 ;;
;;                   DATA.S                                    ;;
;;                                                             ;;
;; GETTING STARTED:  To get started, load PARSE.FSL and both   ;;
;;                   auxiliary files at the PC Scheme System   ;;
;;                   prompt.  Then follow the instructions     ;;
;;                   and/or examples provided with some of the ;;
;;                   algorithms below.                         ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;; PARSE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; The procedure (PARSE OBJECT) accepts either a string or a   ;;
;; list of strings. Each string is to represent either a       ;;
;; Boolean formula or an equation whose two members are        ;;
;; Boolean formulas. Corresponding to a Boolean equation G = H ;;
;; is an equivalent Boolean equation F = 0; the procedure      ;;
;; PARSE replaces the equation G = H by the function F.        ;;
;; Similarly, the Boolean inclusion G < H is equivalent to a   ;;
;; Boolean equation F = 0 (specifically, G H' = 0). Thus each  ;;
;; of the strings in a list is treated as a Boolean function.  ;;
;; The list itself represents the Boolean sum of its           ;;
;; constituent functions. Each string is transformed by        ;;
;; PARSE-STRING into a list-based SOP (sum-of-products) form.  ;;
;; A list of strings is transformed into a list-based SOP form ;;
;; representing the sum of the constituent functions.          ;;
;;                                                             ;;
;;                  Sample-sessions:                           ;;
;;                                                             ;;
;;    [1] (parse "((a + x5)' ! a' ! b)' b")                    ;;
;;    (((A) X5 B))                                             ;;
;;                                                             ;;
;;    [2] (parse "((a + x5)' + tom * bill)' b")                ;;
;;    ((X5 (TOM) B) (X5 (BILL) B) (A (BILL) B) (A (TOM) B))    ;;
;;                                                             ;;
;;    [3] (parse '("x = a" "y = a + b"))                       ;;
;;    (((A) X) (A (X)) ((A) (B) Y) (A (Y)) (B (Y)))            ;;
```

```
;;    [4] (parse '("f = x' + y z"                                    ;;
;;                 "g = x y' + z'"                                   ;;
;;                 "h = x' + y' + z'") )                             ;;
;;    (((G) (Z)) (G (H)) ((Y) (H)) ((X) (H)) ((F) (X))              ;;
;;     ((F) (G)) ((X) G Z) (Y G Z) (F H X))                         ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse object)
  (cond ( (pair? object)
          (submin
            (parse-system object) ))
        ( (string? object)
          (submin
            (parse-string object) ))
        ( else
          (princ "I can't parse the object.") (princ object)
          (newline) )))

(define (parse-system lst)
  (if (null? lst) '()
      (add (parse-string (car lst))
           (parse-system (cdr lst)) )))

;;;;;;;;;;;;;;;;;;;;;;;; PARSE-DESIGN ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; PARSE-DESIGN is an auxillary algorithm developed to work         ;;
;; with the DESIGN function.  It differs from PARSE only in         ;;
;; that the parsed formula is not passed on to SUBMIN for           ;;
;; further minimization. Since DESIGN takes the complement of       ;;
;; this parsed formula, minimization is performed later by the      ;;
;; DESIGN function.                                                 ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-design object)
  (cond ( (pair? object)
          (parse-system object) )
        ( (string? object)
          (parse-string object) )
        ( else
          (princ "I can't parse the object.") (princ object)
          (newline) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;  PARSE-STRING  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;; The procedure (PARSE-STRING STRING) accepts a string              ;;
;; representing a Boolean formula, and returns an equivalent         ;;
;; formula in list-based SOP (sum-of-products) form.  If the         ;;
;; formula denotes an equation, i.e., if it has the form             ;;
;; F = G, it is treated as if it were the string "(F) ! (G)",        ;;
;; the Exclusive OR of F and G.  If the formula denotes an           ;;
;; inclusion, i.e., if it has the form F < G, it is treated as       ;;
;; if it were the string "F G'".                                     ;;
;;                                                                   ;;
;;                        STRING-SYNTAX:                             ;;
;;                                                                   ;;
;; Arguments:   Arguments may be any LISP symbols; lower-case        ;;
;;              letters are accepted but are converted to            ;;
;;              upper-case.                                          ;;
;;                                                                   ;;
;; Operators:   The legal infix operators are +,*, =, <, and !       ;;
;;              (the latter denoting XOR). Multiplication may        ;;
;;              be represented by juxtaposition, as well as by       ;;
;;              use of the * operator. Complementation is            ;;
;;              denoted by postfix '.                                ;;
;;                                                                   ;;
;; Parentheses: Subformulas may be set off by parentheses.           ;;
;;              Subformulas involving the ! operator should be       ;;
;;              enclosed appropriately in parentheses to avoid       ;;
;;              ambiguities in operator-precedence.                  ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-string string)
    (parse3                     ;; CONVERT TO LIST-BASED SOP
      (parse2                   ;; CONVERT TO PREFIX AND-OR-NOT
        (parse1 string) )))     ;; TOKENIZE
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; PARSE1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; This procedure tokenizes a string representing a Boolean    ;;
;; formula. The following sequence of subordinate functions is ;;
;; executed:                                                   ;;
;;                                                             ;;
;;  STRING->LIST:   Converts a string to a list of characters. ;;
;;  SPECIAL-TOKENS: Converts special characters, such as #\+   ;;
;;                  and #\SPACE, to tokens.                    ;;
;;  MAKE-SYMBOLS:   Character-sequences between special tokens ;;
;;                  are converted to symbols.                  ;;
;;  REMOVE-SPACES:  Space-tokens, previously left in place to  ;;
;;                  help delimit symbols, are removed.         ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse1 string)
  (remove-spaces
    (make-symbols
      (special-tokens
         (string->list string) ))))

(define (remove-spaces tokens)
  (cond ( (null? tokens)
          '() )
        ( (equal? (car tokens) 'space)
          (remove-spaces (cdr tokens)) )
        ( else
          (cons (car tokens)
                (remove-spaces (cdr tokens)) ))))

(define (make-symbols char-list)
  (cond ( (null? char-list)
          '() )
        ( (spec-token? (car char-list))
          (cons (car char-list)
                (make-symbols (cdr char-list)) ))
        ( else
          (cons (string->symbol
                  (list->string
                    (left-part char-list) ))
                (make-symbols
                  (right-part char-list) )))))
```

```
(define (spec-token? token)
  (member token '(< > space eq leq and or xor not one zero)) )

(define (left-part char-list)
  (cond ( (null? char-list)
          '() )
        ( (spec-token? (car char-list))
          '() )
        ( else
          (cons (car char-list)
                (left-part (cdr char-list)) ))))

(define (right-part char-list)
  (cond ( (null? char-list)
          '() )
        ( (spec-token? (car char-list))
          char-list )
        ( else
          (right-part (cdr      list)) )))

(define (special-tokens char-list)
  (cond ( (null? char-list)
          '() )
        ( (equal? (car char-list) #\( )
          (cons '< (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\) )
          (cons '> (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\+ )
          (cons 'or (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\' )
          (cons 'not (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\* )
          (cons 'and (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\! )
          (cons 'xor (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\= )
          (cons 'eq (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\< )
          (cons 'leq (special-tokens (cdr char-list))) )
        ( (and (equal? (car  char-list) #\SPACE )
               (equal? (cadr char-list) #\SPACE ) )
          (special-tokens (cdr char-list)) )
```

```
        ( (equal? (car char-list) #\SPACE )
          (cons 'space (special-tokens (cdr char-list))) )
        ( (equal? (car char-list) #\NEWLINE )
          (cons 'space (special-tokens (cdr char-list))) )
        ( else
          (cons (char-upcase (car char-list))
                (special-tokens (cdr char-list)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; PARSE2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; This parser converts a Boolean formula, expressed as a list          ;;
;; of tokens, into binary prefix form. The possible prefixes            ;;
;; are EQ, LEQ, AND, OR, NOT, and XOR (denoting Exclusive Or).          ;;
;; The underlying grammar is expressed by the following                 ;;
;; productions:                                                         ;;
;;                    eqn --> exp EQ  exp                               ;;
;;                    eqn --> exp LEQ exp                               ;;
;;                    exp --> term                                      ;;
;;                    exp --> term OR  exp                              ;;
;;                    exp --> term XOR exp                              ;;
;;                   term --> factor                                    ;;
;;                   term --> factor AND term                           ;;
;;                   term --> factor term                               ;;
;;                 factor --> atom                                      ;;
;;                 factor --> < exp >                                   ;;
;;                 factor --> factor NOT                                ;;
;;                                                                      ;;
;; The normal precedence among AND, OR, and NOT is built into           ;;
;; the grammar. Sub-expressions involving XOR should be                 ;;
;; enclosed in parentheses, however, if there is any question           ;;
;; about precedence.                                                    ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse2 tokens)
  (if (or (member 'eq  tokens)
          (member 'leq tokens) )
      (parse-eqn tokens)
      (parse-exp tokens) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                    eqn --> exp EQ  exp                                   ;;
;;                    eqn --> exp LEQ exp                                   ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-eqn tokens)
  (parse-eqn-aux '() tokens) )

(define (parse-eqn-aux left right)
  (cond ( (null? right)
          'error-in-parsing-an-equation )
        ( (equal? (car right) 'EQ)
          (list 'XOR (parse-exp left) (parse-exp (cdr right))) )
        ( (equal? (car right) 'LEQ)
          (list 'AND (parse-exp left)
                     (list 'NOT (parse-exp (cdr right))) ))
        ( else
          (parse-eqn-aux (append left (list (car right)))
                       (cdr right) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                    exp --> term                                          ;;
;;                    exp --> term OR exp                                   ;;
;;                    exp --> term XOR exp                                  ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-exp tokens)
   (cond ( (parse-t tokens)) ( else (parse-exp-aux '()
  tokens) )))
```

```
(define (parse-exp-aux left right)
  (cond ( (null? right)
          '() )
        ( (and (parse-t left)
               (equal?  (car right) 'OR)
               (parse-exp (cdr right)) )
          (list 'OR (parse-t left) (parse-exp (cdr right))) )
        ( (and (parse-t left)
               (equal?  (car right) 'XOR)
               (parse-exp (cdr right)) )
          (list 'XOR (parse-t left) (parse-exp (cdr right))) )
        ( else
          (parse-exp-aux (append left (list (car right)))
                         (cdr right) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                  term --> factor                                         ;;
;;                  term --> factor term                                    ;;
;;                  term --> factor AND term                                ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-t tokens)
  (cond ( (parse-f tokens))
        ( else
          (parse-t-aux '() tokens) )))

(define (parse-t-aux left right)
  (cond ( (null? right)
          '() )
        ( (and (parse-f left)
               (equal?  (car right) 'AND)
               (parse-t (cdr right)) )
          (list 'AND (parse-f left) (parse-t (cdr right))) )
        ( (and (parse-f left)
               (parse-t right) )
          (list 'AND (parse-f left) (parse-t right)) )
        ( else
          (parse-t-aux (append left (list (car right)))
                       (cdr right) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                factor --> atom                                           ;;
;;                factor --> < exp >                                        ;;
;;                factor --> factor NOT                                     ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse-f tokens)
  (cond ( (null? tokens)
          '() )
        ( (and (null? (cdr tokens))
               (not (spec-token? (car tokens))) )
          tokens )
        ( (and (equal? (car tokens) '<)
               (equal? (last  tokens) '>) )
          (parse-exp (all-but-last (cdr tokens))) )
        ( (and (equal? (last tokens) 'NOT)
               (parse-f (all-but-last tokens)) )
          (list 'NOT (parse-f (all-but-last tokens))) )
        ( else
          '() )))

(define (last lst)
  (car (reverse lst)) )

(define (all-but-last lst)
  (reverse (cdr (reverse lst))) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; PARSE3 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This parser converts a Boolean formula, expressed in binary ;;
;; prefix form, into list-based SOP (sum-of-products) form.     ;;
;; The possible prefixes are AND, OR, NOT, and XOR (denoting    ;;
;; Exclusive Or).   The basic SOP-processing functions          ;;
;; COMPLEMENT, MULT, and XOR (and the functions they call)      ;;
;; must be loaded.                                              ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (parse3 formula)
 (cond ( (null? formula)
          (display "Syntax error detected in PARSE3.")
          (newline) )
        ( (and (null? (cdr formula))
               (equal? (car formula) '|1|) )
          (list '()) )
        ( (and (null? (cdr formula))
               (equal? (car formula) '|0|) )
          '() )
        ( (and (null? (cdr formula))
               (not (spec-token? (car formula))) )
          (list formula) )
        ( (equal? (car formula) 'NOT)
          (complement (parse3 (cadr formula))) )
        ( (equal? (car formula) 'OR)
          (append (parse3 (cadr formula))
                  (parse3 (caddr formula)) ))
        ( (equal? (car formula) 'XOR)
          (xor (parse3 (cadr formula))
               (parse3 (caddr formula)) ))
        ( (equal? (car formula) 'AND)
          (mult (parse3 (cadr formula))
                (parse3 (caddr formula)) ))
        ( else
          (display "Syntax error detected in PARSE3.")
          (newline) )))
```

B.3 TABULAR.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;            T A B U L A R   M O D U L E                     ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; A specification for a given circuit design is condidered to ;;
;; be TABULAR, if and only if it can be represented by a truth ;;
;; table.  The proof for this was developed by Dr. Frank Brown ;;
;; and can be found as Theorem 9.3.1 (page 220) in his book    ;;
;; "Boolean Reasoning: The Logic of Boolean Equations".  The   ;;
;; purpose of this module is to provide a set of algorithms    ;;
;; that work in conjunction with the BORIS Toolset.  These     ;;
;; algorithms will determine if a given ciruit specification   ;;
;; is tabular.  Also, given a non-tabular specification, we    ;;
;; can find a tabular specification for the given circuit.  At ;;
;; this point the algorithms are designed to be used indepen-  ;;
;; dently.  Additional work will be required to incorporate    ;;
;; them as auxillary tools within the BORIS Framework.         ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; FILE NAME:        TABULAR.S or TABULAR.FSL                 ;;
;;                                                            ;;
;; DESCRIPTION:      Tabular Specification Development System ;;
;;                                                            ;;
;; AUTHOR:           Eric J. Knutson                          ;;
;;                                                            ;;
;; DATE:             18 JUL 90                                ;;
;;                                                            ;;
;; LANGUAGE:         PC SCHEME                                ;;
;;                                                            ;;
;; AUXILIARY FILES:  From the BORIS System Software           ;;
;;                                                            ;;
;;                   TOOLS.FSL                                ;;
;;                   PARSE.FSL                                ;;
;;                   DATA.S                                   ;;
```

```
;; GETTING STARTED:  To get started, load tabular.fsl and all  ;;
;;                   auxiliary files at the PC Scheme System   ;;
;;                   prompt.  Then follow the instructions     ;;
;;                   and/or examples provided with each of the ;;
;;                   algorithms found below.                   ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;; TABULAR? & TABULAR-AUX ;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; TABULAR? accepts an equation (or set of equations) in       ;;
;; standard BORIS format and a list of specified outputs       ;;
;; (arguements).  It returns #T (true) if the system described ;;
;; by the inputs is tabular and '() (false) if the system is   ;;
;; non-tabular.                                                ;;
;;                                                             ;;
;; TABULAR? calls an auxillary function TABULAR-AUX? and       ;;
;; passes to it the parsed equations (in a list format) and a  ;;
;; list of the other arguements that the outputs will be       ;;
;; evaluated with respect to.  TABULAR? then goes through a    ;;
;; recursive process to generate the discriminants that        ;;
;; describe the specification.  By definition, if any of the   ;;
;; discriminants evaluate to something other than zero or a    ;;
;; term on the designated output variables, then the specifi-  ;;
;; cation is non-tabular and the function returns '() (false). ;;
;; Otherwise if all of the discriminants evaluate to either    ;;
;; zero or a term on the output variables, then the specifi-   ;;
;; cation is tabular and the function returns #T (true).  An   ;;
;; example is shown below:                                     ;;
;;                                                             ;;
;;    [1] (tabular? '("q' j + q k' = s + q' t + q r' t'"       ;;
;;                    "0 = r s + r t + s t")                   ;;
;;                  '(j k) )                                   ;;
;;    #T                                                       ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (tabular? equations args)
  (let ( (f (simplify (complement (parse-system equations)))))
    (tabular-aux? f (other-args f args)) ))
```

```
(define (tabular-aux? f new-args)
  (cond ( (or (null? new-args)
              (independent? f new-args))
          (single-term? (unabsorb f)) )
        (else
          (and (tabular-aux? (divide f (bar (car new-args)))
                             (cdr new-args) )
               (tabular-aux? (divide f (car new-args))
                             (cdr new-args) )))))

(define (single-term? f)
  (null? (cdr f)) )

(define (independent? f args)
  (independent-aux? (get-args f) args) )

(define (independent-aux? args1 args2)
  (cond ( (null? args2))
        ( else
          (and (not (member (car args2) args1))
               (independent-aux? args1 (cdr args2)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; MAKE-TABULAR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; MAKE-TABULAR accepts as an input a non-tabular specification;;
;; consisting of one or more equations and a list of proposed  ;;
;; outputs (Z-values).  The equations get parsed and reduced  ;;
;; into a single equation which is set equal to one.  With    ;;
;; the outputs provided, we determine the inputs (all the rest ;;
;; of the literals).  This information is then passed on to    ;;
;; DISCRIMINANTS which returns a complete tabular listing of  ;;
;; all the discriminants.  Finally this listing gets sent to  ;;
;; OUTPUT-TABULAR where the final result is produced.  An     ;;
;; example is shown below:                                    ;;
;;    [1] (make-tabular  '"q' j + q k' = s q' t + q r' t'"    ;;
;;                        "0 = r s + r t + s t"               ;;
;;                       '(r s t) )                           ;;
;;                                                            ;;
;;     How Do You Want to Display Your Result?                ;;
;;                                                            ;;
;;     1.  Raw List Form                                      ;;
;;     2.  Horizontal SOP Form                                ;;
;;     3.  Vertical SOP Form                                  ;;
;;     4.  Reduced Blake Canonical Form                       ;;
;;                                                            ;;
;;     What Choice Do You Want To Select?  4                  ;;
;;                                                            ;;
;;     1 = J'K'R'S'T' + J'Q'S'T' + J K'R'S T' + J Q'R'S T'    ;;
;;         K Q R'S'T + K'Q R'T'                               ;;
;;     ()                                                     ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-tabular equations args)
  (let* ( (f       (simplify (complement (parse equations))))
          (other   (other-args f args))
          (listing (discriminants f other '())) )
    (output-tabular listing) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; DISCRIMINANTS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This function accepts an equation (in list form) and a list ;;
;; of inputs (X-values) from MAKE-TABULAR.  The accumulator     ;;
;; (acc) is initialized to NULL.  DISCRIMINANTS generates a     ;;
;; tabular list of all the discriminants with respect to the    ;;
;; inputs.  As the discriminants are being generated, they are ;;
;; filtered to ensure each of the discriminants is either zero ;;
;; or a single term.  The selection of which term to use and    ;;
;; which terms to eliminate is a deterministic process          ;;
;; depending on the order of the terms.                         ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (discriminants f other acc)
  (cond ( (null? other)
          (if (single-term? f)
              (multiply (list acc) f)
              (list (car (multiply (list acc) f))) ))
        (else
          (append
             (discriminants (divide f (bar (car other)))
                            (cdr other)
                            (append acc (list (bar (car other)))) )
             (discriminants (divide f (car other))
                            (cdr other)
                            (append acc (list (car other))) )))))

;;;;;;;;;;;;;;;;;;;;;;;; OUTPUT-TABULAR ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; This function allows the user to select his choice for the  ;;
;; output form of a resulting tabular equation.   It is called ;;
;; by the MAKE-TABULAR function and provides the user four     ;;
;; output options:                                              ;;
;;                                                              ;;
;;    1.  Raw List Form                                         ;;
;;    2.  Horizontal SOP Form                                   ;;
;;    3.  Vertical SOP Form                                     ;;
;;    4.  Reduced Blake Canonical Form                          ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (output-tabular result)
   (scroll 5)
   (writeln "How Do You Want To Display Your Result?")
   (newline)
   (writeln "1.  Raw List Form")
   (writeln "2.  Horizontal SOP Form")
   (writeln "3.  Vertical SOP Form")
   (writeln "4.  Reduced Blake Canonical Form")
   (scroll 13)
   (display "What Choice Do You Want to Select?  ")
   (let ( (choice (read)))
     (cond ( (eqv? choice 1)
             (begin (scroll 2)
                    (display result) ))
           ( (eqv? choice 2)
             (begin (scroll 2)
                    (display "1 = ")
                    (show-h result) ))
           ( (eqv? choice 3)
             (begin (scroll 2)
                    (writeln "1 =")
                    (show result) ))
           ( (eqv? choice 4)
             (begin (scroll 2)
                    (display "1 = ")
                    (show-h (bcf result)) ))
           (else (writeln "Invalid Input")) )))

;;;;;;;;;;;;;;;;;;;;;;;;; SCROLL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;; This is a simple function that inserts a selected number of        ;;
;; blank lines at the location from which it is called.  You          ;;
;; simply input the number of lines that you would like to            ;;
;; scroll.  It is used as part of the OUTPUT-TABULAR Function.        ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (scroll lines)
  (if (zero? lines)
      (princ "")
      (begin (newline)
             (scroll (- lines 1)) )))
```

B.4 MDS.S File

```
;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;; MIN-DETERMINING ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; This function is used by DESIGN, a circuit optimization         ;;
;; algorithm.  MIN-DETERMINING accepts as its inputs a parsed      ;;
;; specification F in normal form and an argument ARG which        ;;
;; represents one of the specified outputs.  It returns a list     ;;
;; of minimal determining subsets for ARG.  Five different         ;;
;; methods can be used to calculate the MDSs.  They all            ;;
;; produce identical results and differ only in the method         ;;
;; used to achieve the results and their corresponding             ;;
;; efficiency.  A method can be chosen by removing the comment     ;;
;; symbols (;;) from in front of the desired choice.               ;;
;;                                                                  ;;
;; The choices are as follows:                                      ;;
;;                                                                  ;;
;;    1) MIN-DETERMING-OLD - Original MDS Algorithm using a        ;;
;;              Redundancy Elimination Technique.                   ;;
;;    2) MDS1 - Redundancy Elimination Technique using an          ;;
;;              alternative interval to bound the output.           ;;
;;    3) MDS2 - Opposing Literals Technique using                  ;;
;;              multiplication & absorption process.                ;;
;;    4) MDS3 - Opposing Literals Technique using a Boolean        ;;
;;              expansion process.                                  ;;
;;    5) MDS4 - Opposing Literals Technique using a Boolean        ;;
;;              expansion process with intellegent selection        ;;
;;              of the variable to expand on.                       ;;
;;                                                                  ;;
;; An example of using MIN-DETERMINING is:                          ;;
;;                                                                  ;;
;; [1] (min-determining '(((b) (g) f) ((a) (g) f) (b a (f) g))      ;;
;;                      'f ))                                       ;;
;;  ((G) (A B))                                                     ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (min-determining f arg)
  ;; (min-determining-old f arg) )
  ;; (mds1 f arg) )
  ;; (sort-mds (mds2 (product-of-sums f arg))) )
  ;; (sort-mds (unabsorb (mds3 (product-of-sums f arg)))) )
     (sort-mds (unabsorb (mds4 (product-of-sums f arg)))) )

;;;;;;;;;;;;;;;;;;;;;;; PRODUCT-OF-SUMS ;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; The function (PRODUCT-OF-SUMS F ARG) accepts as its inputs    ;;
;; a parsed specification F in normal form and an argument       ;;
;; ARG.  It produces a product-of-sums formula such that if      ;;
;; all the products were multiplied out and redundant terms      ;;
;; absorbed, the remaining terms correspond to the minimal       ;;
;; determining subsets associated with ARG.  MDS2, MDS3 and      ;;
;; MDS4 all present unique ways to perform this multiplication   ;;
;; process and produce the MDSs.                                 ;;
;;                                                               ;;
;; Note that the LITERAL function below must be modified         ;;
;; according to which MDS procedure one is using.  If MDS3 or    ;;
;; MDS4 were being used the result a result will appear as:      ;;
;;                                                               ;;
;; [1] (product-of-sums '(((b) (g) f) ((a) (g) f) (b a (f) g))   ;;
;;                      'f                                       ;;
;;  ((B G) (A G))                                                ;;
;;                                                               ;;
;; However, if MDS2 were used the result would appear as:        ;;
;;                                                               ;;
;; [2] (product-of-sums '(((b) (g) f) ((a) (g) f) (b a (f) g))   ;;
;;                                                               ;;
;;  (((B) (G)) ((A) (G)))                                        ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (product-of-sums f arg)
  (define (mult-list-list lst1 lst2)
    (define (remove-extra-terms lst)
      (cond ( (null? lst)
              '() )
            ( (member (car lst) (cdr lst))
              (remove-extra-terms (cdr lst)) )
            (else
              (cons (car lst) (remove-extra-terms (cdr lst))) )))
    (define (mult-term-list term lst)
      (define (opposed p q)
        (define (opposed-aux p q acc)
          (cond ( (null? p)
                  acc )
                ( (member (bar (car p)) q)
                  (opposed-aux (cdr p) q (cons (literal (car p))
                                               acc) ))
                (else
                  (opposed-aux (cdr p) q acc) )))
        (opposed-aux p q '()) )
      (if (null? lst)
          '()
          (let ( (opposed-literals (opposed term (car lst))))
            (if (null? opposed-literals)
                (mult-term-list term (cdr lst))
                (cons (sort-term (opposed term (car lst)))
                      (mult-term-list term (cdr lst)) )))))
    (if (null? lst1)
        '()
        (remove-extra-terms
          (append (mult-term-list (car lst1) lst2)
                  (mult-list-list (cdr lst1) lst2) ))))
  (mult-list-list
    (divide f arg)
    (divide f (bar arg)) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; LITERAL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; For MDS3 or MDS4, this function accepts an input X and               ;;
;; returns it in an atomic form.  For example:                          ;;
;;                                                                      ;;
;;     [1] (literal 'z1)      or      [1] (literal '(z1))               ;;
;;     Z1                             Z1                               ;;
;;                                                                      ;;
;; For MDS2, this function accepts an input X and returns it            ;;
;; enclosed in parenthesis.  For example:                               ;;
;;                                                                      ;;
;;     [2] (literal 'z1)      or      [2] (literal '(z1))               ;;
;;     (Z1)                           (Z1)                             ;;
;;                                                                      ;;
;; Ensure that the comment symbols (;;) are removed from in             ;;
;; front of the lines of the desired option below.                      ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (literal x)
  (if (pair? x)
      (bar x)          ;; Used for MDS3 & MDS4
       x ))            ;; Used for MDS3 & MDS4
;;     x               ;; Used for MDS2
;;    (bar x) ))       ;; Used for MDS2

;;;;;;;;;;;;;;;;;;;;;;;;;; SORT-MDS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (SORT-MDS LST) is an auxiliary function called by                    ;;
;; MIN-DETERMINING.  It accepts a list of minimal determining           ;;
;; sets and sorts them alpha-numerically and by the size of             ;;
;; each set.  An example is shown below:                                ;;
;;                                                                      ;;
;;   [1] (sort-mds '((z3 z1 w u) (x a z2) (c b) (z1 x w)))              ;;
;;   ((B C) (A X Z2) (W X Z1) (U W Z1 Z3))                              ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (sort-mds lst)
  (define (sort-terms lst)
    (cond ( (null? lst)
            '() )
          ( (null? (car lst))
            (sort-terms (cdr lst)) )
          (else
            (cons (sort-term (car lst)) (sort-terms (cdr lst))) )))
  (define (size-sort f)
    (define (insert-term term lst)
      (define (lower-term? term1 term2)
        (cond ( (null? term1)
                '() )
              ( (lower-literal? (car term1) (car term2))
                #T )
              ( (equal? (car term1) (car term2))
                (lower-term? (cdr term1) (cdr term2)) )
              (else '()) ))
      (cond ( (null? lst)
              (list term) )
            ( (< (length term) (length (car lst)))
              (cons term lst) )
            ( (and (= (length term) (length (car lst)))
                   (lower-term? term (car lst)) )
              (cons term lst) )
            (else (cons (car lst)
                        (insert-term term (cdr lst)) ))))
    (if (equal? (length f) 1)
        f
        (insert-term (car f) (size-sort (cdr f))) ))
  (size-sort (sort-terms lst)) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; MDS2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                        ;;
;;          Opposing Literals Technique Using Boolean                     ;;
;;               Multiplication and Absorbtion.                           ;;
;;                                                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                        ;;
;; This function uses an opposing literals technique to                   ;;
;; find the minimal determining subsets.   The technique was              ;;
;; introduced in Section 7.2.3 of Brown's book "Boolean                   ;;
;; Reasoning:  The Logic of Boolean Equations."  To avoid                 ;;
;; creating an inordinate amount of redundant terms as                    ;;
;; the multiplication process is carried out, we introduce a              ;;
;; process that repetitively carries out multiplications and              ;;
;; absorbitons.                                                           ;;
;;                                                                        ;;
;; When using MDS2, be sure to remove the appropriate comment             ;;
;; marks (;;) and add the appropriate comment marks in front              ;;
;; of the designated lines in the LITERAL function above. This            ;;
;; translates each product, in the product-of-sums formula,               ;;
;; into a form that is recognized by the MUPROD operator.                 ;;
;;                                                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mds2 lst)
  (cond ( (null? (cdr lst))
          (car lst) )
        (else
          (mds2
            (cons (unabsorb (muprod (car lst) (cadr lst)))
                  (cddr lst)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; MDS3 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;;     Opposing Literals Technique Using Boolean Expansion          ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; This algorithm is similiar to MDS2.  It differs only in the     ;;
;; way that the product-of-sums formula is translated into a       ;;
;; sums-of-products.  While MDS2 uses a multiplication and         ;;
;; absorption process, MDS3 utilizes a Boolean expansion           ;;
;; technique.  This function accepts as its input LST, which       ;;
;; represents a product-of-sums formula.   The variable x,         ;;
;; which we will expand on, is chosen as the first literal         ;;
;; appearing in LST.  The correct expansion is:                    ;;
;;                                                                  ;;
;;                      f = x S + R S                               ;;
;;                                                                  ;;
;; where R is the product of all factors involving x with x        ;;
;; set to 0 and S is the product of all factors not involving      ;;
;; x.  If xS or RS are not in sum-of-products form, then           ;;
;; they are expanded further. This expansion continues in a        ;;
;; recursive fashion until f is reduced to a sum-of-products       ;;
;; form where each product is a term.  For example, a call to      ;;
;; MDS3 using the POS formula (s + c)(b + c)(b + d + s)(a + c)     ;;
;; appears as follows:                                             ;;
;;                                                                  ;;
;; [1] (mds3 '((s c) (b c) (b d s) (a c)) )                        ;;
;; ((S B A) (S B C) (S C A) (S C) (C B A) (B C) (D C A) (D C))     ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (mds3 lst)
  (cond ( (null? lst)
          '() )
        ( (null? (cdr lst))
          (sop lst) )
        ( (null? (cdar lst))
          (mult-all (caar lst) (mds3 (cdr lst))) )
        (else
          (let* ( (x (caar lst))
                  (S (get-S lst x))
                  (R (get-R lst x)) )
            (cond ( (and (null? S) (null? R))
                    (list (list x)) )
                  ( (null? S)
                    (append (list (list x)) (mds3 R)) )
                  ( (null? R)
                    (mult-all x (mds3 S)) )
                  (else
                    (append (mult-all x (mds3 S))
                            (mds3 (append R S)) )))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; GET-S ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; (GET-S LST X) is called from the MDS3 and MDS4 algorithms.   ;;
;; LST represents a POS formula and X represents a variable     ;;
;; to expand on.  The result is the product of all factors not  ;;
;; involving X.  Using the same example as described in MDS3    ;;
;; above, we have:                                              ;;
;;                                                              ;;
;;     [1] (get-s '((s c) (b c) (b d s) (a c)) )                ;;
;;     ((B C) (A C))                                            ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-S lst x)
  (cond ( (null? lst)
          lst)
        ( (member x (car lst))
          (get-S (cdr lst) x) )
        (else (cons (car lst)
                    (get-S (cdr lst) x))) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; GET-R ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (GET-R LST X) is called from the MDS3 and MDS4 algorithms.    ;;
;; LST represents a POS formula and X represents a variable      ;;
;; to expand on.  The result is the product of all factors       ;;
;; involving X with X set to 0.  Using the same example as       ;;
;; described in MDS3 above, we have:                             ;;
;;                                                               ;;
;;     [1] (get-r '((s c) (b c) (b d s) (a c)) )                 ;;
;;     ((C) (B D))                                               ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-R lst x)
  (cond ( (null? lst)
          lst)
        ( (single-x? lst x)
          '() )
        ( (not (member x (car lst)))
          (get-R (cdr lst) x) )
       (else (cons (remove x (car lst))
                   (get-R (cdr lst) x))) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; SINGLE-X? ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This is an auxiliary function called by GET-R that            ;;
;; determines if a given POS forumula contains a product with    ;;
;; only one literal and that literal is equal to X.   As an      ;;
;; example:                                                      ;;
;;   [1] (single-x? '((a b) (c) (a d) (c e)) 'c)                 ;;
;;   #T                                                          ;;
;;   [2] (single-x? '((a b) (a d) (c e)) 'c)                     ;;
;;   ()                                                          ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (single-x? lst x)
  (cond ( (null? lst)
          '() )
        ( (and (null?  (cdar lst))
               (equal? x (caar lst)) )
          #T )
        (else (single-x? (cdr lst) x)) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; MULT-ALL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (MULT-ALL ITEM LST) is an auxiliary function, called by         ;;
;; MDS3 and MDS4, that multiplies every term in the SOP form       ;;
;; LST by the ITEM.  For example:                                  ;;
;;                                                                  ;;
;;    [1] (mult-all 'a '((b c) (c d) (f g)))                       ;;
;;    ((a b c) (a c d) (a f g))                                     ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mult-all item lst)
  (define (mult-factor item lst)
     (if (null? lst)
         '()
         (if (member item lst)
             (list lst)
             (list (cons item lst)) )))
  (if (null? lst)
      '()
      (append (mult-factor item (car lst))
              (mult-all item (cdr lst)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; SOP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (SOP LST) is an auxiliary function called by MDS3 and MDS4      ;;
;; that converts LST (a sum of literals) from a POS format to      ;;
;; SOP format.  For example:                                       ;;
;;                                                                  ;;
;;    [1] (sop '((a b c)))                                         ;;
;;    ((A) (B) (C))                                                ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (sop lst)
  (if (null? (car lst))
      '()
      (append (list (list (caar lst)))
              (sop (list (cdar lst)))) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; MDS4 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;;     Opposing Literals Technique Using Boolean Expansion      ;;
;;          With Intelligent Variable Selection.                ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; This technique is identical to the one described in MDS3     ;;
;; with one notable exception; the literal that is chosen to    ;;
;; expand on is the one that appears most frequently in the     ;;
;; expression being expanded.  It uses the auxiliary function   ;;
;; GET-MAX to determine the variable to expand on.              ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mds4 lst)
  (cond ( (null? lst)
          '() )
        ( (null? (cdr lst))
          (sop lst) )
        ( (null? (cdar lst))
          (mult-all (caar lst) (mds4 (cdr lst))) )
        (else
          (let* ( (x    (get-max (make-count (flatten lst))))
                  (S    (get-S lst x))
                  (R    (get-R lst x)) )
            (cond ( (and (null? S) (null? R))
                    (list (list x)) )
                  ( (null? S)
                    (append (list (list x))
                            (mds4 R)) )
                  ( (null? R)
                    (mult-all x (mds4 S)) )
                  (else
                    (append (mult-all x (mds4 S))
                            (mds4 (append R S)) )))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; MAKE-COUNT ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (MAKE-COUNT LST) is an auxiliary function called by MDS3      ;;
;; and MDS4.  It counts the number of times each literal          ;;
;; appears in a given list.  It returns a list composed of        ;;
;; each literal and the number of times it appeared.  For         ;;
;; example:                                                       ;;
;;                                                                ;;
;;     [1] (make-count '(s c b c b d s a c))                      ;;
;;     ((S 2) (C 3) (B 2) (D 1) (A 1))                            ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-count lst)
  (define (count-literals item lst)
    (cond ( (null? lst) 0)
          ( (equal? item (car lst))
            (+ 1 (count-literals item (cdr lst))))
          (else (count-literals item (cdr lst))) ))
  (define (remove-all item lst)
     (cond ( (null? lst)
             '() )
           ( (equal? (car lst) item)
             (remove-all item (cdr lst)) )
           (else (cons (car lst) (remove-all item (cdr lst)))) ))
  (let ( (item (car lst)))
    (if (null? lst)
        '()
        (cons (list item (count-literals item lst))
              (make-count (remove-all item lst)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; GET-MAX ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (GET-MAX LST) is an auxiliary function called by               ;;
;; MAKE-COUNT.  It accepts a list containing literals and         ;;
;; number of times they appear in an expression.  It returns      ;;
;; the literal that appears the most often in the list.           ;;
;;                                                                ;;
;;    [1] (get-max '((S 2) (C 3) (B 2) (D 1) (A 1)))              ;;
;;    C                                                           ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (get-max lst)
   (cond ( (null? (cdr lst))
           (caar lst) )
         ( (> (cadar lst) (cadadr lst))
           (get-max (cons (car lst) (cddr lst))) )
         (else (get-max (cdr lst))) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; MDS1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;;               Redundancy Elimination Technique               ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; Given a function f and an output z, MDS1 uses a redundancy  ;;
;; elimination technique to find the minimal determining       ;;
;; subsets of f in terms of z.   The technique was introduced  ;;
;; in Section 4.9 of Brown's book "Boolean Reasoning: The      ;;
;; Logic of Boolean Equations."  Given a specification in the  ;;
;; form f = 1, the output z is expressed as an interval of the ;;
;; form [LO,HIGH*] where LOW is the lower bound expressed by   ;;
;;                                                              ;;
;;                 LO = (f/z')' * (f/z)                         ;;
;;                                                              ;;
;; and HIGH* is the complement of the upper bound expressed by ;;
;;                                                              ;;
;;              HIGH* = (f/z') * (f/z)' .                       ;;
;;                                                              ;;
;; Using a depth-first search process, MDS1 finds the minimal  ;;
;; determining subsets for functions in that interval.         ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mds1 f z)
  (let* ( (args   (remove z (find-args f)))
          (lo-hi* (interval f z))
          (lo     (car lo-hi*))
          (hi*    (cadr lo-hi*))
          (lists0 '((()) () ()))
          (max-elims (search lists0 lo hi* args)) )
        (complement-sets max-elims args) ))
```

```
;;;;;;;;;;;;;;;;;;;; MIN-DETERMINING-OLD ;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;;              Redundancy Elimination Technique                  ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This algorithm is identical to MDS1 except that the output    ;;
;; z is expressed as an interval of the form [LO,HI] where       ;;
;; LO = f/z' and HIGH = f/z.                                     ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (min-determining-old f z)
  (let* ( (args (remove z (find-args f)))
          (f0   (divide f (bar z)))
          (f1   (divide f z))
          (lists0 '((()) () ()))
          (max-elims (search lists0 f0 f1 args)) )
        (complement-sets max-elims args) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; MDS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Given an interval [LO,HI] of Boolean functions, the           ;;
;; procedure (MDS LO HI) returns the minimal determining         ;;
;; subsets for functions in the interval.                        ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (mds lo hi)
  (let* ( (args   (find-args (append lo hi)))
          (hi*    (complement hi))
          (lists0 '((()) () ()))
          (max-elims (search lists0 lo hi* args)) )
        (complement-sets max-elims args) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; INTERVAL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (INTERVAL F X) returns a list of the form (LO HI*), where     ;;
;; LO is the lower limit, and HI* is the complement of the       ;;
;; upper limit, of the interval of allowed values of X implied   ;;
;; by the equation F = 1.                                        ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (interval f x)
  (let* ( (f0  (divide f (bar x)))
          (f1  (divide f x))
          (lo  (simplify (mult f1 (complement f0))))
          (hi* (simplify (mult f0 (complement f1)))) )
        (list lo hi*) ))

;;;;;;;;;;;;;;;;;;;;;;;; COMPLEMENT-SETS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (COMPLEMENT-SETS SETS ARGS) is an auxiliary function called ;;
;; by MDS, MDS1 and MIN-DETERMINING-OLD.  SETS represents a    ;;
;; a list of maximal-redundancy subsets and ARGS is a list of  ;;
;; all possible arguments in the original specification.  This ;;
;; function returns a list of minimal determining subsets.     ;;
;; They represent the complement of the maximal-redundancy     ;;
;; subsets with respect to ARGS.  An example is shown below:   ;;
;;                                                                ;;
;;     [1] (complement-sets '((f h y z) (f g y z) (f g h))     ;;
;;                          '(f g h x y z) )                   ;;
;;     ((G X) (H X) (X Y Z))                                   ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (complement-sets sets args)
  (define (complement-set set args)
    (if (null? set)
        args
        (complement-set (cdr set)
                        (remove (car set) args) )))
  (if (null? sets)
      '()
      (cons (complement-set (car sets) args)
            (complement-sets (cdr sets) args) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; SEARCH ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;  (SEARCH LISTS F0 F1 ARGS) accepts the following arguments: ;;
;;                                                            ;;
;;    LISTS has the form (OPEN CLOSED MAXED), each component  ;;
;;       of which is a list of subsets of ARGS. OPEN comprises ;;
;;       subsets known to be eliminable, but not yet known to ;;
;;       be maximal. CLOSED comprises minimal subsets known   ;;
;;       not to be eliminable. MAXED comprises maximal        ;;
;;       eliminable subsets. All subsets are ordered, for     ;;
;;       convenience.                                         ;;
;;    F = 1 represents the known information.                 ;;
;;    F0 = F/X' and F1 = F/X, where X is the deduced argument. ;;
;;    ARGS is the list of arguments of F0 and F1.             ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (search lists f0 f1 args)
  (if (null? (car lists))
      (third lists)
      (search (expand lists f0 f1 args) f0 f1 args) ))

;;;;;;;;;;;;;;;;;;;;;;;;;; EXPAND ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; EXPAND is an auxiliary procedure called by SEARCH that     ;;
;; returns the new LISTS resulting from one cycle of          ;;
;; expansion.                                                 ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (expand lists f0 f1 args)
  (define (poss-children lists args)          ;; POS-CHILDREN
    (define (successors list args)            ;; SUCCESSORS
      (if (null? list)
          args
          (cdr (member (car list) args)) ))
    (if (null? (car lists))
        '()
        (successors (head lists) args) ))
  (define (harvest lists f0 f1 possibles)     ;; HARVEST
    (define (eliminable? subset f0 f1)        ;; ELIMINABLE?
      (let* ( (e0 (edis f0 subset))
              (e1 (edis f1 subset)) )
```

```
          (if (equal? (mult e0 e1) '())
              #T
              '() )))
    (let  ( (subset (cons (car possibles) (head lists))) )
      (cond ( (null? possibles) '())
            ( (null? (car lists)) '())
            ( (eliminable? subset f0 f1)
              (if (equal? (head lists) '())
                  (cons (list (car possibles))
                        (harvest lists f0 f1 (cdr possibles)) )
                  (cons subset
                        (harvest lists f0 f1 (cdr possibles)) )))
            ( else
              (cons (cons '* subset)
                    (harvest lists f0 f1 (cdr possibles)) )))))
(define (distribute yield lists)
  (define (remove-head lists)                    ;; REMOVE-HEAD
    (list (cdar lists)
          (second lists)
          (third lists) ))
  (define (redundant? set sets)                  ;; REDUNDANT?
    (cond ( (null? sets) '())
          ( (subset? set (car sets))
            #T )
          ( else
            (redundant? set (cdr sets)) )))
  (define (all-starred? subsets)                 ;; ALL-STARED
    (and (equal? (caar subsets) '*)
         (or (null? (cdr subsets))
             (all-starred? (cdr subsets)) )))
  (define (head-to-max lists)                    ;; HEAD-TO-MAX
     (list (cdar lists)
           (second lists)
           (cons (head lists)
                 (third lists) )))
  (define (distribute-aux yield lists)           ;; DISTRIBUTE-AUX
    (cond ( (null? (car lists)) lists)
          ( (null? yield)
             (list (cdar lists) (second lists) (third lists)) )
          ( (equal? (caar yield) '*)
            (cond ( (absorbed? (cdar yield) (second lists))
```

```
                    (distribute-aux
                      (cdr yield)
                      lists ))
                  ( else
                    ; Put this subset in the closed list.
                    (distribute-aux
                      (cdr yield)
                      (list (car lists)
                            (cons (cdar yield)
                                  (second lists) )
                            (third lists) )))))
          ( (redundant? (car yield) (third lists))
            ; A subset of one of the max-lists...
            ; forget it.
            (distribute-aux
              (cdr yield)
              lists ))
          ( else
            (distribute-aux
              (cdr yield)
              (list (append (car lists)
                            (list (car yield)) )
                    (second lists)
                    (third  lists) )))))
  (let ( (redundant (absorbed? (head lists)
                               (third lists) )))
         (cond ( redundant
                 (remove-head lists) )
               ( (and (not (redundant? (head lists)
                                       (cdar lists) ))
                      (or (null? yield)
                          (all-starred? yield) ))
                 (head-to-max lists) )
               ( else
                 (distribute-aux yield lists) ))))
(let* ( (possibles (poss-children lists args))
        (yield     (harvest lists f0 f1 possibles))
        (newlists  (distribute yield lists)) )
      newlists ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; FIND-DEDUCIBLES ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (FIND-DEDUCIBLES F) finds all of the arguments that are         ;;
;; deducible from F = 1.  For example:                             ;;
;;                                                                  ;;
;; [1] (find-deducibles '(((b) (g) f) ((a) (g) f) (b a (f) g)));;
;; Deducible: F                                                    ;;
;; Deducible: G                                                    ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (find-deducibles f)
  (define (find-deducibles-aux f args)
    (cond ( (null? args) '())
          ( else
            (cond ( (deducible? f (car args))
                    (princ "Deducible: ") (princ (car args))
                    (newline) ))
            (find-deducibles-aux f (cdr args)) )))
  (let* ( (args (find-args f)))
        (find-deducibles-aux f args) ))

;;;;;;;;;;;;;;;;;;;;;;;;; DEDUCE-ALL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (DEDUCE-ALL F) prints all arguments deducible from F = 1,       ;;
;; along with their minimal determining subsets.                   ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (deduce-all f)
  (define (deduce-all-aux f args)
    (cond ( (null? args) '())
          ( else
            (cond ( (deducible? f (car args))
                    (princ "Deducible argument:  ")
                    (princ (car args)) (newline)
                    (princ "Determining subsets: ") (newline)
                    (list-terms
                        (min-determining f (car args)) )))
            (deduce-all-aux f (cdr args)) )))
  (let* ( (args (find-args f)))
        (deduce-all-aux f args) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; DEDUCIBLE? ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                   ;;
;; (DEDUCIBLE? F X) determininines if the argument X is             ;;
;; deducible from F = 1.                                            ;;
;;                                                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (deducible? f x)
  (let* ( (f0 (divide f (bar x)))
          (f1 (divide f x))
          (product (mult f0 f1)) )
        (if (null? product)
            #T
            '() )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; UTILITIES ;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (second list)
  (cadr list) )

(define (third list)
  (caddr list) )

(define (head list)
  (caar list) )
```

B.5 COST.S File

```
;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;; GATE-INPUT-COST ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; GATE-INPUT-COST accepts a Boolean formula F in a list-based ;;
;; SOP form.  It then calculates a gate-input cost by:          ;;
;;                                                                ;;
;;    1)  Counting the number of literals in all of the         ;;
;;        products if the product contains more than one.       ;;
;;    2)  Adding the number of sums that are present to the     ;;
;;        total found in Step 1.                                ;;
;;                                                                ;;
;; An example is shown below:                                   ;;
;;                                                                ;;
;;    [1] (gate-input-cost '((x1 x2 x3) (x2 x3 (x4)) (x5)))     ;;
;;                                                                ;;
;;     9                                                         ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (gate-input-cost f)
  (if (= (length f) 1)
      (length (car f))
      (+ (length f)
         (length
           (flatten
             (remove-singletons f) )))))

;;;;;;;;;;;;;;;;;;;;;;; GATE-INPUT-COST1 ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This procedure is identical to GATE-INPUT-COST with the      ;;
;; exception that if F is a Boolean formula containing only a ;;
;; single literal, then the cost would be 0.  This compares     ;;
;; to a cost of 1 that would be produced by the COST            ;;
;; procedure.  An example is shown below:                       ;;
;;                                                                ;;
;;    [1] (gate-input-cost1 '((z1)))                            ;;
;;                                                                ;;
;;     0                                                         ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (gate-input-cost1 f)
  (if (= (length f) 1)
      (if (null? (cdar f))
          0
          (length (car f)))
      (+ (length f)
         (length
           (flatten
             (remove-singletons f) )))))

;;;;;;;;;;;;;;;;;;;;;;;;;; GATE-COST ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; GATE-COST accepts a Boolean formula F in a list-based SOP    ;;
;; form.  It then calculates a gate cost by:                    ;;
;;                                                              ;;
;;    1)  Counting the number of terms in the SOP formula       ;;
;;        that contain more than one literal.                   ;;
;;    2)  Adding the number of sums that are present to the     ;;
;;        total found in Step 1.                                ;;
;;                                                              ;;
;; An example is shown below:                                   ;;
;;                                                              ;;
;;    [1] (gate-cost '((x1 x2 x3) (x2 x3 (x4)) (x5)))           ;;
;;                                                              ;;
;;     3                                                        ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (gate-cost f)
  (cond ( (null? f)
          0 )
        ( (= 1 (length f))
          1 )
        (else
          (+ 1 (length (remove-singletons f))) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;; REMOVE-SINGLETONS ;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This is an auxillary procedure used by the cost functions     ;;
;; above.  It removes from a list of terms, those terms that     ;;
;; contain only one literal.                                     ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (remove-singletons f)
  (cond ( (null? f)
          '() )
        ( (null? (cdar f))
          (remove-singletons (cdr f)) )
        (else
          (cons (car f)
                (remove-singletons (cdr f)) ))))
```

B.6 SEARCH.S File

```
;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; SOLVE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; SOLVE controls the branch-and-bound search process.  It is    ;;
;; called by (SOLVE QUEUE MDS OUTPUTS MAXCOST).   QUEUE will     ;;
;; used to maintain a list of partial paths through the search   ;;
;; space, but is initialized to begin with.  MDS represents a    ;;
;; list of all of the MDSs, the output they are associated       ;;
;; with, their SOP representation and their resulting cost.      ;;
;; OUTPUTS simply represents the designated outputs for the      ;;
;; system.  And finally, MAXCOST represents an upper bound on    ;;
;; the accumulated cost that will be allowed to occur before     ;;
;; the search process is terminated.   SOLVE checks to ensure    ;;
;; that a number of conditions are met throughout the search     ;;
;; process and detects when a solution is achieved.  This        ;;
;; procedure is called by the DESIGN procedure in the            ;;
;; DESIGN.S file.  An example is shown below:                    ;;
;;                                                               ;;
;;    [1] (solve '((0 ())) '((f 1 (((g))) g)                     ;;
;;                           (f 2 (((b)) ((a))) a b)             ;;
;;                           (g 1 (((f))) f)                     ;;
;;                           (g 2 ((a b)) a b)) '(f g) 1000)     ;;
;;    (0)                                                        ;;
;;    (2 (F 2 A B))                                              ;;
;;    (2 (G 2 A B))                                              ;;
;;                                                               ;;
;;    (3 (F 1 G) (G 2 A B))                                      ;;
;;       F = G'                                                  ;;
;;       G = A B                                                 ;;
;;                                                               ;;
;;    (3 (F 2 A B) (G 1 F))                                      ;;
;;    DONE                                                       ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (solve queue mds outputs maxcost)
  (cond ( (null? queue)
          (newline)
           'fail )
        ( (and (subset? outputs (cadar queue))
               (= maxcost 1000) )
          (newline) (newline) (print-assignment (car queue))
          (newline) (print-simplified-fcns (cddar queue))
          (solve-cycle queue mds outputs (caar queue)) )
```

```
        ( (and (not (subset? outputs (cadar queue)))
               (< maxcost 1000) )
          (solve-cycle queue mds outputs maxcost) )
        ( (and (subset? outputs (cadar queue))
               (= maxcost (caar queue)) )
          (newline) (print-assignment (car queue))
          (solve-cycle queue mds outputs (caar queue)) )
        ( (< maxcost (caar queue))
          (newline) (beep) (beep)
          'done )
        ( else
          (newline) (print-assignment (car queue))
          (solve-cycle queue mds outputs 1000) )))

;;;;;;;;;;;;;;;;;;;;;;; PRINT-ASSIGNMENT ;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                 ;;
;; PRINT-ASSIGNMENT prints the current path being examined.        ;;
;; For example:                                                    ;;
;;                                                                 ;;
;;    [1] (print-assignment '(3 (f g) (f 1 (((g))) g)              ;;
;;                                    (g 2 ((a b)) a b)))          ;;
;;    (3 (F 1 G) (G 2 A B))                                        ;;
;;                                                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (print-assignment assgn)
  (define (extract-assignments mds-list)
    (define (remove-fcn-part mds)
      (cons (car mds)
            (cons (cadr mds)
                  (cdddr mds) )))
    (cond ( (null? mds-list)
            '() )
          ( else
            (cons (remove-fcn-part (car mds-list))
                  (extract-assignments (cdr mds-list)) ))))
  (princ (cons (car assgn)
               (extract-assignments (cddr assgn)) )))
```

```
;;;;;;;;;;;;;;;;;;;; PRINT-SIMPLIFIED-FCNS ;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; PRINT-SIMPLIFIED-FCNS extracts the solution from the queue    ;;
;; and displays it.  For example:                                ;;
;;                                                               ;;
;;    [1] (print-simplified-fcns '((f 1 (((g))) g)               ;;
;;                                 (g 2 ((a b)) a b)))           ;;
;;       F = G'                                                  ;;
;;       G = A B                                                 ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (print-simplified-fcns assignment)
  (cond ( (null? assignment)
          '() )
        ( else
          (princ "   ") (princ (caar assignment)) (princ " = ")
          (show-h (caddar assignment))
          (print-simplified-fcns (cdr assignment)) )))

;;;;;;;;;;;;;;;;;;;;;;;; SOLVE-CYCLE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; SOLVE-CYCLE expands the least-cost node and then passes the   ;;
;; new information on to SOLVE.  SOLVE then checks to see if     ;;
;; a terminating condition exists.  If not, SOLVE transfers      ;;
;; control back to SOLVE-CYCLE for further expansion.  This      ;;
;; cyclic action continues until a solution or some other        ;;
;; terminating condition is encountered.                         ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (solve-cycle queue mds outputs maxcost)
  (let* ( (mother (car queue))
          (kids (collect-children mother mds outputs)) )
        (solve
          (best-first
            (insert-kids kids (cdr queue)) )
          mds
          outputs
          maxcost )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;; INSERT-KIDS ;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; This procedure helps to avoid duplicate assignments.         ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (insert-kids kids others)
  (cond ( (null? kids)
          others )
        ( (member (car kids) others)
          (insert-kids (cdr kids) others) )
        ( else
          (cons (car kids)
                (insert-kids (cdr kids) others) ))))

;;;;;;;;;;;;;;;;;;;;;;;;; BEST-FIRST ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (BEST-FIRST QUEUE) accepts a queue of partial paths through ;;
;; the state-space. It returns the queue, re-arranged so that  ;;
;; the leading member has cost minimal in the queue. The cost  ;;
;; of a partial path is the first element in the list          ;;
;; representing the partial path.                              ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (best-first queue)
  (define (path-cost partial-path)
    (car partial-path) )
  (cond ( (null? queue) nil)
        ( else
          (let ( (bf-cdr (best-first (cdr queue))) )
               (cond ( (null? bf-cdr)
                       queue )
                     ( (> (path-cost (car queue))
                          (path-cost (car bf-cdr)) )
                       (append bf-cdr (list (car queue))) )
                     ( else
                       (cons (car queue)
                             bf-cdr )))))))
```

```
;;;;;;;;;;;;;;;;;;;;;; COLLECT-CHILDREN ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; It is called by (COLLECT-CHILDREN ASSGN MDS OUTPUTS) where     ;;
;; OUTPUTS is a list of outputs to which arguments are to be      ;;
;; assigned and MDS is a list of the minimal determining          ;;
;; subsets associated with each output. For example:              ;;
;;                                                                ;;
;; OUTPUTS = (Z1 Z2 Z3)                                           ;;
;;     MDS = ((Z1 (...) 5 A D Z2) (Z1 7 (...) A D Z3)             ;;
;;            (Z1 9 (...) A B C D) (Z2 4 (...) C Z1)              ;;
;;            (Z2 12 (...) A B C D) (Z2 10 (...) A C D Z3)        ;;
;;            (Z3 3 (...) B C) (Z3 5 (...) B Z2))                 ;;
;;         ...where (...) is an SOP formula.                      ;;
;;                                                                ;;
;; ASSGN is an assignment-sequence, i.e., a list of the form      ;;
;;                                                                ;;
;;     (PATH-COST ASSIGNED (OUT COST FCN ARG ARG ...)             ;;
;;                         (OUT COST FCN ARG ARG ...) ...).       ;;
;;                                                                ;;
;; representing a path in assignment-space. COST is the sum of    ;;
;; the individual function-costs, ASSIGNED is a list of           ;;
;; the outputs currently assigned, OUT is the name of an          ;;
;; output-variable to which arguments are being assigned, FCN     ;;
;; is the list-representation of an SOP formula, COST is the      ;;
;; gate-input cost of FCN, and ARG ARG ... are the arguments      ;;
;; of FCN.  For example:                                          ;;
;;                                                                ;;
;;     (13 (Z2 Z1) (Z2 4 (...) C Z1) (Z1 9 (...) A B C D))        ;;
;;                                                                ;;
;; A child of an assignment-sequence S is a one-step extension    ;;
;; of S, i.e., an assignment-sequence in which all of the         ;;
;; assignments in A are maintained and in which an additional     ;;
;; output variable, Zi, is assigned. The child is LEGAL if        ;;
;; every argument assigned to Zi is either a basic input or       ;;
;; one of the outputs already assigned.  For example:             ;;
;;                                                                ;;
;;     (16 (Z3 Z2 Z1) (Z3 3 (...) B C) (Z2 4 (...) C Z1)          ;;
;;                    (Z1 9 (...) A B C D))                       ;;
```

```
;; This procedure returns a list of all legal children of the  ;;
;; partial assignment ASSGN. Each assignment in MDS is         ;;
;; examined to see if it is is part of a legal child of ASSGN; ;;
;; if so, it is used to form a child of ASSGN; the child is    ;;
;; added to the list of children to be returned by             ;;
;; COLLECT-CHILDREN.                                           ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (collect-children assgn mds outputs)
  (cond ( (null? mds) nil)
        ( (not  (illegal-child? (car mds) assgn outputs))
          (cons (extended-assgn (car mds) assgn)
                (collect-children assgn (cdr mds) outputs) ))
        ( else
          (collect-children assgn (cdr mds) outputs) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; ILLEGAL-CHILD ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                                ;;
;; Given an assignment-sequence ASSGN and an output-list                          ;;
;; OUTPUT, let ASSIGNED denote the outputs already assigned.                      ;;
;; An assignment MDS-ENTRY, e.g., (Z2 C Z1), of arguments to                      ;;
;; an output is illegal if                                                        ;;
;;    (a) the output has already been assigned or                                 ;;
;;    (b) the argument-set is illegal.                                            ;;
;;                                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (illegal-child? mds-entry assgn outputs)
  (let ( (assigned (cadr assgn)) )
       (cond ( (or (member (car mds-entry) assigned)
                   (illegal-arguments? (cdr mds-entry)
                                       assigned
                                       outputs ))
               t ))))
```

```
;;;;;;;;;;;;;;;;;;;;; ILLEGAL-ARGUMENTS? ;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; ARGS is a list of arguments, ASSIGNED the list of outputs     ;;
;; already assigned, and OUTPUTS the list of all outputs to be   ;;
;; assigned. We will allow an output Z to be an argument for     ;;
;; another output provided Z has already been assigned; we       ;;
;; avoid feedback-loops, therefore, by allowing only             ;;
;; "feed-forward" paths of outputs to outputs. Thus a member     ;;
;; of ARGS is illegal iff it is a member of OUTPUTS but not a    ;;
;; member of ASSIGNED.                                           ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (illegal-arguments? args assigned outputs)
  (cond ( (null? args) nil)
        ( (and (member (car args) outputs)
               (not (member (car args) assigned)) )
          t )
        ( else
          (illegal-arguments? (cdr args) assigned outputs) )))

;;;;;;;;;;;;;;;;;;;;;;;; EXTENDED-ASSGN ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (EXTENDED-ASSGN MDS-ENTRY ASSGN) returns the extended         ;;
;; assignment-sequence formed by introducing the assignment      ;;
;; MDS-ENTRY into the assignment-sequence ASSGN. The tasks       ;;
;; here are to update the cost, to augment the list of           ;;
;; assigned outputs, and to introduce the new assignment. As     ;;
;; noted in the discussion for COLLECT-CHILDREN, ASSGN is an     ;;
;; assignment-sequence, i.e., a list of the form                 ;;
;;                                                               ;;
;;     (PATH-COST ASSIGNED (OUT COST FCN ARG ARG ...)            ;;
;;                         (OUT COST FCN ARG ARG ...) ...).      ;;
;;                                                               ;;
;; MDS-ENTRY has the form (Z COST FCN ARG ARG ...), where Z is   ;;
;; the output under consideration, ARG ARG ... are the inputs    ;;
;; to be used, FCN is a subminimal SOP formula realizing Z       ;;
;; from ARG ARG ..., and COST is the gate-input cost of FCN.     ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (extended-assgn mds-entry assgn)
  (define (insert-mds mds-entry others)
    (cond ( (null? others)
            (list mds-entry) )
          ( (lower-literal? (car mds-entry) (caar others))
            (cons mds-entry others) )
          ( else
            (cons (car others)
                  (insert-mds mds-entry (cdr others)) ))))
  (cons (+ (car assgn) (cadr mds-entry))  ;; Add the cost of the
                                          ;; new mds-entry,
        (cons (sort-term (cons (car mds-entry) ;; introduce the
                               (cadr assgn) )) ;; new output,
              (insert-mds mds-entry            ;; and introduce
                          (cddr assgn) ))))    ;; the new mds-
                                               ;; entry itself.
```

B.7 DATA.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;                   D A T A   M O D U L E                    ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; This module defines a variety of circuit specifications.  ;;
;; Once the circuit specification is defined, one needs only ;;
;; refer to its designated name when using it.  For example, ;;
;; it can be used with the DESIGN function as follows:       ;;
;;                                                            ;;
;;   [1] (design ckt1 '(f g))                                 ;;
;;                                                            ;;
;; Additional circuit specifications may be added to this    ;;
;; database as they are encountered.                         ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; FILE NAME:        DATA.S                                   ;;
;;                                                            ;;
;; DESCRIPTION:      Circuit Specifications                   ;;
;;                                                            ;;
;; AUTHORS:          Frank M. Brown & Eric J. Knutson         ;;
;;                                                            ;;
;; DATE:             1 NOV 90                                 ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;,;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;; CKT1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ckt1
  '("f = a' + b'"
    "g = a b") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; CKT2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ckt2
  '("f = x' + y z"
    "g = x y' + z'"
    "h = x' + y' + z'") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; CKT3 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ckt3
  '("z1 = a b' + a' b + b c"
    "z2 = a' b' + a b"
    "z3 = b' + c'") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; CKT4 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ckt4
  '("w = a p + q"
    "x = a p"
    "y = p + a' r"
    "z = q") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; CKT5 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ckt5
  '("v = q' r + t"
    "w = a p + q"
    "x = a p"
    "y = p + a' r"
    "z = q") )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; WSU-CKT ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define wsu-ckt
  '("z1 = a b + a c' + b d' + c' d'"
    "z2 = b' c + a' c d"
    "z3 = b' c") )

;;;;;;;;;;;;;;;;;;;;;;;;;; EXAMPLE ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define example '("d = a b + a c + b c"
                  "s = a ! b ! c"
                  "u = a b s' + a' b' s") )

;;;;;;;;;;;;;;;;;;;;;;;;;; SAMPLE ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define sample
  '("z1 = x1' x2 x3 + x1 x2' x3' + x1 x2' x3 + x1 x2 x3'"
    "z2 = x1' x2 x3 + x1 x2' x3 + x1 x2 x3'"
    "z3 = x1' x2' x3' + x1' x2 x3 + x1 x2' x3 + x1 x2 x3'") )

;;;;;;;;;;;;;;;;;;;;;;;;;; EX-951 ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define ex-951
  '("z1 = x1 + x2' x3' + x2 x3"
    "z2 = x1' x2 + x1' x3"
    "z3 = x1' x2 x3") )

;;;;;;;;;;;;;;;;;;;;;;;;;; BCDTO3 ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define bcdto3
  '("w = a + b c + b d"
    "x = b' c + b' d + b c' d'"
    "y = c d + b' d + b c' d'"
    "z = d'") )

;;;;;;;;;;;;;;;;;;;;;;;;;; MANO1 ;;;;;;;;;;;;;;;;;;;;;;;;;;

(define mano1
  '("f1 = a' b c' + a' b' c + a b' c' + a b c"
    "f2 = a b + a c + b c") )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; HALF-ADD ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define half-add
  '("s = x' y + x y'"
    "c = x y") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; NONTAB1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define nontab1
  '("q' j + q k' = s + q' t + q r' t'"
    "0 = r s + r t + s t") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; NONTAB2 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define nontab2
  '("z1 + z2 = x1 x2' x3") )
```

B.8 TOOLS.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;                   T O O L S   M O D U L E                  ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; This program provides procedures for processing Boolean    ;;
;; sum-of-products (SOP) formulas. The representation assumed ;;
;; for an SOP formula is a list of representations of the     ;;
;; terms in the formula. A term is represented as a list of   ;;
;; representations of the literals in the term. A literal X is ;;
;; represented by the symbol X (X may be an arbitrarily       ;;
;; complex symbol); a literal X' is represented by the list   ;;
;; (X). Thus the Boolean formula                              ;;
;;                                                            ;;
;;    ax + bc'x'y + y'z' + dz + xy'                           ;;
;;                                                            ;;
;; is represented by the list                                 ;;
;;                                                            ;;
;;    ( (a x) (b (c) (x) y) ((y) (z)) (d z) (x (y)) ) .  (*)  ;;
;;                                                            ;;
;; If the symbol FOO has value "ax + bc'x'y + y'z' + dz + xy'" ;;
;; then (PARSE FOO) also returns the list (*).   A string     ;;
;; accepted by PARSE is not restricted to be an SOP formula   ;;
;; (see the discussion accompanying the definition of PARSE); ;;
;; however, PARSE returns an equivalent SOP formula of the    ;;
;; form shown in (*).  Let F1,...,Fn, G1,...,Gn be represen-  ;;
;; tations for SOP formulas. Then the system                  ;;
;;                                                            ;;
;;              F1 = G1                                       ;;
;;              F2 = G2                                       ;;
;;                ...                                         ;;
;;              Fn = Gn                                       ;;
;;                                                            ;;
;; is represented by the list                                 ;;
;;                                                            ;;
;;    ( (eq F1 G1) (eq F2 G2)  .. (eq Fn Gn) ) .              ;;
```

```
;; A logical inclusion F =< G (i.e., F implies G) appearing in ;;
;; a system is represented by the sub-list (le F G), read      ;;
;; "F is less than or equal to G." Thus the system             ;;
;;                                                             ;;
;;            a'b'c'        =   x'yz                           ;;
;;            ab + ac       =   x'y' + y'z'                    ;;
;;            a'b + b'c     =<  xy + x'z                       ;;
;;            ac + bc + a'c' =< x' + yz                        ;;
;;            ab'           =<  x' + z'                        ;;
;;                                                             ;;
;; is represented by the list                                  ;;
;;                                                             ;;
;;   ( (eq (((a) (b) (c)))        (((x) y (z))) )              ;;
;;     (eq ((a b) (a c))          (((x) (y)) ((y) (z))) )      ;;
;;     (le (((a) b) ((b) c))      ((x y) ((x) z)) )            ;;
;;     (le ((a c) (b c) ((a) (c))) (((x)) (y z)) )             ;;
;;     (le ((a (b)))              (((x)) ((z))) ) )) .         ;;
;;                                                             ;;
;; A friendlier representation, based on strings, is           ;;
;; recognized:                                                 ;;
;;                                                             ;;
;;   (system "a'b'c'        "  eq  "x'yz       "               ;;
;;           "ab + ac       "  eq  "x'y' + y'z' "              ;;
;;           "a'b + b'c     "  le  "xy + x'z   "               ;;
;;           "ac + bc + a'c'"  le  "x' + yz    "               ;;
;;           "ab'           "  le  "x' + z'    " ) .           ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; REDUCE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; This function reduces a system of equations and inclusions  ;;
;; to an SOP formula F such that F = 0 is equivalent to the    ;;
;; original system. The system may have one of two forms:      ;;
;;                                                             ;;
;;  FORM1:  (system "a + b'"          eq  "a b + (tom ! sam)" ;;
;;                  "x' * y + bill'"  le  "mary"               ;;
;;                  "z"               eq  "a' ! tom" )         ;;
;;                                                             ;;
;;  FORM2:  ( (eq ((x)) ((tom bill) (x (y))))                  ;;
;;          (le (((u) v)) (((x) y) (mary x))) )                ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (reduce system)
  (if (equal? (car system) 'system)
      (parse-reduce (cdr system))
      (plain-reduce  system) ))

(define (parse-reduce sys)
  (cond ( (null? sys)
          '() )
        ( (and (string? (car sys))
               (string? (caddr sys))
               (equal? (cadr sys) 'eq ) )
          (append (xor (parse (car sys))
                       (parse (caddr sys)) )
                  (parse-reduce (cdddr sys)) ))
        ( (and (string? (car sys))
               (string? (caddr sys))
               (equal? (cadr sys) 'le ) )
          (append (mult (parse (car sys))
                        (complement (parse (caddr sys))) )
                  (parse-reduce (cdddr sys)) ))
        ( else
          (princ "Syntax-error encountered in PARSE-REDUCE.")
          (newline) )))

(define (plain-reduce sys)
  (cond ( (null? sys) nil)
        ( (eq? (caar sys) 'eq)
          (append (xor (cadar sys) (caddar sys))
                  (reduce (cdr sys)) ))
        ( (eq? (caar sys) 'le)
          (append (mult (cadar sys) (complement (caddar sys)))
                  (reduce (cdr sys)) ))
        ( (eq? (caar sys) 'ge)
          (append (mult (caddar sys) (complement (cadar sys)))
                  (reduce (cdr sys)) ))
        ( else
          (writeln "Error in PLAIN-REDUCE procedure.") )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;       B A S I C   B O O L E A N   O P E R A T O R S                  ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; The following is a collection of Boolean operators designed          ;;
;; to work on SOP formulas.  They include multiplication                ;;
;; addition, addition, complement, Exclusive-Or, Exclusive-Nor          ;;
;; and others.                                                          ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (add f1 f2) returns the logical OR, in SOP form, of two SOP          ;;
;; formulas. Elementary simplification-housekeeping is                  ;;
;; performed on the result.                                             ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (add f1 f2)
  (cond ( (null? f1) f2)
        ( (absorbed? (car f1) f2)
          (add (cdr f1) f2) )
        ( else
          (add (cdr f1)
               (cons (car f1)
                     (remove-supersets (car f1) f2) )))))

(define (sum f1 f2)    ;; Alternative notation.
  (add f1 f2) )

(define (remove-supersets term f)
  (cond ( (null? f) nil)
        ( (subset? term (car f))
          (remove-supersets term (cdr f)) )
        ( else
          (cons (car f)
                (remove-supersets term (cdr f)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; COMPLEMENT ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (complement f) returns the complement, in SOP form, of the    ;;
;; SOP formula f.  The recursive rule                            ;;
;;                                                               ;;
;;     f' = x'(f/x')' + x(f/x)'                                  ;;
;;                                                               ;;
;; is implemented, where x is any argument of f. Thus each       ;;
;; term of the formula for f' will contain x' or x as a          ;;
;; literal.                                                      ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (complement f)
  (cond ((null? f) (list nil))
        ((member nil f) nil)
        (else
         (let* ((arg  (first-arg f))
                (narg (bar arg))
                (f0   (divide f narg))
                (f1   (divide f arg))
                (comp0 (complement f0))
                (comp1 (complement f1)) )
               (append (prefix narg comp0)
                       (prefix arg  comp1) )))))

;;;;;;;;;;;;;;;;;;;;;;;;;; MULT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (mult f g) returns the logical AND, in SOP form, of the       ;;
;; SOP formulas f and g. The recursive rule                      ;;
;;                                                               ;;
;;          f * g = x'(f/x' * g/x') + x (f/x * g/x)              ;;
;;                                                               ;;
;; is implemented, where x is any argument appearing in f.       ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (mult f g)
  (cond ((null? f) nil)
        ((null? g) nil)
        ((member nil f) g)
        ((member nil g) f)
        (else
         (let* ((arg  (first-arg f))
                (narg (bar arg))
                (f0   (divide f narg))
                (f1   (divide f arg))
                (g0   (divide g narg))
                (g1   (divide g arg))
                (product0 (mult f0 g0))
                (product1 (mult f1 g1)) )
               (append (prefix narg product0)
                       (prefix arg  product1) )))))

(define (multiply f g)    ;; Alternative notation.
  (mult f g) )

(define (prod f g)        ;; Alternative notation.
  (mult f g) )

(define (product f g)     ;; Alternative notation.
  (mult f g) )

;;;;;;;;;;;;;;;;;;;;;;;;;; MU-PRODUCT ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; The mu-product of two SOP formulas is their term-by-term       ;;
;; product.                                                       ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (muprod f g)
  (cond ( (null? f) nil)
        ( else
          (append (muprod-tf (car f) g)
                  (muprod (cdr f) g) ))))
```

```
(define (muprod-tf term f)
  (cond ( (null? term) f)
        ( (null? f)    nil)
        ( else
          (let ( (prod (muprod-tt term (car f))))
               (cond ( (equal? prod 0)
                       (muprod-tf term (cdr f)) )
                     (else
                       (cons prod
                             (muprod-tf term (cdr f)) )))))))

(define (muprod-tt t1 t2)
  (cond ( (null? t1) t2)
        ( (member (bar (car t1)) t2)
          0 )
        ( else
          (let ( (rest (muprod-tt (cdr t1) t2)))
               (cond ( (equal? rest 0)
                       0 )
                     ( (member (car t1) t2)
                       rest )
                     ( else
                       (cons (car t1) rest) ))))))

;;;;;;;;;;;;;;;;;;;;;;;;;; DIVIDE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; If F is a Boolean SOP formula and x is a literal, then f/x   ;;
;; is represented in the following equation:                    ;;
;;                                                              ;;
;;     f = (f/x)x + r                                           ;;
;;                                                              ;;
;; where r is the remainder.  In other words, f/x is the        ;;
;; result of removing an x from all of the terms that contain   ;;
;; an x and deleting all of the terms that don't contain an x.  ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(dofine (divide f x)
  (cond ((null? f) nil)
        ((member (bar x) (car f))
         (divide (cdr f) x) )
        (else (cons (remove x (car f))
                    (divide (cdr f) x) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; DIVIDE-BY-TERM ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This operation divides a Boolean SOP formula by a term. The ;;
;; result remains in a SOP form                                  ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (divide-by-term f term)
  (cond ( (null? term) f)
        ( else
          (divide (divide-by-term f (cdr term))
                  (car term) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; FACTOR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (factor f x) returns the result of factoring a literal x      ;;
;; from the Boolean SOP formula f.  The result remains in        ;;
;; SOP form.                                                     ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (factor f x)
  (cond ( (null? f) nil)
        ( (member x (car f))
          (cons (remove x (car f))
                (factor (cdr f) x) ))
        ( else
          (factor (cdr f) x) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; XOR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (XOR F G) returns the logical EXCLUSIVE-OR, in SOP form, of ;;
;; the SOP formulas F and G. The recursive rule                  ;;
;;                                                                ;;
;;     f XOR g = x'(f/x' XOR g/x') + x (f/x XOR g/x)             ;;
;;                                                                ;;
;; is implemented, where x is any argument appearing in f.       ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (xor f g)
  (cond ((null? f) g)
        ((null? g) f)
        ((member nil f) (complement g))
        ((member nil g) (complement f))
        (else
         (let* ((arg  (first-arg f))
                (narg (bar arg))
                (f0   (divide f narg))
                (f1   (divide f arg))
                (g0   (divide g narg))
                (g1   (divide g arg))
                (xor0 (xor f0 g0))
                (xor1 (xor f1 g1)) )
           (append (prefix narg xor0) (prefix arg  xor1) )))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; XNOR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                        ;;
;; (XNOR F G) returns the logical EXCLUSIVE-NOR, in SOP form,             ;;
;; OF the SOP formulas F and G. The recursive rule                        ;;
;;                                                                        ;;
;;     f XNOR g = x'(f/x' XNOR g/x') + x (f/x XNOR g/x)                   ;;
;;                                                                        ;;
;; is implemented, where x is any argument appearing in f.                ;;
;;                                                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (xnor f g)
  (cond ((null? f) (complement g))
        ((null? g) (complement f))
        ((member nil f) g)
        ((member nil g) f)
        (else
         (let* ((arg  (first-arg f))
                (narg (bar arg))
                (f0   (divide f narg))
                (f1   (divide f arg))
                (g0   (divide g narg))
                (g1   (divide g arg))
                (xnor0 (xnor f0 g0))
                (xnor1 (xnor f1 g1)) )
           (append (prefix narg xnor0)
                   (prefix arg  xnor1) )))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;     C A N O N I C A L   F O R M   O P E R A T O R S        ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; BCF ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; (BCF F) returns the Blake canonical form of F, where F is a ;;
;; SOP formula.  The interior list-format is returned.         ;;
;; (LISTBCF F) returns BCF(F) in conveniently-readable form.   ;;
;; Both procedures are based on the relation                   ;;
;;                                                             ;;
;;    BCF(f) = ABS([x' + BCF(f/x)] # [x + BCF(f/x')])   (1)    ;;
;;                                                             ;;
;; where                                                       ;;
;;                                                             ;;
;;    -- ABS is an operator which removes absorbed terms;      ;;
;;    -- f/u denotes the quotient of f with respect to u,      ;;
;;       i.e., the result of making the substitution u = 1     ;;
;;       in f; and                                             ;;
;;    -- # is the "mu-product" operator, indicating explicit   ;;
;;       term-by-term cross-multiplication.                    ;;
;;                                                             ;;
;; When (1) is multiplied out, the result is                   ;;
;;                                                             ;;
;;    BCF(f) = ABS(x'BCF0 + x BCF1 + PROD)              (2)    ;;
;;                                                             ;;
;; where BCF0 denotes BCF(f/x'), BCF1 denotes BCF(f/x ) and    ;;
;; PROD denotes BCF0 # BCF1. The only absorptions possible     ;;
;; are (a) those within PROD and (b) absorptions of terms in   ;;
;; x'BCF0 or x BCF1 by terms in PROD.                          ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (bcf f)
  (cond ( (null? f) f)
        ( (null? (cdr f)) f)
        ( (member nil f) (list nil))
        ( else
          (let ((arg (opposed-arg f)))
               (cond ( (null? arg)
                       (unabsorb f) )
                     ( else
                       (let* ( (narg (bar arg))
                               (f0   (divide f narg))
                               (f1   (divide f arg))
                               (bcf0 (bcf f0))
                               (bcf1 (bcf f1))
                               (prod (muprod bcf0 bcf1))
                               (absprod (unabsorb prod))
                               (nf0  (absorb-rel bcf0 absprod))
                               (nf1  (absorb-rel bcf1 absprod)) )
                             (append (prefix narg nf0)
                                     (prefix arg  nf1)
                                     absprod ))))))))

(define (listbcf f)
  (list-terms (bcf f)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; BCF-SE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (BCF-SE F) returns BCF(F), making use of the method of         ;;
;; successive extraction.                                         ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (bcf-se f)
  (unabsorb
    (bcf-se2 f (opposed-args f)) ))

(define (bcf-se2 f arglist)
  (cond ( (null? arglist) f)
        ( else
          (bcf-se2
            (unabsorb
              (append f (yield f (car arglist))) )
            (cdr arglist) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; YIELD ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; The "yield" of a function with respect to an argument is        ;;
;; the set of consensuses formed by oppositions in that            ;;
;; argument. The factor-function permits factoring of the          ;;
;; function f with respect to an argument x: f0 is the             ;;
;; quotient wrt x' of the terms in which x' appears explicitly     ;;
;; and f1 is defined similarly for x.                              ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (yield f arg)
  (let* ( (narg (bar arg))
          (f0 (factor f narg))
          (f1 (factor f arg)) )
        (muprod f0 f1) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (MCF F LST) returns the minterm canonical form of the           ;;
;; function F with respect to the variables enumerated in the      ;;
;; list LST.  Two examples are shown below:                        ;;
;;                                                                  ;;
;;    [1] (show (mcf (parse "x1 z2'+ x2'z2 + x1'x2'z1 z2'")))       ;;
;;                                                                  ;;
;;    X1'X2'Z2    <-- The coefficient of the minterm X1'X2' is     ;;
;;    X1'X2'Z1        Z1 + Z2.                                      ;;
;;    X1 X2'                                                        ;;
;;    X1 X2 Z2'                                                     ;;
;;                                                                  ;;
;;    [2] (show (mcf '((a b (c) d (e)) (b c e) ((a) c (d)))         ;;
;;                   '(a b c) ))                                    ;;
;;    A'B'C D'                                                      ;;
;;    A'B C D'                                                      ;;
;;    A'B C E                                                       ;;
;;    A B C'D E'                                                    ;;
;;    A B C E                                                       ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (mcf f lst)
  (if (null? lst)
      f
      (let* ( (arg  (car lst))
              (narg (bar arg))
              (f0 (submin (divide f narg)))
              (f1 (submin (divide f arg))) )
            (append (prefix narg (mcf f0 (cdr lst)))
                    (prefix arg  (mcf f1 (cdr lst))) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;;          A B S O R P T I O N   O P E R A T O R S                           ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                            ;;
;; (UNABSORB F) returns a subformula of F that contains no                    ;;
;; terms that are absorbed by other terms in the subformula.                  ;;
;;                                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (unabsorb f)
  (cond ( (null? f) nil)
        ( else
          (insert-abs (car f)
                      (unabsorb (cdr f)) ))))

(define (insert-abs term f)             ; Insert a term into an
  (cond ( (null? f) (list term))        ; absorbed-out formula,
        ( (subset? term (car f))        ; and carry out all ab-
          (insert-abs term (cdr f)) )   ; sorptions on the result.
        ( (subset? (car f) term)
          f )
        ( else
          (cons (car f)
                (insert-abs term (cdr f)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; ABSORB-REL ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (ABSORB-REL F G) returns those terms of the SOP formula F     ;;
;; that are not absorbed by any term of the SOP formula G.       ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (absorb-rel f g)
  (cond ( (null? g) f)
        ( else
          (absorb-rel (term-absorb f (car g)) (cdr g)) )))

(define (term-absorb f term)
  (cond ( (null? f) nil)
        ( (subset? term (car f))
          (term-absorb (cdr f) term) )
        ( else
          (cons (car f) (term-absorb (cdr f) term)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;; ABSORBED? ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (ABSORBED? TERM F) is a predicate returning TRUE in case       ;;
;; TERM is absorbed by some term of the SOP formula F.            ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (absorbed? term f)
  (cond ((null? f) nil)
        ((subset? (car f) term) true)
        (else
         (absorbed? term (cdr f)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;          E L I M I N A T I O N   O P E R A T O R S                   ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ECON ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (ECON F TERM) returns the conjunctive eliminant of F with            ;;
;; respect to the arguments in TERM.                                    ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (econ f term)
  (simplify (econ2 f term)) )

(define (econ2 f term)
  (cond ((null? f) f)
        ((member nil f) (list nil))
        ((null? term) f)
        (else
         (let* ((arg (car term))
                (part (partition f arg))
                (p   (car part))
                (q   (cadr part))
                (r   (caddr part))
                (prod (mult p q)) )
               (econ2 (append prod r) (cdr term)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ECON-BCF ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (ECON-BCF F TERM) returns the conjunctive eliminant of F             ;;
;; with respect to the arguments in TERM, provided F is in              ;;
;; Blake canonical form.                                                ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (econ-bcf f term)
  (residue f term) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; RESIDUE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (RESIDUE F TERM) returns all of the terms of F that do not    ;;
;; contain any argument appearing in TERM. This function pro-    ;;
;; duces the conjunctive eliminant of F with respect to the      ;;
;; arguments in TERM in case f is in Blake canonical form.       ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (residue f term)
  (cond ( (null? f) nil)
        ( (common-args? (car f) term)
          (residue (cdr f) term) )
        ( else
          (cons (car f)
                (residue (cdr f) term) ))))

(define (common-args? term1 term2)
  (cond ( (null? term2) nil)
        ( (or (member (car term2) term1)
              (member (bar (car term2)) term1) )
          #t )
        ( (common-args? term1 (cdr term2))
          #t )
        ( else
          (writeln "Error in COMMON-ARGS? procedure") )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; PCON ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (PCON F ARGS) accepts a Blake canonical form, F, and a         ;;
;; list, ARGS, of arguments. The conjunctive projection of F      ;;
;; with respect to ARGS is returned. The terms of this            ;;
;; projection are the prime implicants of F that involve only     ;;
;; arguments belonging to ARG.                                    ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (pcon f args)     ;; f must be in BCF !
  (cond ( (null? f)
          '() )
        ( (subset? (depolarize-term (car f)) args)
          (cons (car f) (pcon (cdr f) args)) )
        ( else
          (pcon (cdr f) args) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; EDIS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;; (EDIS F TERM) returns the disjunctive eliminant of F with          ;;
;; respect to the arguments in TERM.                                  ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (edis f term)
  (cond ( (null? term) f)
        ( else
          (edis (replace-by-one f (car term))
                (cdr term) ))))

(define (replace-by-one f x)
  (cond ( (null? f) nil)
        ( else
          (cons (replace-term (car f) x)
                (replace-by-one (cdr f) x) ))))

(define (replace-term term x)
  (cond ( (null? term) nil)
        ( (or (equal? (car term) x)
              (equal? (car term) (bar x)) )
          (cdr term) )
        ( else
          (cons (car term)
                (replace-term (cdr term) x) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; PDIS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (PDIS F ARGS) accepts an SOP formula, F, and a list, ARGS,           ;;
;; of arguments. The disjunctive projection of F with respect           ;;
;; to ARGS is returned. Each term of F undergoes modification           ;;
;; to become a term of the returned projection. The modifica-           ;;
;; tion consists of removing all literals in the term that do           ;;
;; not belong to ARGS.                                                  ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pdis f args)
  (if (null? f)
      '()
      (cons (modify-term (car f) args)
            (pdis (cdr f) args) )))

(define (modify-term term args)
  (cond ( (null? term)
          '() )
        ( (member (debar (car term)) args)
          (cons (car term)
                (modify-term (cdr term) args) ))
        ( else
          (modify-term (cdr term) args) )))

;;;;;;;;;;;;;;;;;;;;;;;;;; PROJECT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (PROJECT FCN TERM) returns a function comprising those               ;;
;; terms in FCN that contain all the literals in TERM.                  ;;
;; Polarity of literals matters.                                        ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (project fcn term)
  (cond ( (null? fcn) nil)
        ( (subset? term (car fcn))
          (cons (car fcn)
                (project (cdr fcn) term) ))
        ( else
          (project (cdr fcn) term) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;          T A U T O L O G Y   C H E C K E R S                         ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; TAUT? ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; Given a Boolean SOP formula f, TAUT? checks to see if f              ;;
;; represents a tautology.                                              ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (taut? f)
  (cond ((member () f) true)
        ((null? f) nil)
        (else
         (taut? (econ f (list (first-arg f)))) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;; TAUT1? ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; This tautology checker uses Zakrevskii's algorithm.  It              ;;
;; appears to be about as efficient as TAUT?.                           ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (taut1? f)
  (newline)
  (list-terms f)
  (cond ((member () f)
         (princ "tautology")
         true)
        ((get-singleton f)
         (princ "singleton-term: ") (newline)
         (princ (car (get-singleton f))) (newline)
         (let ((arg (car (get-singleton f))))
              (taut1? (divide f (bar arg))) ))
        ((opposed-arg f)
         (princ "opposed argument: ")
         (princ (opposed-arg f))
```

```
        (let* ((x (opposed-arg f))
               (xbar (bar x))
               (f0 (divide f xbar))
               (f1 (divide f x)) )
              (and (taut1? f0) (taut1? f1)) ))
       (else  nil) ))

(define (get-singleton f)
  (cond ((null? f) nil)
        ((null? (cdar f)) (car f))
        (else (get-singleton (cdr f))) ))

(define (partition f x)
  (let*  ( (arg (debar x))
           (narg (bar arg)) )
         (partition1 f arg narg) ))

(define (partition1 f arg narg)
  (cond ((null? f) (list nil nil nil))
        ((member narg (car f))
         (let ((next (partition1 (cdr f) arg narg)))
              (cons (cons (remove narg (car f))
                          (car next) )
                    (cdr next) )))
        ((member arg (car f))
         (let ((next (partition1 (cdr f) arg narg)))
              (list (car next)
                    (cons (remove arg (car f))
                          (cadr next) )
                    (caddr next) )))
        (else
         (let ((next (partition1 (cdr f) arg narg)))
              (list (car next)
                    (cadr next)
                    (cons (car f)
                          (caddr next) ))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                    S E T   M A N I P U L A T I O N                       ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; SUBSET? ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; (SUBSET? T1 T2) returns TRUE in case every member of the                 ;;
;; list T1 is also a member of the list T2.                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (subset? t1 t2)
   (cond ((null? t1) true)
         ((and (member  (car t1) t2)
               (subset? (cdr t1) t2) ))
         (else nil) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; UNION ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; Returns the union of LIST1 and LIST2 assuming that LIST2                 ;;
;; has no duplicate elements.                                               ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (union list1 list2)
  (cond ( (null? list1) list2)
        ( (member (car list1) list2)
          (union (cdr list1) list2) )
        ( else
          (union (cdr list1) (cons (car list1) list2)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; DIFFERENCE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; (DIFFERENCE S T) returns the set S - T, i.e., every member               ;;
;; of list S that is not a member of list T.                                ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (difference s1 s2)
  (cond ( (null? s1)
          '() )
        ( (member (car s1) s2)
          (difference (cdr s1) s2) )
        ( else
          (cons (car s1)
                (difference (cdr s1) s2) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                          R A N G E S                                     ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; RANGE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; (RANGE F ARG) accepts an SOP formula F and an argument ARG.              ;;
;; The function returns the range (LO HI) bounding ARG, given               ;;
;; the equation F = 1. LO and HI are defined as follows:                    ;;
;;                                                                          ;;
;;      LO = (f/zi')' * (f/zi)                                              ;;
;;      HI = (f/zi')' + (f/zi)                                              ;;
;;                                                                          ;;
;; HI is returned in Blake canonical form, to enable sub-                   ;;
;; sequent conjuntive eliminants to be calculated efficiently.              ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (range f arg)
  (let* ( (farg** (complement (divide f (list arg))))
          (farg   (divide f arg))
          (lo     (simplify (mult farg** farg)))
          (hi     (bcf (add  farg** farg))) )
        (list lo hi) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;; ELIM-ARGS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (ELIM-ARGS RANGE ARGS) accepts a RANGE of the form (LO HI),   ;;
;; in which LO and HI are bounding SOP formulas and a list       ;;
;; ARGS of arguments. If the arguments in ARGS can be elimi-     ;;
;; nated, to produce a new range (NEW-LO NEW-HI), the new        ;;
;; range is returned; otherwise, '() is returned. The initial    ;;
;; upper bound, HI, must be in Blake canonical form.             ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (elim-args range args)
  (let ( (new-lo (simplify (edis (car range) args)))
         (new-hi (econ-bcf (cadr range) args)) )
       (if (formally-included? new-lo new-hi)
           (list new-lo new-hi)
           '() )))

(define (project-range range args)
  (let ( (new-lo (simplify (pdis (car range) args)))
         (new-hi (pcon (cadr range) args)) )
       (if (formally-included? new-lo new-hi)
           (list new-lo new-hi)
           '() )))

(define (get-range ckt output args)
  (project-range (range (complement
                          (parse ckt) )
                        output )
                 args ))

(define (show-lo range)
  (list-terms (car range)) )

(define (show-hi range)
  (list-terms (cadr range)) )

(define (show-range range)
   (princ "Lower bound:") (newline)
   (show-lo range)
   (princ "Upper bound:") (newline)
   (show-hi range) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;                    S I M P L I F I C A T I O N                       ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; SIMPLIFY ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; This procedure is a slight modification of BCF-ITER, which          ;;
;; implements the method of iterated consensus.  It differs            ;;
;; from BCF-ITER only in the procedure "consensus."  The idea          ;;
;; here is to generate only consensus-terms which absorb at            ;;
;; least one of their parents. The principal utility of this           ;;
;; procedure is to clean up functions like MULT and ECON               ;;
;; (which uses MULT) whose recursive definition causes them to         ;;
;; produce large numbers of terms that differ in only one              ;;
;; literal.                                                            ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (simplify f)
  (unabsorb
    (simplify2 nil f) ))

(define (simplify2 left right)
  (cond ( (null? right) left)
        ( (null? left)
          (simplify2 (list (car right))
                     (cdr right) ))
        ( (absorbed? (car right) left)
          (simplify2 left (cdr right) ))
        ( else
          (let ((newcons (sweep (car right) left)))
               (cond ( (equal? newcons '(()))
                       '(()) )
                     ( (equal? (car newcons) 'drop-term)
                       (simplify2 left
                                  (append (cadr newcons)
                                          (cdr right) )))
                     ( else
                       (simplify2 (cons (car right) left)
                                  (append (cadr newcons)
                                          (cdr right) ))))))))
```

```
(define (sweep term f)
  (sweep2 term nil f) )

(define (sweep2 term acc partf)
  (cond ( (null? partf) (list 'ok acc))
        ( else
          (let ((consens (consensus term (car partf))))
               (cond ( (null? consens)
                       (sweep2 term acc (cdr partf)) )
                     ( (equal? consens '(()))
                       '(()) )
                     ( (subset? consens term)
                       (list 'drop-term
                             (cons consens acc) ))
                     ( else
                       (sweep2 term (cons consens acc)
                                    (cdr partf) )))))))

;;;;;;;;;;;;;;;;;;;;;;;;;; CONSENSUS ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; p and q are terms. The consensus of p and q is returned      ;;
;; only if the consensus absorbs at least one of its parents.   ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (consensus p q)
  (let ( (consens (consensus2 0 p q)) )
       (cond ( (equal? consens 'no-dice)
               '() )
             ( (null? consens)
               '(()) )
             ( (or (subset? consens p)
                   (subset? consens q) )
               consens )
             ( else '() ))))
```

```
(define (consensus2 count p acc)
  (cond ( (= count 0)
          (cond ( (null? p)
                  'no-dice )
                ( (member (bar (car p)) acc)
                  (consensus2 1 (cdr p)
                              (remove (bar (car p)) acc) ))
                ( (member (car p) acc)
                  (consensus2 0 (cdr p) acc) )
                ( else
                  (consensus2 0 (cdr p) (cons (car p) acc)) )))
        ( else
          (cond ( (null? p)
                  acc )
                ( (member (bar (car p)) acc)
                  'no-dice )
                ( (member (car p) acc)
                  (consensus2 1 (cdr p) acc) )
                ( else
                  (consensus2 1 (cdr p) (cons (car p) acc)) )))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;;                  M I N I M I Z A T I O N                                 ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                          ;;
;; These functions are concerned with the minimization of SOP              ;;
;; formulas. One idea to pursue, discussed by Gaines, is to                ;;
;; look at implication-relations among the prime implicants of             ;;
;; a formula. We can do that by setting up a system of                     ;;
;; equations of the form A = first PI, B = second PI, etc.,                ;;
;; reducing the equations to the form F = 0, eliminating the               ;;
;; xyz's from that equation, and then isolating all terms                  ;;
;; having just a single unbarred literal, e.g. A'B C'D'. That              ;;
;; term can be given the interpretation                                    ;;
;; "B is included in A + C + D".                                           ;;
;;                                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;  GET-IMPLIERS  ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This function removes all terms from an SOP formula that      ;;
;; have other than one unbarred literal.                          ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-impliers f)
  (bcf (get-impliers1 f)) )

(define (get-impliers1 f)
  (cond ((null? f) nil)
        ((bad-barcount? (car f))
         (get-impliers1 (cdr f)) )
        (else
         (cons (car f)
               (get-impliers1 (cdr f)) ))))

(define (bad-barcount? term)
  (let ((count (count-unbars term)))
       (cond ((= count 1) nil)
             (else t) )))

(define (count-unbars term)
  (cond ((null? term) 0)
        ((atom? (car term))
         (+ 1 (count-unbars (cdr term))) )
        (else
         (count-unbars (cdr term)) )))

(define (irr-subsets poslubs)
  (cond ( (null? poslubs) nil)
        ( (null? (cdr poslubs)) (complement poslubs))
        ( else
          (unabsorb
            (expand-out (car poslubs)
                        (irr-subsets (cdr poslubs)) )))))

(define (expand-out term f)
  (cond ( (null? term) nil)
        ( else
          (append (expand-arg-f (car term) f)
                  (expand-out (cdr term) f) ))))
```

```
(define (expand-arg-f x f)
  (cond ( (null? f) nil)
        ( (member x (car f))
          (cons (car f)
                (expand-arg-f x (cdr f)) ))
        ( else
          (cons (cons x (car f))
                (expand-arg-f x (cdr f)) ))))

(define (pos-lubs f)
  (cond ( (null? f) nil)
        ( else
          (let ( (pos-term (pos-term-lub (car f))))
               (cond ( (null? pos-term)
                       (pos-lubs (cdr f)) )
                     ( else
                       (cons pos-term
                             (pos-lubs (cdr f)) )))))))

(define (pos-term-lub term)
  (cond ( (null? term) nil)
        ( (atom? (car term))
          (cons (car term)
                (pos-term-lub (cdr term)) ))
        ( else
          (pos-term-lub (cdr term)) )))

;;;;;;;;;;;;;;;;;;;;;;;; ALL-IRREDUNDANT ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (ALL-IRREDUNDANT SYS F ARGLIST) returns clauses of the form ;;
;; F ---> A + B + ... , where A, B, ... are names assigned to  ;;
;; prime implicants. SYS is the name of a system of equations  ;;
;; or inclusions defining the problem, F is the name of the    ;;
;; function and ARGLIST is a list of argument-names            ;;
;; (e.g., x, y, ...) to be eliminated. A typical form for      ;;
;; SYS is                                                      ;;
;;          ( (le ((f)) ((w (x) (y)) ((x) (y) z)) )           ;;
;;            (eq ((a1)) (((w) (x) z)) )                      ;;
;;            (eq ((a2)) ((w x z)) )                          ;;
;;            (eq ((a3)) (((y) z)) )                          ;;
;;            (eq ((a4)) ((w (x) (z))) )                      ;;
;;            (eq ((a5)) ((w (x) (y))) ) )                    ;;
```

```
;; in which the first clause expresses f =< lower bound (oddly ;;
;; enough) and a1,...,a5 are the prime implicants of the upper ;;
;; bound. The returned clauses correspond to the irredundant   ;;
;; formulas for F.                                             ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (all-irredundant sys f arglist)
  (list-clauses
    (match-antecedents (bcf (econ (reduce sys) arglist))
                       (list f) )))

;;;;;;;;;;;;;;;;;;;;;;;; MATCH-ANTECEDENTS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; (MATCH-ANTECEDENTS F TERM) returns all terms in F whose     ;;
;; positive sub-terms match TERM. This enables all clauses     ;;
;; to be returned whose left-hand sides are the same as TERM.  ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (match-antecedents f antecedent)
  (cond ( (null? f) nil)
        ( (equal-terms? (car (segregate (car f)))
                        antecedent )
          (cons (car f)
                (match-antecedents (cdr f) antecedent) ))
        ( else
          (match-antecedents (cdr f) antecedent) )))

;;;;;;;;;;;;;;;;;;;;;;;; SUBMINIMIZATION ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; (SUBMIN F) returns a sub-minimal formula representing the   ;;
;; SOP formula F.                                              ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (submin f)
  (make-irredundant (bcf f)) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;; MAKE-IRREDUNDANT ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (MAKE-IRREDUNDANT F) returns an irredundant subformula of       ;;
;; the SOP formula F.                                               ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (make-irredundant f)
  (make-irredundant* nil (biggest-first f)) )

(define (make-irredundant* front back)
;;;  (princ front) (newline) (princ back) (newline) (newline)
  (cond ( (null? back) front)
        ( (included-term? (car back)
                          (append front (cdr back)) )
          (make-irredundant* front (cdr back)) )
        ( else
          (make-irredundant* (cons (car back) front) (cdr back)) )))

(define (included-term? term f)
  (taut? (divide-by-term f term)) )

;;;;;;;;;;;;;;;;;;;;;;;; SUBMIN-INTERVAL ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (SUBMIN-INTERVAL RANGE) returns a subminimal formula for         ;;
;; a function in the interval specified by RANGE = (LO HI).         ;;
;; HI must be in Blake canonical form.                              ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (submin-interval range)            ;; HI MUST BE IN BCF!
  (submin-lohi (car range) (cadr range)) )

;;;;;;;;;;;;;;;;;;;;;;;;; SUBMIN-DC ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; (SUBMIN-DC DO-CARES DONT-CARES) specifies the interval by        ;;
;; "do-care" and "dont-care" formulas.                              ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (submin-dc do-cares dont-cares)
  (submin-lohi do-cares
               (add do-cares dont-cares) ))
```

```
(define (submin-lohi lo hi)
  (irr-inter* (complement lo)
              (biggest-first hi)
              nil ))

(define (irr-inter* lobar rem acc)
  (cond ( (null? rem) acc)
        ( (taut? (add (add lobar acc)
                      (cdr rem) ))
          (irr-inter* lobar (cdr rem) acc) )
        ( else
          (irr-inter* lobar (cdr rem) (cons (car rem) acc) ))))

(define (biggest-first f)
  (cond ( (null? f) nil)
        ( (null? (cdr f)) f)
        ( else
          (let ((sortcdr (biggest-first (cdr f))) )
               (cond ( (< (length (car f))
                          (length (car sortcdr)) )
                       (cons (car sortcdr)
                             (biggest-first
                                (cons (car f)
                                      (cdr sortcdr) ))))
                     ( else
                       (cons (car f)
                             sortcdr )))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;       M I S C E L L A N E O U S   F U N C T I O N S                  ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;; FORMALLY-INCLUDED ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; (FORMALLY-INCLUDED? G F) returns TRUE in case the SOP                ;;
;; formula G is formally included in the sop formula F, i.e.,           ;;
;; in case every term of G is a superset of some term in F. If          ;;
;; F is in Blake canonical form, then FORMALLY-INCLUDED? is             ;;
;; the same as INCLUDED?.                                               ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (formally-included? g f)
  (define (term-included? term f)
    (if (null? f)
        '()
        (or (subset? (car f) term)
            (term-included? term (cdr f)) )))
  (or (null? g)
      (and (term-included?     (car g) f)
           (formally-included? (cdr g) f) )))

(define (equal-terms? term1 term2)
   (and (subset? term1 term2)
        (subset? term2 term1) ))

(define (flatten lst)
  (cond ( (null? lst) nil)
        ( (atom? (car lst))
          (cons (car lst)
                (flatten (cdr lst)) ))
        ( else
          (append (flatten (car lst))
                  (flatten (cdr lst)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; FIND-ARGS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (FIND-ARGS F) returns a sorted list of all of the             ;;
;; arguments appearing in the SOP formula F.                      ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (find-args f)
  (sort-term
    (undup
      (flatten f) )))

(define (beep)
  (princ (ascii->symbol 7)) )

;;;;;;;;;;;;;;;;;;;;;;;; DEPOLARIZE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (DEPOLARIZE F) returns F, with all complemented literals       ;;
;; uncomplemented. Example:                                       ;;
;;                                                                ;;
;;      [1] (depolarize '((a (b) d) ((c) (d) f) (e (f))))         ;;
;;                                                                ;;
;;      ((A B D) (C D F) (E F))                                   ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (depolarize f)
  (cond ( (null? f) nil)
        ( else
          (cons (depolarize-term (car f))
                (depolarize (cdr f)) ))))

(define (depolarize-term term)
  (cond ( (null? term) nil)
        ( (atom? (car term))
          (cons (car term)
                (depolarize-term (cdr term)) ))
        ( else
          (cons (caar term)
                (depolarize-term (cdr term)) ))))
```

```
(define (debar literal)
  (cond ((atom? literal) literal)
        (else (bar literal)) ))

(define (opposed-args f)
  (list-opposed (get-literals f)) )

;;;;;;;;;;;;;;;;;;;;;;;;; LIST-OPPOSED ;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (LIST-OPPOSED LST) operates on a list of literals, presumed ;;
;; to have no duplicates, returning a list of unbarred         ;;
;; variables, one variable for each each opposed pair in the   ;;
;; original list.                                               ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (list-opposed lst)
  (cond ( (null? lst) nil)
        ( (member (bar (car lst)) (cdr lst))
                  (cons (debar (car lst))
                        (list-opposed (cdr lst)) ))
        ( else
          (list-opposed (cdr lst)) )))

;;;;;;;;;;;;;;;;;;;;;;;;; OPPOSED-ARG ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (OPPOSED-ARG F) returns an argument that is opposed in F,   ;;
;; if one exists, otherwise, it returns nil. An argument is    ;;
;; opposed in F if the argument appears uncomplemented in one  ;;
;; term of F and complemented in another.                       ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (opposed-arg f)
  (cond ( (null? f) nil)
        ( (null? (cdr f)) nil)
        ( (member nil f) nil)
        ( else
          (make-letter
            (seek-opposed (get-literals f)) ))))
```

```
(define (seek-opposed args)
  (cond ( (null? args) nil)
        ( (null? (cdr args)) nil)
        ( (member (bar (car args)) (cdr args))
          (car args) )
        ( else
          (seek-opposed (cdr args)) )))

(define (make-letter literal)
  (cond ( (atom? literal) literal)
        ( else
          (bar literal) )))

;;;;;;;;;;;;;;;;;;;;;;;;;; GET-LITERALS ;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (GET-LITERALS F) collects in a list all of the literals        ;;
;; explicit in SOP formula F. Duplicates are excluded.            ;;
;; An example is shown below:                                     ;;
;;                                                                ;;
;;    [1] (get-literals '((a (b) d) ((c) (d) f) (e (f))))         ;;
;;                                                                ;;
;;    (D (B) A F (D) (C) (F) 2)                                   ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-literals f)
  (cond ( (null? f) nil)
        ( else
          (union (car f) (get-literals (cdr f))) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; FIRST-ARG ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Return, uncomplemented, the first argument encountered in      ;;
;; the function F.                                                ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (first-arg f)
  (cond ((null? f) nil)
        ((member nil f) nil)
        (else
         (debar (car (car f))) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; GET-ARGS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; Return a list of all of the arguments in F.                      ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (get-args f)
  (undup (flatten f)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; OTHER-ARGS ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; Returns a list of all of the arguments in F that do not          ;;
;; belong to the set ARGS.                                          ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (other-args f args)
  (difference (get-args f) args) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;; REMOVE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; Remove one occurrence of element X from list LST.                ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (remove x lst)
  (cond ((null? lst) nil)
        ((equal? (car lst) x)
         (cdr lst))
        (else
          (con  ar lst)
                (remove x (cdr lst)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; UNDUP ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; Remove duplicate elements from a list.                           ;;
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (undup lst)
  (cond ( (null? lst)
          '() )
        ( (member (car lst) (cdr lst))
          (undup (cdr lst)) )
        ( else
          (cons (car lst) (undup (cdr lst))) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; PREFIX ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Prefix each term in the formula f by the literal x.           ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (prefix x f)
  (cond ((null? f) nil)
        (else
          (cons (cons x (car f))
                (prefix x (cdr f)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; BAR ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Complement (bar) a literal x.  Thus (BAR TOM) returns (TOM)    ;;
;; and (BAR (TOM)) returns TOM.                                   ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (bar x)
  (cond ((atom? x) (list x))
        (else (car x)) ))

;;;;;;;;;;;;;;;;;;;;;;;;;; LABEL-AND-REDUCE ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (LABEL-AND-REDUCE F) sets each term of F equal to a label,     ;;
;; using GENSYM, and reduces the resulting system of equations.   ;;
;; An example is shown below:                                     ;;
;;                                                                ;;
;;    [1] (label-and-reduce '((a b) ((b) c)))                     ;;
;;                                                                ;;
;;    ( ((G0) A B)    (G0 (A))      (G0 A (B))                    ;;
;;      ((G1) (B) C)  (G1 (B) (C))  (G1 B)      )                 ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (label-and-reduce f)
  (cond ( (null? f) nil)
        ( else
          (append (xor (list (list (gensym))) (list (car f)))
                  (label-and-reduce (cdr f)) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;; SUBSTITUTE ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (SUBSTITUTE F X G) substitutes the function f for the         ;;
;; argument x in the function g.                                  ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (substitute f x g) :-
  (econ (append (xor f (list (list x))) g) (list x)) )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;;              D I S P L A Y   F O R M U L A                     ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; The following procedures provide a variety of display          ;;
;; formats for SOP formulas.                                      ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; SHOW ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (SHOW F) produces a display, listed vertically, of the         ;;
;; terms in the SOP formula F. The argument-names are sorted      ;;
;; in each term.                                                  ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (show fcn)
  (list-terms fcn) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;; SHOW-H ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (SHOW-H F) produces a horizontal display of the terms in      ;;
;; the SOP formula F. The terms are connected with plus-signs.   ;;
;; As in SHOW, the argument-names are sorted in each term.       ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (show-h fcn)
  (cond ( (member nil fcn)
          (princ "1") (newline) )
        ( (null? fcn)
          (princ "0") (newline) )
        ( else
          (show-h-aux fcn) )))

(define (show-h-aux fcn)
  (cond ( (null? fcn)
          (newline) )
        ( (null? (cdr fcn))
          (write-term (sort-term (car fcn)))
          (newline) )
        (else
          (write-term (sort-term (car fcn)))
          (princ "+ ")
          (show-h-aux (cdr fcn)) )))

(define (list-terms fcn)
  (cond ( (member nil fcn)
          (princ "1") (newline) )
        ( (null? fcn)
          (princ "0") (newline) )
        ( else
          (list-terms-aux fcn) )))

(define (list-terms-aux fcn)
  (cond ( (null? fcn)
          (newline) )
        (else
            (write-term (sort-term (car fcn)))
            (newline)
            (list-terms-aux (cdr fcn)) )))
```

```
(define (write-term term)
  (cond ( (null? term)
          '() )
        ( (atom? (car term))
          (princ (car term)) (princ " ")
          (write-term (cdr term)) )
        ( else
          (princ (car (car term))) (princ "'")
          (write-term (cdr term)) )))

(define (sort-term term)
  (cond ((null? term) nil)
        ((null? (cdr term)) term)
        (else
          (let ((sort-cdr (sort-term (cdr term))))
               (cond ((lower-literal? (car term)
                                      (car sort-cdr) )
                      (cons (car term) sort-cdr) )
                     (else
                      (cons (car sort-cdr)
                            (sort-term (cons (car term)
                                             (cdr sort-cdr)
                                             ))))))))))

(define (lower-literal? x y)
  (lower-symbol? (debar x) (debar y)) )

(define (lower-symbol? x y)
  (let ((stringx (symbol->string x))
        (stringy (symbol->string y)) )
       (cond ( (string<? stringx stringy) true)
             (else nil) )))

(define (list-clauses fcn)
  (cond ((member nil fcn) (list nil) )
        ((null? fcn)

            (newline))
        (else
            (write-clause  (car fcn))
            (list-clauses  (cdr fcn)) )))
```

```
(define (write-clause term)
  (let ((partition (segregate term)))
       (write-lhs (sort-term (car partition)))
       (princ " --->  ")
       (write-rhs (sort-term (cadr partition)))
       (newline) ))

(define (write-lhs term)
  (cond ((null? term) (princ "1 "))
        (else (write-leftargs term)) ))

(define (write-rhs term)
  (cond ((null? term) (princ "0"))
        (else (write-rightargs term)) ))

(define (write-leftargs term)
  (cond ( (null? term) t)
        ( else (princ (car term))
               (princ " ")
               (write-leftargs (cdr term)) )))

(define (write-rightargs term)
  (cond ( (null? term) t)
        ( (null? (cdr term))
          (princ (car term)) )
        ( else (princ (car term))
               (princ " + ")
               (write-rightargs (cdr term)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;; SEGREGATE ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; (SEGREGATE TERM) returns a set of the form (POS NEG), in      ;;
;; which POS is a list comprising all of the uncomplemented      ;;
;; variables in TERM and NEG comprises all of the comple-        ;;
;; mented variables. Both POS and NEG consist of uncomple-       ;;
;; mented literals.                                              ;;
;;                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (segregate term)
  (cond ( (null? term)
          (list nil nil) )
        ( (atom? (car term))
          (list (cons (car term)
                      (car (segregate (cdr term))) )
                (cadr (segregate (cdr term))) ))
        ( else
          (list (car (segregate (cdr term)))
                (cons (caar term)
                      (cadr (segregate (cdr term))) )))))
```

B.9 NEW_DSGN.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;;    DESIGN SYSTEM WITH A MODIFIED COST CALCULATION PROCESS   ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; The current optimizations system DESIGN.S calculates all of ;;
;; the costs of the MDSs before proceeding with the search.   ;;
;; If the search process arrives at a solution before a given ;;
;; node is examined, the effort to calculate the cost of the  ;;
;; MDS corresponding to that node is wasted. This file        ;;
;; contains slightly modified algorithms from the DESIGN.S    ;;
;; file and the SEARCH.S file.  They contribute to a modified ;;
;; optimization process that delays the cost calculations     ;;
;; until a given node is examined.  Once the cost for a given ;;
;; MDS is calculated, it is stored using a MEMOIZE procedure  ;;
;; so that it will not have to be calculated again.           ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; FILE NAME:        NEW_DSGN.S or NEW_DSGN.FSL               ;;
;;                                                            ;;
;; DESCRIPTION:      A Modified Circuit Optimization System   ;;
;;                                                            ;;
;; AUTHOR:           Eric J. Knutson                          ;;
;;                                                            ;;
;; DATE:             7 NOV 90                                 ;;
;;                                                            ;;
;; AUXILLARY FILES:  From the BORIS System Software           ;;
;;                                                            ;;
;;                   TABULAR.FSL       SEARCH.FSL             ;;
;;                   PARSE.FSL         TOOLS.FSL              ;;
;;                   MDS.FSL           COST.FSL               ;;
;;                   DESIGN.FSL        DATA.S                 ;;
```

```
;; GETTING STARTED:  To get started, load NEW_DSGN.FSL and all ;;
;;                   of the auxiliary files at the PC Scheme   ;;
;;                   System prompt.  Then follow the           ;;
;;                   instructions and/or examples provided     ;;
;;                   with some of the algorithms below.        ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; NEW-DESIGN ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; The optimization of a specific circuit is initiated by an   ;;
;; input of the form (NEW-DESIGN CIRCUIT OUTPUTS) where         ;;
;; CIRCUIT represents the circuit specification and OUTPUTS    ;;
;; represents the designated circuit outputs.  An example is   ;;
;; shown below:                                                ;;
;;                                                             ;;
;; [1] (new-design CKT1 '(f g h))                              ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (new-design circuit outputs)
  (define (new-design-fcn circuit f outputs)
    (newline) (princ "Function:") (newline)
    (list-terms f)
    (newline) (princ "* Calculating The Range For Each Output")
    (newline) (newline)
    (store-ranges f outputs)
    (let ( (mds (out-mds-lists1 f outputs)) )
         (solve1 '((0 ())) mds outputs 1000) ))
  (newline)
  (princ "* Parsing and Reduction of Specification")
  (newline) (newline)
    (let ( (spec (simplify (complement (parse-design circuit)))))
      (princ "* Checking To See If Specification Is Tabular:  ")
      (if (tabular-spec? spec outputs)
          (begin (princ "PASSED!") (newline)
                 (new-design-fcn circuit spec outputs) )
          (begin (princ "FAILED!") (newline) (newline)
                 (princ "* Converting To A Tabular Form.")
                 (newline)
                 (new-design-fcn circuit
                                 (make-tabular-spec spec outputs)
                                 outputs) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; STORE-RANGES ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; This is an auxiliary procedure, called by NEW_DESIGN, that  ;;
;; stores a range corresponding to each of the outputs.  This   ;;
;; range is used to find a reduced, SOP formula composed soley ;;
;; of arguments contained in the MDS.  From this formula, the   ;;
;; cost associated with each MDS can be determined.  The        ;;
;; procedure is called by (STORE-RANGES F OUTPUTS)              ;;
;; where F is the parsed specification of the circuit in        ;;
;; normal form (F = 1) and OUTPUTS represents the designated    ;;
;; outputs.                                                     ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (store-ranges f outputs)
  (cond ( (null? outputs) )
        (else
          (memo-range f (car outputs))
          (store-ranges f (cdr outputs)) )))

;;;;;;;;;;;;;;;;;;;; MEMO-RANGE & MEMOIZE ;;;;;;;;;;;;;;;;;;;;;;
;;                                                              ;;
;; (MEMO-RANGE F Z) calculates the range of an output Z, given  ;;
;; a parsed specification F in normal form.  MEMO-RANGE first   ;;
;; checks through a table to see if that range has already      ;;
;; been calculated for the given output Z.  If it has, it       ;;
;; simply retrives it from a table.  If it has not, it          ;;
;; calculates it.                                               ;;
;;                                                              ;;
;; MEMOIZE accepts a procedure PROC as an argument and returns  ;;
;; another procedure which is a memoized version of PROC.       ;;
;; In this case MEMOIZE is used to create a memoized version    ;;
;; of RANGE.  This technique is described in "Scheme and the    ;;
;; Art of Programming," by G. Springer and D. Friedman (93).    ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (memoize proc)
  (define lookup
    (lambda (obj table success-proc failure-proc)
      (letrec ((lookup (lambda (table)
                  (if (null? table)
                      (failure-proc)
                      (let ((pr (car table)))
                        (if (equal? (car pr) obj)
                            (success-proc pr)
                            (lookup (cdr table)) ))))))
        (lookup table))))
  (let ((table '()))
    (lambda (function arg)
      (lookup arg
              table
              cdr
              (lambda ()
                (let ((val (proc function arg)))
                  (set! table (cons (cons arg val)
                                    table)) val ))))))

(define memo-range (memoize range))

;;;;;;;;;;;;;;;;;;;;;;;; OUT-MDS-LISTS1 ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This is identical to the OUT-MDS-LIST procedure found in       ;;
;; DESIGN.S, with one noteable exception;  the SOP formula        ;;
;; composed from the arguments in the MDS and the associated      ;;
;; cost is not calculated at this point.  Using the same          ;;
;; example used by OUT-MDS-LIST, this procedure returns           ;;
;;                                                                ;;
;;  [1] (out-mds-lists1 '(((f) b a g) (f (a) (g)) ((b) f (g))) ;;
;;                      '(f g) )                                  ;;
;;  ((F 0 () G) (F 0 () A B) (G 0 () F) (F 0 () A B)) .           ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (out-mds-lists1 f outputs)
  (define (out-mds-lists-aux1 f outputs)
    (cond ( (null? outputs) nil)
          ( else
            (let* ( (z          (car outputs))
                    (arg-lists  (min-determining f z))
                    (arg-lists* (zero-costs arg-lists)) )
                  (princ z) (princ "   ")
                  (princ arg-lists) (newline)
                  (append
                    (splice z arg-lists*)
                    (out-mds-lists-aux1 f (cdr outputs)) )))))
  (princ "Minimal Determining Subsets:") (newline)
  (out-mds-lists-aux1 f outputs) )

;;;;;;;;;;;;;;;;;;;;;;;;; ZERO-COSTS ;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; This is an auxiliary procedure called by OUT-MDS-LISTS1     ;;
;; that accepts a list of MDSs and returns a list with the     ;;
;; SOP formula and cost initialized.  For example:             ;;
;;                                                             ;;
;;    [1] (zero-costs '((g) (a b)) )                           ;;
;;    ((0 () G) (0 () A B))                                    ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (zero-costs arg-lists)
  (if (null? arg-lists)
      '()
      (cons (cons 0 (cons '() (car arg-lists)))
            (zero-costs (cdr arg-lists)) )))

;;;;;;;;;;;;;;;;;;;; SOLVE1 and SOLVE-CYCLE1 ;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;; These procedures are identical to SOLVE and SOLVE-CYCLE     ;;
;; found in SEARCH.S.                                          ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (solve1 queue mds outputs maxcost)
  (cond ( (null? queue)
          (newline)
           'fail )
        ( (and (subset? outputs (cadar queue))
               (= maxcost 1000) )
          (newline) (newline) (print-assignment (car queue))
          (newline) (print-simplified-fcns (cddar queue))
          (solve-cycle1 queue mds outputs (caar queue)) )
        ( (and (not (subset? outputs (cadar queue)))
               (< maxcost 1000) )
          (solve-cycle1 queue mds outputs maxcost) )
        ( (and (subset? outputs (cadar queue))
               (= maxcost (caar queue)) )
          (newline) (print-assignment (car queue))
          (solve-cycle1 queue mds outputs (caar queue)) )
        ( (< maxcost (caar queue))
          (newline)
          'done )
        ( else
          (newline) (print-assignment (car queue))
          (solve-cycle1 queue mds outputs 1000) )))

(define (solve-cycle1 queue mds outputs maxcost)
  (let* ( (mother (car queue))
          (kids (collect-children1 mother mds outputs)) )
        (solve1
          (best-first
            (insert-kids kids (cdr queue)) )
          mds
          outputs
          maxcost )))

;;;;;;;;;;;;;;;;;;;;;; COLLECT-CHILDREN1 ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; This procedure is similiar to COLLECT-CHILDREN found in the    ;;
;; SEARCH.S file.  It differs in that, when the children are      ;;
;; collected, minimized SOP formulas and a corresponding          ;;
;; cost are assigned to each.                                     ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (collect-children1 assgn mds outputs)
  (let* ( (z       (caar mds))
          (child   (car mds))
          (one-mds (cdddar mds)) )
    (cond ( (null? mds) nil)
          ( (not  (illegal-child? child assgn outputs))
            (cons (extended-assgn
                    (memo-child-cost z one-mds) assgn)
                  (collect-children1 assgn (cdr mds) outputs) ))
          ( else
            (collect-children1 assgn (cdr mds) outputs) ))))

;;;;;;;;;;;;;;;;;;;;;;;;;; CHILD-COST ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                            ;;
;; This is the procedure that actually calculates the SOP     ;;
;; formula and cost associated with each MDS.  It accepts as  ;;
;; its inputs an output Z and a MDS.  Given Z, it retrieves   ;;
;; the previously stored range that corresponds to that output.;;
;; With the range, MDS and Z, all of the necessary informa-   ;;
;; tion can be calculated.  An example is shown below:        ;;
;;                                                            ;;
;;   [1] (child-cost 'f '(a b))                               ;;
;;   (F 2 (((B)) ((A))) A B)                                  ;;
;;                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (child-cost z one-mds)
  (define (node-cost range arg)
    (let* ( (new-range   (project-range range arg))
            (min-formula (submin-interval new-range))
            (new-cost    (gate-input-cost min-formula)) )
          (cons new-cost
                (cons min-formula
                      arg ))))
  (cons z (node-cost (memo-range '() z) one-mds )))
```

```
;;;;;;;;;;;;;;;; MEMO-CHILD-COST & MEMOIZE1 ;;;;;;;;;;;;;;;;;;;
;;                                                               ;;
;; (MEMO-CHILD-COST Z ONE-MDS) checks through a table to see    ;;
;; if the SOP formula and cost have already been calculated     ;;
;; for a node given by an output Z and a minimal determining    ;;
;; subset ONE-MDS.  If the information exists, it retrieves     ;;
;; the data.  If the information does not exist, it calls the   ;;
;; procedure CHILD-COST to calculate the SOP formula and cost   ;;
;; associated with a given MDS.  This information is then       ;;
;; stored and returned.                                         ;;
;;                                                              ;;
;; As with MEMOIZE, MEMOIZE1 accepts a prcedure PROC as an      ;;
;; argument and returns another procedure which is a memoized   ;;
;; version of PROC.  In this case MEMOIZE1 is used to produce   ;;
;; memoized version of CHILD-COST.                              ;;
;;                                                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (memoize1 proc)
  (define lookup1
    (lambda (obj set table success-proc failure-proc)
      (letrec ((lookup1 (lambda (table)
                  (if (null? table)
                      (failure-proc)
                      (let ((pr (car table)))
                        (if (and (equal? (car pr) obj)
                                 (equal? (cadr pr) set) )
                          (success-proc pr)
                          (lookup1 (cdr table)) ))))))
        (lookup1 table))))
  (let ((table '()))
    (lambda (arg set)
      (lookup1 arg set table cddr
        (lambda ()
          (let ((val (proc arg set)))
            (set! table (cons (cons arg (cons set val))
                              table)) val ))))))

(define memo-child-cost (memoize1 child-cost))
```

B.10 NON_MDS.S File

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;        NON-MINIMAL DETERMINING SUBSET DESIGN SYSTEM         ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;  This file contains some modified design procedures that    ;;
;;  enable one to optimize a design using NON-MINIMAL Deter-   ;;
;;  mining Subsets. It has been shown that the use of minimal  ;;
;;  determining subsets does not always produce an optimal     ;;
;;  solution.  These modifications cause the introduction of   ;;
;;  additional outputs into the minimal determining subsets.   ;;
;;  The hope is that the introduction of these outputs will    ;;
;;  lead one to a better solution.  The drawback is that       ;;
;;  because it introduces additional modified subsets, the     ;;
;;  design is slowed down considerably.                        ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;; PROGRAM DETAILS ;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;  FILE NAME:        NON_MDS.S or NON_MDS.FSL                 ;;
;;                                                             ;;
;;  DESCRIPTION:      Circuit Optimization System Using        ;;
;;                    Non-Minimal Determining Subsets          ;;
;;                                                             ;;
;;  AUTHOR:           Eric J. Knutson                          ;;
;;                                                             ;;
;;  DATE:             28 SEP 90                                ;;
;;                                                             ;;
;;  AUXILLARY FILES: From the BORIS System Software            ;;
;;                                                             ;;
;;                    TOOLS.FSL     COST.FSL                   ;;
;;                    DESIGN.FSL    SEARCH.FSL                 ;;
;;                    PARSE.FSL     DATA.S                     ;;
;;                    MDS.FSL       TABULAR.FSL                ;;
```

```
;;  GETTING STARTED: To get started, load non_mds.s and all of ;;
;;                   the auxillary files at the PC Scheme      ;;
;;                   System Prompt.  Then follow the examples  ;;
;;                   and guidance provided below.              ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;; NON-MDS-DESIGN ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                             ;;
;;  (NON-MDS-DESIGN CIRCUIT OUTPUTS) finds the least-cost      ;;
;;  assignment of arguments to the outputs listed in OUTPUTS.  ;;
;;  CIRCUIT is a system of equations that specify the desired  ;;
;;  behavior of the circuit.  As an example, the input format  ;;
;;  for a logic circuit is shown below:                        ;;
;;                                                             ;;
;;  [1] (non-mds-design '("w = a p + q"                        ;;
;;                        "x = a p"                            ;;
;;                        "y = p a' + r"                       ;;
;;                        "z = q")        '(w x y z) )         ;;
;;                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (non-mds-design circuit outputs)
  (define (design-fcn1 f outputs)
    (newline) (princ "Function:") (newline)
    (list-terms f)
    (let ( (mds (out-mds-lists1 f outputs)) )
         (solve '((0 ())) mds outputs 1000) ))
  (newline)
  (princ "* Parsing Specification and Reducing to Normal Form")
  (newline) (newline)
    (let ( (spec (simplify (complement (parse-design circuit)))))
      (princ "* Checking To See If Specification Is Tabular:  ")
      (if (tabular-spec? spec outputs)
          (begin
            (princ "PASSED!") (newline)
            (design-fcn1 spec outputs) )
          (begin
            (princ "FAILED!") (newline) (newline)
            (princ "* Converting To A Tabular Form.")
            (newline)
            (design-fcn1 (make-tabular-spec spec outputs)
                         outputs) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;; OUT-MDS-LISTS1 ;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                ;;
;; Given a parsed specification in normal form (F = 1),          ;;
;; (OUT-MDS-LISTS F OUTPUTS) finds the minimal determining        ;;
;; subsets, and associated cost, for each of the ouputs in        ;;
;; OUTPUTS.  An example is shown below:                           ;;
;;                                                                ;;
;;  [1] (out-mds-lists1 '(((f) b a g) (f (a) (g)) ((b) f (g)))  ;;
;;                      '(f g) )                                  ;;
;;  ( (F 1 (((G))) G) (F 1 (((G))) A B G)                         ;;
;;    (F 2 (((B)) ((A))) A B) (G 1 (((F))) F)                     ;;
;;    (G 1 (((F))) A B F) (G 2 ((A B)) A B) )                     ;;
;;                                                                ;;
;; The format of the output is                                    ;;
;;                                                                ;;
;;   ( (OUTPUT COST FORM MDS) (OUTPUT COST FORM MDS) ... )        ;;
;;                                                                ;;
;; where OUTPUT represents an argument found in OUTPUTS, MDS      ;;
;; represents the arguments of one of OUTPUT's minimal            ;;
;; determining subsets, FORM represents a minimal, SOP formula    ;;
;; that produces OUTPUT using the arguments found in MDS, and     ;;
;; COST represents the gate-input cost associated with FORM.      ;;
;;                                                                ;;
;; This function will only display the MDSs corresponding to a    ;;
;; given output.  To display all of the information concerning    ;;
;; MDSs, including their associated cost and SOP formulas,        ;;
;; place a semi-colon in front of the OPT1 lines below and        ;;
;; remove the semi-colon from in front of the OPT2 line.          ;;
;;                                                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (out-mds-lists1 f outputs)
  (define (out-mds-lists-aux1 f outputs outsave)
    (cond ( (null? outputs) nil)
          ( else
            (let* ( (z          (car outputs))
                    (mds-lists  (min-determining f z))
                    (new-lists  (undup (mds-expand mds-lists z
                                                   outsave)))
                    (mds-lists* (attach-costs f z new-lists)) )
                  (princ z) (princ "  ")                ;; OPT1
                  (princ new-lists) (newline)           ;; OPT1
                ; (display-cost new-lists* z) (newline) ;; OPT2
                  (append
                    (splice z mds-lists*)
                    (out-mds-lists-aux1 f (cdr outputs)
                                        outsave) )))))
  (princ "Minimal Determining Subsets:") (newline)
  (out-mds-lists-aux1 f outputs outputs) )

;;;;;;;;;;;;;;;;;;;;;;;; MDS-EXPAND ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                  ;;
;; The auxillary procedure (MDS-EXPAND MDS Z OUTPUTS) accepts       ;;
;; a list MDS of minimal determining subsets for the output Z       ;;
;; and the list of OUTPUTS for the circuit.  It returns an          ;;
;; expanded set of subsets which are no longer necessarily          ;;
;; minimal.  New sets are created by adding one additional          ;;
;; output.  An example is illustrated below:                        ;;
;;                                                                  ;;
;; [1] (mds-expand '((X1 X2 X3) (X2 X3 Z1)) 'Z3 '(Z1 Z2 Z3))       ;;
;;                                                                  ;;
;; ((X1 X2 X3 Z1)(X1 X2 X3 Z2)(X1 X2 X3)(X2 X3 Z1 Z2)(X2 X3 Z1));
;;                                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define (mds-expand mds z outputs)
  (define (mds-expand-aux mds outputs new-ds)
    (define (make-new-ds minset outputs result)
      (cond ( (null? outputs)
              (append result (list minset)) )
            ( (member (car outputs) minset)
              (make-new-ds minset (cdr outputs) result) )
           (else
              (make-new-ds minset (cdr outputs)
                 (append result
                         (list (sort-term (cons (car outputs)
                                                minset)))) ))))
    (cond ( (null? mds) new-ds)
          (else
            (append new-ds minset)
            (mds-expand-aux
              (cdr mds)
              outputs
              (append new-ds
                      (make-new-ds (car mds) outputs '())) ))))
  (mds-expand-aux mds (remove z outputs) '()) )
```

Bibliography

1. Arevalo, Z. and J.G. Bredeson, "A Method to Simplify a Boolean Function Into a Near Minimal Sum-Of-Products for programmable logic arrays," *IEEE Transactions on Computers, C-27*: 1028-1039 (November 1978).
2. Bartlett, Karen A. and Gary D. Hachtel. "Library Specific Optimization of Multilevel Combinational Logic," *Proceedings of the IEEE International Conference on Computer Design.* 411-415. Washington, D.C.: IEEE Computer Society Press, 1985.
3. Bartlett, Karen and others. "Multilevel Logic Minimization Using Implicit Don't Cares," *IEEE Transactions on Computer Aided Design, 7*: 723-740 (June 1988).
4. Bartlett, Karen and others. "Synthesis and Optimization of Multi-Level Logic Under Timing Constraints," *IEEE Transactions on Computer-Aided Design, CAD-5*: 582-596 (October 1986).
5. Bartlett, Karen and others. "Synthesis and Optimization of Multi-Level Logic Under Timing Constraints," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 290-292. Washington, D.C.: IEEE Computer Society Press, 1985.
6. Berman, Leonard. "Applications of Global Flow Analysis in Logic Synthesis," *1988 International Symposium on Circuits and Systems.* 901-904. Washington, D.C.: IEEE Computer Society Press, 1988.
7. Berman, Leonard and Louise Trevillyan. "Improved Logic Optimization Using Global Flow Analysis," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 102-105. Washington, D.C.: IEEE Computer Society Press, 1988.
8. Birmingham, William P. and Jin H. Kim. "DAS/Logic: A Rule-Based Logic Design Assistant," *The Second conference on AI Applications, IEEE Computer Society.* 264-269. Washington, D.C.: IEEE Computer Society Press, 1985.
9. Bostick, D. and others. "The Boulder Optimal Logic Design System," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 62-65. Washington, D.C.: IEEE Computer Society Press, 1987.
10. Brayton, R.K. and F. Somenzi "An Exact Minimizer for Boolean Relations," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 316-319. Washington, D.C.: IEEE Computer Society Press, 1987.
11. Brayton, R.K., E.M. Sentovich and F. Somenzi. "Don't Cares and Global Flow Analysis of Boolean Networks," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 98-101. Washington, D.C.: IEEE Computer Society Press, 1988.
12. Brayton, Robert K. and others. *Logic Minimization Algorithms for VLSI Synthesis.* Hingham, MA: Kluwer Academic Publishers, 1984.
13. Brayton, Robert K. "Minimization of Boolean Relations," *International Symposium on Circuits and Systems.* 738-743. Washington, D.C.: IEEE Computer Society Press, 1989.

14. Brayton, Robert K. and others. "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on Computer-Aided Design, CAD-6*: 1062-1081 (November 1987).

15. Brayton, Robert K. and others. "Multi-Level Logic Optimization and The Rectangular Covering Problem," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 66-69. Washington, D.C.: IEEE Computer Society Press, 1987.

16. Brayton, Robert K. and others. "Multiple-Level Logic Optimization System," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 356-359. Washington, D.C.: IEEE Computer Society Press, 1986.

17. Brayton, R.K., G.D. Hachtel and A.L. Sangiovanni-Vincentelli. "Multilevel Logic Synthesis," *Proceedings of the IEEE, 78*: 264-300 (February 1990).

18. Brayton, Robert K. and Curt McMullen. "Synthesis and Optimization of Multistage Logic," *Proceedings of the IEEE International Symposium on Circuits and Systems.* 23-28. Washington, D.C.: IEEE Computer Society Press, 1982.

19. Brayton, Robert K. and Curt McMullen. "The Decomposition and Factorization of Boolean Expressions," *Proceedings of the IEEE International Symposium on Circuits and Systems.* 49-54. Washington, D.C.: IEEE Computer Society Press, 1982.

20. Bryant, Randal E. "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers, 8*: 677-689. (August 1986).

21. Brown, Douglas W. "A State-Machine Synthesizer-SMS," *Proceedings of the Eighteenth Design Automation Conference,* 301-305. Washington, D.C.: IEEE Computer Society Press, 1981.

22. Brown, Frank M. *Boolean Reasoning: The Logic of Boolean Equations.* Boston: Kluwer Academic Publishers, 1990.

23. Camposano, R. and R.K. Brayton. "Partitioning Begore Logic Synthesis," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 324-326. Washington, D.C.: IEEE Computer Society Press, 1987.

24. Camposano, R. and L.H. Trevillyan. "The Integration of Logic Synthesis and High-Level Synthesis," *1989 International Symposium on Circuits and Systems.* 744-747. Washington, D.C.: IEEE Computer Society Press, 1989.

25. Chen, Kuang-Chien and Saburo Muroga. "SYLON-DREAM: A Multi-Level Network Synthesizer," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 552-555. Washington, D.C.: IEEE Computer Society Press, 1989.

26. Cho, H. and others. "BEAT-NP: A Tool for Partitioning Boolean Networks," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 10-13. Washington, D.C.: IEEE Computer Society Press, 1988.

27. Dagenais, Michel R., Minod K. Agarwal and Micholas C. Rumin. "McBOOLE: A New Procedure for Exact Logic Minimization," *IEEE Transactions on Computer-Aided Design, CAD-5*: 229-237 (January 1986).

28. Darringer, John A. and William H. Joyner, Jr. "A New Look at Logic Synthesis," *17th Design Automation Conference.* 543-549. Washington, D.C.: IEEE Computer Society Press, 1980.

29. Darringer, John A. and others. "Logic Synthesis Through Local Transformations," *IBM Journal of Research and Development, 25*: 272-280 (July 1981).

30. Darringer, John A. and others. "LSS: A System for Production Logic Synthesis," *IBM Journal of Research and Development, 28*: 537-545 (September 1984).

31. de Geus, Aart J. and William W. Cohen. "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design and Test of Computers, 2*: 22-32 (August 1985).

32. de Geus, Aart J. "Logic Synthesis and Optimization Benchmarks for the 1986 Design Automation Conference," *23rd Design Automation Conference*. 78. Washington, D.C.: IEEE Computer Society Press, 1986.

33. De Micheli, Giovanni and David C. Ku. "HERCULES: A System for High-Level Synthesis," *25th ACM/IEEE Design Automation Conference*. 483-488. Washington, D.C.: IEEE Computer Society Press, 1988.

34. De Micheli, Giovanni and Alberto Sangiovanni-Vincentelli. "Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications," *IEEE Transactions on Computer-Aided Design, CAD-2*: 151-166 (July 1983).

35. De Micheli, Giovanni. "Symbolic Minimization of Logic Functions," *Proceedings of the IEEE International Conference on Computer-Aided Design*. 293-295. Washington, D.C.: IEEE Computer Society Press, 1985.

36. Detjens, Ewald and others. "Technology Mapping in MIS," *Proceedings of the IEEE International Conference on Computer-Aided Design*. 116-119. Washington, D.C.: IEEE Computer Society Press, 1987.

37. Devadas, Srinivas. "Approaches to Multi-Level Sequential Synthesis," *26th ACM/IEEE Design Automation Conference*. 270-276. New York, NY: ACM Press, 1989.

38. Devadas, Srinivas. "Boolean Decomposition in Multi-Level Logic Optimization," *Proceedings of the IEEE International Conference on Computer-Aided Design*. 290-293. Washington, D.C.: IEEE Computer Society Press, 1988.

39. Devadas, Srinivas, and others. "MUSTANG: State Assignment of Finite State Machines Targeting Multilevel Logic Implementations," *IEEE Transactions on Copmputer-Aided Design, CAD-7*: 1290-1300 (December 1988).

40. Diaz-Olavarrita, L. and S.G. Zaky. "Goal-Oriented Synthesis of Switching Functions," *1988 International Symposium on Circuits and Systems*. 1855-1859. Washington, D.C.: IEEE Computer Society Press, 1988.

41. Enomoto, Kiyoshi and others. "LORES-2: A Logic Reorginization System," *IEEE Design and Test of Computers, 2*: 35-41 (October 1985).

42. Fleisher, H. and others. "Simulated Annealing as a Tool for Logic Optimization in a CAD Environment," *Proceedings of the IEEE International Conference on Computer-Aided Design*. 203-205. Washington, D.C.: IEEE Computer Society Press, 1985.

43. Fujita, Masahiro, Hisanori Fujisawa and Nobuaki Kawato. "Evalutaion and Improvements of Boolean Comparison Method based on Binary Decision Diagrams," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 2-5. Washington, D.C.: IEEE Computer Society Press, 1988.

44. Garrison, Karl and others. "Automatic Area and Performance Optimization of Combinational Logic," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 212-214. Washington, D.C.: IEEE Computer Society Press, 1984

45. Ghazala, M.J. "Irredundant disjunctive and conjunctive forms of a Boolean function," *IBM Journal of Research and Development, 1*: 171-176 (April 1957).

46. Gilkinson, J.L. and others. "Automated Technology Mapping," *IBM Journal of Research and Development, 28*: 546-555 (September, 1984).

47. Goldberg, D.E. *Genetic Algorithms in Search and Machine Learning.* Reading, MA: Addison-Welsey, 1989.

48. Gore, Rajeev P. and Kotagiri Ramamohanarao. "Automatic Synthesis of Boolean Equations Using Programmable Array Logic," *26th ACM/IEEE Design Automation Conference.* 283-289. New York, NY: ACM Press, 1989.

49. Gregory, David, Karen Bartlett and Aart J. de Geus. "Automatic Generation of Combinatorial Logic From a Functional Specification," *1984 International Symposium on Circuits and Systems.* 986-989. Washington, D.C.: IEEE Computer Society Press, 1984.

50. Gregory, David and others. "SOCRATES: A System For Automatically Synthesizing and Optimizing Combinational Logic," *23rd Design Automation Conference.* 79-85. Washington, D.C.: IEEE Computer Society Press, 1986.

51. Gurunath, B. and Nripendra N. Biswas. "An Algorithm for Multiple Output Minimization," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 74-77. Washington, D.C.: IEEE Computer Society Press, 1987.

52. Hachtel, Gary D. and Michael R. Lightner. "Don't Care Conditions in Top Down Synthesis," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 316-319. Washington, D.C.: IEEE Computer Society Press, 1987.

53. Hachtel, Gary and others. "Performance Enhancements in BOLD using 'Implications'," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 94-97. Washington, D.C.: IEEE Computer Society Press, 1988.

54. Harvard Computation Laboratory Staff. *Synthesis of Electronic Computing and Control Circuits*, Annals of the Computation Lab., vol. 27. Cambridge MA: Harvard University Press, 1951.

55. Hayes, John P. *Computer Architecture and Organization.* New York: McGraw-Hill Book company, 1988.

56. Ho, B. "NAND Synthesis of Multiple-Output Combinational Logic Using Implicants Containing Output Variables," Ph.D. Dissertation, University of Wisconson, 1976.

57. Hong, S.J., R.G. Cain, and D.L. Ostapko. "MINI: A Heuristic Approach to Logic Minimization," *IBM Journal of Research and Development, 18*: 443-458 (September 1974).

58. Huntington, E.V. "Sets of Independent Postulates for the Algebra of Logic." *Transactions of the American Mathematical Society, 5*: 288-309 (1904).

59. Ishikawa, J. and others. "A Rule Based Logic Reorganization System LORES/EX," *1988 IEEE International Conference on Computer Design.* 262-266. Washington, D.C.: IEEE Computer Society Press, 1988.

60. Jacoby, R. and others. "New ATPG Techniques for Logic Optimization," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 548-551. Washington, D.C.: IEEE Computer Society Press, 1989.

61. Joyner, William H. and others. "Technology Adaptation in Logic Synthesis," *23rd Design Automation Conference.* 94-100. Washington, D.C.: IEEE Computer Society Press, 1986.

62. Kabakcioglu, A.M., P.K. Varshney and C.R.P. Hartmann. "An Artificial Intelligence Approach to PLA Optimization," *1988 International Symposium on Circuits and Systems.* 1861-1864. Washington, D.C.: IEEE Computer Society Press, 1988.

63. Kobrinsky, N.E. and B.A. Trakhtenbrot. *Introduction to the Theory of Finite Automata.* Amsterdam: North-Holland Publishing Co., 1965.

64. Kainec, James J. "A Diagnostic System Using Boolean Reasoning," *MS Thesis AFIT/GE/ENG/88D-16.* School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1988.

65. Kainec, James J. "Fundamentals of Boolean Algebra," Handout. Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, January 1990.

66. Kainec, James J. "The Use of Boolean Reasoning and Informed Search for Multi-Level Logic Circuit Optimization," *PhD Prospectus AFIT/DS/ENG/91.* School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990.

67. Kirkpatrick, S., C.D. Gelatt, Jr and M. P. Vecchi. "Optimization by Simulated Annealing," *Science, 220*: 671-680 (13 May 1983).

68. Lam, Jimmy and Jean-Marc Delosme. "Logic Minimization Using Simulated Annealing," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 348-351. Washington, D.C.: IEEE Computer Society Press, 1986.

69. Lightner, Michael and Wayne Wolf. "Experiments in Logic Optimization," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 286-289. Washington, D.C.: IEEE Computer Society Press, 1988.

70. Malik, Abdul A. and others. "A Modified Approach to Two-Level Logic Minimization," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 106-109. Washington, D.C.: IEEE Computer Society Press, 1988.

71. Malik, Sharad and R.H. Katz. "Combining Multi-Level Decomposition and Topological Partitioning for PLAs," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 112-115. Washington, D.C.: IEEE Computer Society Press, 1987.

72. Malik, Sharad and others. "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 6-9. Washington, D.C.: IEEE Computer Society Press, 1988.

73. Mano, M. Morris. *Digital Logic and Computer Design.* Englewood Cliffs NJ: Prentice-Hall Inc., 1979.

74. Marchesi, Michele. "A New Class of Optimization Algorithms for Circuit Design and Modelling," *1988 International Symposium on Circuits and Systems.* 1691-1695. Washington, D.C.: IEEE Computer Society Press, 1988.

75. Mathony, H.J. and U.G. Baitinger. "CARLOS: An Automated Multilevel Logic Design System for CMOS Semi-Custom Integrated Circuits," *IEEE Transactions on Computer-Aided Design, 7*: 346-355 (March 1988).

76. Mathony, H.J. and U.G. Baitinger. "Fast Efficient Algorithms for the Factoring of Multiple Output Logic Functions," *1988 International Symposium on Circuits and Systems.* 1851-1854. Washington, D.C.: IEEE Computer Society Press, 1988.

77. Matsunaga, Yusuke and Fujita Masahiro. "Multi-Level Logic Optimization Using Binary Decision Diagrams," *1989 IEEE International Conference on Computer-Aided Design.* 556-559. Washington, D.C.: IEEE Computer Society Press, 1989.

78. Mithani, D. "Implementation of NAND Synthesis Using Implicants Containing Output Variables," M.S. thesis, Department of Electrical Engineering, University of Wisconson, 1977.

79. Moore, Timothy P. and Aart J. de Geus. "Simulated Annealing Controlled by a Rule-Based Expert System," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 200-202. Washington, D.C.: IEEE Computer Society Press, 1985.

80. Nakamura, Shunichiro, Shinichi Murai and Chiyoji Tanaka. "LORES: Logic Reorganization System," *15th Design Automation Conference.* 250-260. Washington, D.C.: IEEE Computer Society Press, 1978.

81. Newton, A.R. and A.L. Sangiovanni-Vincentelli. "Computer-Aided Design for VLSI Circuits," *IEEE Computer, 19*: 38-60 (April 1986).

82. Pearl, Judea. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1984.

83. Perkins, Sharon R. and Tom Rhyne. "An Algorithm for Identifying and Selecting the Prime Implicants of a Multiple-Output Boolean Function," *IEEE Transactions on Computer-Aided Design, CAD-7*: 1215-1219 (November 1988).

84. Pratt, W.C. "Transformation of Boolean Equations for the Design of Multiple-Output Networks," Dissertation, Electrical Engineering Department, University of Illinois, 1976.

85. Quine, W.V. "The Problem of Simplifying Truth Functions," *American Mathematical Monthly, 59*: 521-531 (October 1952).

86. Quine, W.V. "A Way To Simplify Truth Functions," *American Mathematical Monthly, 62*: 627-631 (November 1955).

87. Rhyne, V. Thomas and others. "A New Technique For Fast Minimization of Switching Functions," *IEEE Transactions on Computers, C-26*: 757-764 (August 1977).

88. Rudell, Richard and Alberto Sangiovanni-Vincentelli. "Exact Minimization of Multiple-Valued Functions for PLA Optimization," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 352-355. Washington, D.C.: IEEE Computer Society Press, 1986.

89. Rudell, Richard and Alberto Sangiovanni-Vincentelli. "Multiple-Valued Minimization for PLA Optimization," *IEEE Transactions on Computer-Aided Design, CAD-6*: 727-750 (September 1987).

90. Saldanha, Alexander and others. "Multi-Level Logic Simplification Using Don't Cares and Filters," *26th ACM/IEEE Design Automation Conference.* 277-282. New York, NY: ACM Press, 1989.

91. Sasao, Tsutomu. "MACDAS: Multi-level AND-OR Circuit Synthesis using Two-Variable Function Generators," *23rd Design Automation Conference.* 86-93. Washington, D.C.: IEEE Computer Society Press, 1986.

92. Savoj, Hamid, Abdul A. Malik and Robert K. Brayton. "Fast Two-Level Logic Minimizers for Multi-Level Logic Synthesis," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 544-547. Washington, D.C.: IEEE Computer Society Press, 1989.

93. Singh, Kanwar J. and others. "Timing Optimization of Combinational Logic," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 282-285. Washington, D.C.: IEEE Computer Society Press, 1988.

94. Springer, George and Daniel P. Friedman. *Scheme and the Art of Programming.* New York: McGraw-Hill Book Company, 1989.

95. Steinberg, Louis I. and Tom M. Mitchell. "The Redesign System: A Knowledge-Based Approach to VLSI CAD," *IEEE Design and Test of Computers, 2*: 45-54 (February 1985).

96. Thomas, Donald. "Artificial Intelligence Techniques in Design and Test," *IEEE Design and Test of Computers, 2*: 21 (August 1985).

97. Trachtenberg, E.A. and D. Varma. "A Design Automation Tool for Fast, Efficient Decomposition of Logical Functions," *Proceedings of the IEEE International Conference on Computer-Aided Design.* 70-77. Washington, D.C.: IEEE Computer Society Press, 1987.

98. Ullman, Jeffrey D. *Computational Aspects of VLSI.* Rockville, MD: Computer Science Press, 1984.

99. Wong, Kong-Ngai and M. Ghazie Ismail. "New Heuristics For The Exact Minimization of Logic Functions," *1988 International Symposium on Circuits and Systems.* 1865-1868. Washington, D.C.: IEEE Computer Society Press, 1988.

Vita

Captain Eric J. Knutson was born on 28 March 1961 in Dover, Deleware. He graduated from high school in Hutchinson, Minnesota in 1979 and went on to attend Concordia College in Moorhead, Minnesota. In 1981 he transferred to the University of Minnesota to pursue a Bachelor of Science degree in Electrical Engineering. In June 1984 he graduated and entered USAF Officer's Training School. He received his commision as a Second Lieutenant in the USAF in September 1984 and subsequently served as a flight test engineer for the 6520th Test Group, Edwards AFB California. In March 1985 he accepted a position as Chief of the Instrumentation Division's Technical Support Branch. He served in that capacity until entering the School of Engineering, Air Force Institute of Technology, in May 1989. He is currently pursuing a Master's Degree in Computer Engineering and scheduled for graduation in December 1990.

Permanent address: Rt 3, Sioux Hills
Hutchinson, MN 55350

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1 DEC 90	MS Thesis

4. TITLE AND SUBTITLE
Recursive Optimization of Digital Circuits

5. FUNDING NUMBERS

6. AUTHOR(S)
Eric J. Knutson, Capt., USAF

7. PEREORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
School of Engineering
AFIT/ENG
Wright-Patterson AFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/90D-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. John W. Hines
Design Branch, Microelectronics Division,
Wright Research and Development Center,
WRDC/ELED
Wright-Patterson AFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The goal of this thesis is twofold: to identify the advantages and disadvantages of existing optimization systems and to develop an optimization system that uses Boolean principles to generate a recursive realization of combinational logic. Current multi-level optimization systems fall into two categories: local optimization which removes redundancy by pattern matching on a local scale and global optimization which works with the equations that specify a circuit rather than with the circuit implementation itself. While global systems are very flexible and can often produce near-optimal solutions, they are inherently complex. This research effort demonstrates that an effective global optimization system can be built upon sound Boolean principles. A recursive optimization system built in Scheme was thoroughly evaluated. The system achieved gate-input reductions as high as 52 percent. Subsequent modifications targeted improving the system's speed and effectiveness. As a result of these efforts, the optimization speed for a variety of sample specifications was doubled. Other findings led to a better understanding of this approach and showed that it is a viable technique for the optimization of digital circuits.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Global Optimization, Multi-level Optimization, Boolean Algebra Logic Minimization, Logic Optimization, Logic Synthesis	305
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL