MIS: A Multiple-Level Logic Optimization System

ROBERT K. BRAYTON, FELLOW, IEEE, RICHARD RUDELL,
ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE, AND ALBERT R. WANG

Abstract—**MIS is both an interactive and a batch-oriented multilevel logic synthesis and minimization system. MIS starts from the combinational logic extracted, typically, from a high-level description of a macrocell. It produces a multilevel set of optimized logic equations preserving the input–output behavior. The system includes both fast and slower (but more optimal) versions of algorithms for minimizing the area, and global timing optimization algorithms to meet system-level timing constraints. This paper provides an overview of the system and a description of the algorithms used. Included are some examples illustrating an input language used for specifying logic and don't-cares. Parts on an industrial chip have been re-synthesized using MIS with favorable results as compared to equivalent manual designs.**

Keywords—**Multiple-level logic, logic minimization, kernels, extraction, resubstitution, global phase assignment, factorization, decomposition, simplification, don't cares.**

I. INTRODUCTION

OVER THE past few years, placement and routing techniques have been developed which perform reasonably well for most block-oriented design styles. However, the synthesis of the circuit itself—deciding how to partition the logic, in what form to implement specific pieces of the logic, and what layout style to use for implementation—is still largely a manual process. Often the control logic portion of the chip is the most time consuming to design, is generally on the critical path for timing, and is implemented in an inefficient way. In addition, control and dataflow logic are generally separated unnaturally, leading to inefficiencies in layout and timing. Automated synthesis of the logic, optimized for speed and area, provides one of the next major challenges for CAD.

Research done over the past 30 years has led to efficient methods for implementing combinational logic in optimal two-level form using programmable logic arrays (PLA's). However, many logic blocks are inappropriate for this kind of implementation. For example, there exist functions whose minimum two-level representation has $2^n - 1$ product terms, where n is the number of primary inputs. In addition, even if a two-level representation contains a reasonable number of terms, there are many cases in which a multilevel representation can be implemented in less area and generally as a much faster circuit. Two-level logic representations can be viewed as special cases of more general multilevel representations. Hence, a general framework for logic design should offer multilevel synthesis tools which can select between two-level or multilevel implementations depending on the area and/or speed that can be obtained. However, to be able to explore the design tradeoffs, such a system should also offer a variety of both electrical design styles (e.g., domino logic, static CMOS) and layout design styles (e.g., Weinberger arrays, gate matrix, standard cells, and gate arrays).

Optimal multilevel logic synthesis is a known difficult problem which also has been studied since the 1950's. However, much work still remains to be done in order to achieve the same level of advancement as for two-level logic synthesis. In recent years, an increasing level of research has been apparent in multilevel logic synthesis. One of the first of the modern developments is the Logic Synthesis System (LSS) [9], [10] at IBM, which has as target technology a variety of gate arrays and standard cells. The Yorktown Silicon Compiler [6], which automatically synthesizes and lays out CMOS dynamic logic, is based completely on multilevel logic and has domino CMOS logic as its primary target technology. The SOCRATES system [11] is a multilevel logic synthesis system which uses gate arrays and standard cells, and is one of the earliest to emphasize optimized timing performance. The recently developed MIS system [7], which is the subject of this paper, is targeted at both area and timing optimization and uses static CMOS complex gates or macrocells, but, similar to the logic synthesis in the Yorktown Silicon Compiler, its algorithms can easily support a variety of target technologies.

A widely accepted optimization criterion for multilevel logic synthesis is to minimize the area occupied by the logic equations (which is measured as a function of the number of gates, transistors, and nets in the final set of equations) while simultaneously satisfying the timing constraints derived from a system-level analysis of the chip. Considerations such as testability should also be included; however, in most current systems, testability is only considered indirectly as a side effect of a less redundant implementation.

For multilevel design, two basic methodologies have evolved: 1) "global" optimization, where the logic functions are "factored" into an optimal multilevel form with little consideration of the form of the original description (e.g., the Yorktown Silicon Compiler, part of Angel [17], SOCRATES, and FDS [13]); 2) "peephole" optimiza-

Manuscript received January 26, 1987; revised July 9, 1987. This work was supported in part by the SRC under Contract 82-11-008, the Defence Advanced Research Projects Agency under Contract N00039-86-R-0365, the California State Micro Program, the National Science Foundation under Subcontract ECS-8430435, and the University of Colorado.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

IEEE Log Number 8716754.

tion, where local transformations are applied to the user-specified (or globally-optimized) logic function (e.g., a part of Angel, LSS, MAMBO [16], and SOCRATES).

Factoring algorithms have been proposed in the past (e.g., [1], [18]), but these techniques have required an exhaustive search which is prohibitively expensive for the complexity of the designs of interest today. Other proposed algorithms have lacked understanding of the technology constraints associated with particular implementations. New algorithms have appeared which are effective in partitioning complex logic functions [3] and can take into consideration the technology constraints of a particular implementation. Rule-based systems, as evidenced by LSS and SOCRATES, also have been effective in practice in the design of large systems.

The system presented in this paper, the Multilevel Logic Interactive Synthesis System (MIS), follows the global optimization paradigm and includes a variety of algorithms for the decomposition, factorization, minimization, and timing optimization of multilevel logic functions. Some of the algorithms are based on the early results of [3] and [4]; however, most algorithms are either new or represent new formulations of the basic ideas and offer better insight and clarify relations among the concepts. In this paper, we present all the algorithms and include appropriate background material.

MIS is an integral part of the current work at Berkeley in the area of automatic synthesis of digital integrated circuits, but it is also a stand-alone system which can read and write the semi-standard Logic Intermediate Format (LIF). Thus, MIS can be easily incorporated as part of other automatic digital design systems. A paradigm for this is the system at Berkeley which starts with a high-level description of the combinational logic written in the language BDS [12]. The program BDSYN extracts a set of logic equations, and stores the equations in the Oct database [15] as a *logic* view of the macrocell. MIS starts with the design in the Oct database and optimizes the logic equations to produce an optimal logic network which preserves the input–output behavior of the macrocell. This network is then stored back into the Oct database. Module generators then synthesize a symbolic layout for a macrocell directly from an optimized logic network. Floor-planning, placement and routing, and compaction tools are used to complete the chip design. Timing constraints, derived from a system-level timing analysis, in terms of input arrival times and output required times, can be passed to and from MIS. These can be used to guide the module generator tools in the placement and routing of the gates within a macrocell, and also by the floor-planning, placement and routing tools to guide the placement and routing of the macro-cells.

MIS is currently being used for complex-gate static CMOS designs. Depending on the module generator used to generate the final layout, either a fixed-library of predesigned gates, or a flexible library of procedurally designed gates can be used. Most important, however, is that the algorithms used are general and, hence, largely independent of the implementation technology.

MIS is organized as a set of operators which are applied to the underlying Boolean network data structure. The sequence in which these are applied can be controlled interactively or automatically by invoking a "script" (a sequence of MIS commands). Currently, several scripts have been developed since the quality of results produced by a single script depends on the type of logic being processed. However, we are experimenting with a universal self-adapting script which should make the results more independent of the script.

MIS is also a continually evolving system; as better algorithms are developed, they are added easily to MIS and invoked at appropriate points of the scripts. The interactive design of MIS is useful in two respects: first as an aid to the developer and second as a tool for the sophisticated user to help determine if a given script is producing satisfactory results. This second mode of operation helps to satisfy originally skeptical users and eases the acceptance of MIS by designers.

An important part of logic minimization is specification and extraction of don't-cares. Since, for some types of logic, the quality of the results depends on the initial multilevel network implied by the input description, in Section II, we start with the input language translator BDSYN used in the Berkeley system as an example of logic and don't-care specification and extraction.

We detail all the major algorithms of MIS. Necessary background material for this is given in Section III. We continue with the global optimization strategy for area minimization, described in Section IV, and the local optimization steps are refining the implementation, in Section V.

MIS can be used first to minimize area without concern for delay. Initial delays can then be estimated and used to calculate the critical paths at the system level. With this information, MIS can be used to restructure the logic equations to tradeoff area for speed. This restructuring consists of collapsing logic functions to fewer levels and duplicating logic functions. More refined timing optimization can be done by sizing transistors, and by passing constraints to the physical design tools to influence the placement and routing of the critical paths. This part of the system is presented in Section VI.

In Section VII, some experimental results are presented, and in Section VIII, some conclusions and future directions in which MIS is evolving are discussed.

II. Design Specification and Logic Extraction

In order to describe how MIS may be used as part of a larger digital systems synthesis system, we describe its use in the Berkeley system. The current version of the Berkeley Synthesis System takes as input a collection of combinational logic blocks and latches, which completely describe the chip being synthesized. The decomposition from a behavioral level into this functional/structure level is presently a manual task performed by the designer. Of ultimate interest are tools to assist and/or automate part of all of the higher-level design process.

In order to facilitate the initial specification of the com-

binational logic blocks, a hardware description translator called BDSYN is used. BDSYN reads BDS language descriptions [12]. (BDS is the behavioral language used by the Digital Equipment Corporation in their DECSIM simulation system. DECSIM supports, among other things, mixed-mode simulation including behavioral descriptions down to timing-level descriptions.) BDS provides a high-level description with a large variety of operators and control structures. It is similar in structure to other hardware description languages such as the ISPS variant known as N.2 [14].

BDS is best thought of as a programming language with the built-in data type of a bit-vector and the basic operations on bit-vectors (i.e., bit selection, logical operations, shift and rotate operations, and arithmetic operations). A list of the operators available in BDS is given in Table I, and a list of the statements in BDS is given in Table II.

The translator, BDSYN, reads a BDS description, and generates a logic network equivalent to the BDS program. This program has a well-defined set of inputs and outputs. The semantics of BDSYN are that each output or intermediate variable is computed as a combinational function of the inputs, intermediate variables, and outputs. It is important to note that the final logic network produced by BDSYN reflects directly the multilevel nature of the input description, thus removing any limitation on the type of logic described, and providing the user with the ability to specify the initial Boolean network for synthesis.

An arbitrary BDS description is allowed (including functions, loops, global variables, and multiple-assignment to variables). This includes premature exits from loops using the LEAVE statement, and premature exits from functions using the RETURN statement. Complex operations such as binary addition, binary comparison, variable-bit shift, and variable-bit extract are handled by inserting a function call to a library routine which performs the complex operation. These library routines are also written in BDS (consisting of about 200 lines of code) and in-line expanded during the BDSYN translation process. Simple optimization techniques such as constant folding and constant propagation are performed during the translation. Iteration loops are required to be static, so that the loop range is known at translation time.

An important feature of BDSYN is that it allows the explicit specification by the designer of don't-care conditions. Specifying this information to the loic synthesis system can dramatically improve the optimization of the logic. Conceptually, the don't-care set for a logic network is a set of input patterns (specified for each output of the logic network) providing the conditions for which an output may assume either a 0 or 1 value. This don't-care set, like any other logic function, can be specified in canonical, two-level, sum-of-products form. However, for the same reasons that it is more convenient to use a multiple-level form for the logic network itself, it is also advantageous to use a multiple-level form to specify the don't-care conditions for the network.

In order to simplify the specification of the don't-care conditions, the designer is able to specify the logic network and the don't-care set for the logic network in the same description file. This is achieved as follows. An extra input called *DONT_CARE* is added to the logic network. During simulation *DONT_CARE* is assigned the value "X" (or undefined). This will help detect errors in the model during simulation. During translation, if a variable is assigned the value *DONT_CARE*, we assign the semantics that, under the conditions leading to the assignment, the assigned variable may assume either a 0 or 1 value, but it is undefined which.

Currently, MIS makes use of the don't-care set by collapsing the network to two-levels and then using a two-level logic minimizer. Hence, MIS has the limitation, at present, that the don't-cares can only be used for those networks which can be represented with reasonable efficiency in two-levels of logic. We are currently exploring modifications to the global and local optimization procedures of MIS to remove this limitation.

To extract the two-level form of the don't-care set from the description, we first translate the network into multiple-level form (including the extra input *DONT_CARE*). We then compute the conditions for which input conditions an output variable changes value when *DONT_CARE* changes value. This provides the *DONT_CARE*

TABLE I
BDSYN OPERATORS

NOT	Boolean not
AND	Boolean and
OR	Boolean inclusive or
NAND	Boolean not-and
NOR	Boolean not-or
EQV	Boolean not exclusive-or
XOR	Boolean exclusive-or
+	Arithmetic plus
-	Arithmetic minus
*	Arithmetic multiply
GTR	Arithmetic greater-than
GEQ	Arithmetic greater-than or equal
LSS	Arithmetic less-than
LEQ	Arithmetic less-than or equal
EQL	Arithmetic equal
NEQ	Arithmetic not-equal
SL0	Shift left inserting 0 from the right
SL1	Shift left inserting 1 from the right
SR0	Shift right inserting 0 from the left
SR1	Shift right inserting 1 from the left
SLR	Rotate left
SRR	Rotate right
SXT	Sign extend
OXT	One-extend
ZXT	Zero-extend
&	Concatenation of bit-strings
<hi:lo>	Select substring of a bit-string

TABLE II
BDSYN STATEMENTS

MODEL	Delimit a model
ROUTINE	Defines a routine
STATE	Defines a variable
CONSTANT	Define a constants
IF ... THEN ... ELSE	Conditional execution
SELECT .. ENDSELECT	Multi-way conditional execution
SELECTALL .. ENDSELECTALL	Multi-way conditional execution
RETURN	Return from a routine
FOR	Defines a for-loop
LEAVE	Pre-mature exit from a for-loop

set for that output variable. This is computed as follows. If f represents a function in the logic network, and d the input variable *DONT_CARE*, then the don't-care set for f is

$$\text{don't-care set} = f_d \oplus f_{\bar{d}}$$

where $\oplus$ is the exclusive-OR operation, and f_d ($f_{\bar{d}}$) is the Shannon cofactor [5] of the function f with respect to d ($\bar{d}$).

To illustrate how logic may be specified for MIS using BDSYN, we discuss two examples. Fig. 1 shows the BDS description of the register file decoder from the SPUR microprocessor [20]. The register file decoder is complicated by the use of the overlapping window scheme [19]. *cwp* is the current window pointer for the register file, and *reg* is the index of the register within the window. A one-hot signal *addr* selects one of the 138 registers in the register file. The operator "&" is bit concatenation, the operator ZXT performs an extension of an operand with zero fill, and the operator SL0 performs a shift-left with zero fill. Note the use of complex operations such as addition, comparison, and variable shift-left (all of which are supported in BDSYN). Also note that the variable *address* is multiply assigned.

The second example is a program which implements a combinational logic circuit for the following specification.

Problem (Count Zeros): The input is an N-bit binary string expected to consist of a string of 1's, a string of 0's, and a string of 1's. Any of these strings may be zero length. For example, with $N = 8$, 11001111, 10000001, and 11110000 are all valid strings, but 11001100 is not. The problem is to design a circuit which, given the N bit string, returns the count of number of consecutive zeros in the string, or returns an error condition if the string is invalid, in which case the output is undefined.

A BDS program for this example is shown in Fig. 2. The constants N and *LOGN* parameterize the model to the size of the input string. The routine **ff** is used to find the index of the first bit in a string which matches a given value (starting from the least significant bit). SR1 is a shift operator which shifts the operand right inserting 1's from the left. The count of the number of zeros is assigned the value *DONT_CARE* if an invalid string is given as input.

As seen from the examples, the BDSYN translator provides a convenient means for describing combinational logic, including the don't-care set for a piece of logic.

```
MODEL decoder
    addr<137:0>        ! output: 1-hot word select
=
    cwp<2:0>,          ! input: current window pointer
    reg<4:0>;          ! input: current register number

BEHAVIOR;

    CONSTANT NUMREGS = 138, NUMGLOBALS = 10;

    ROUTINE main;
        STATE address<7:0>;

        ! Check for reference to global register
        IF reg LSS NUMGLOBALS THEN
            address = reg
        ELSE BEGIN
            ! compute address, check overflow
            address = (cwp & 0000#2) + reg;
            IF address GTR (NUMREGS - 1) THEN
                address = address - (NUMREGS - NUMGLOBALS);
        END;

        ! Create the one-hot decode based on the address
        addr = (ZXT {WIDTH=138} 1) SL0 address;
    ENDROUTINE main;

ENDBEHAVIOR;
ENDMODEL decoder;
```

Fig. 1. BDSYN program for the SPUR register file decoder.

```
MACRO N = 8 $ENDMACRO;
MACRO LOGN = 3 $ENDMACRO;

MODEL count_zeros
    error<0>,                          ! output: indicate error in string
    count<LOGN-1:0>                    ! output: count of number of zeros
=
    in<N-1:0>;                         ! input: bit string

BEHAVIOR;

    ! find first bit matching 'val' (return index of the bit)
    ROUTINE ff<LOGN:0>(x<N-1:0>, val<0>);
        STATE i<>;
        FOR i FROM 0 TO N DO
            IF x<i> EQL val THEN
                RETURN i;
        RETURN N;
    ENDROUTINE ff;

    ROUTINE main;
        STATE x<N-1:0>;
        x = in SR1 ff(x, 0);           ! strip off leading 1's
        count = ff(x, 1);              ! count number of 0's
        x = x SR1 count;               ! strip off the 0's
        error = ff(x, 0) NEQ N;        ! error if any 0's are left
        IF error THEN
            count = DONT_CARE;
    ENDROUTINE main;

ENDBEHAVIOR;
ENDMODEL count_zeros;
```

Fig. 2. BDSYN program for the count zeros problem.

III. Definitions

In this section, we provide some basic definitions and concepts which are helpful in describing the algorithms used in MIS as well as in understanding how to use MIS interactively.

A. Sum-of-Products Form

A *variable* is a symbol representing a single coordinate of the Boolean space (e.g., $\bar{a}$).

A *literal* is a variable or its negation (e.g., a or $\bar{a}$).

A *cube* is a set C of literals such that $x \in C$ implies $\bar{x} \notin C$ (e.g., $\{a, b, \bar{c}\}$ is a cube, and $\{a, \bar{a}\}$ is not a cube). A cube represents the conjunction of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1, respectively.

An *expression* is a set f of cubes. For example, $\{\{a\}, \{b, \bar{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \bar{c}\}$. An expression represents the disjunction of its cubes.

We use the conventional algebraic notation for cubes and expressions. For example, the cube $\{a, b, \bar{c}\}$ can be written as $ab\bar{c}$, and the expression $\{\{a\}, \{b, \bar{c}\}\}$ can be written as $a + b\bar{c}$.

An expression is *nonredundant* if no cube in the expression properly contains another. For example, $a + ab$ is redundant because $\{a\} \subset \{a, b\}$.

A *Boolean expression* is a nonredundent expression.

The *support* of an expression f is $sup(f) = \{a \mid \exists$ *cube* $C \in f$ such that $x \in C$ or $\bar{x} \in C\}$. For example, $sup(ab + \overline{ac}) = \{a, b, c\}$.

Two expressions f and g have *disjoint support* if $sup(f) \cap sup(g) = \emptyset$.

An expression provides a natural representation of the *sum-of-products* form of a function.

We caution that care should be taken while interpreting the set operators $\cup$ and $\cap$ when applied to cubes written in algebraic notation. For example, $abc \cup d$ equals $abcd$ and **not** $abc + d$ (because $\{a, b, c\} \cup \{d\} = \{a, b, c, d\} = abcd$ and $abc + d = \{\{a, b, c\}, \{d\}\}$). Thus, $\cup$ when applied to two cubes corresponds to the intersection of the corresponding Boolean spaces (and not the union as might be expected).

B. Factored Forms

The usual representation of a logic function is the *sum-of-products* form. We define another (more useful for multilevel logic) representation, the *factored form*. The factored form is defined recursively: 1) a literal is a factored form; 2) a sum of factored forms is a factored form; 3) a product of factored forms is a factored form.

In other words, a factored form is a sum of products of sums of products, ..., of arbitrary depth. For example, the expression

$$abe\bar{g} + abfg + ab\bar{e}g + ace\bar{g} + acfg + ac\bar{e}g + de\bar{g} + dfg + d\bar{e}g + bh + bi + ch + ci$$

can be written in factored form as

$$(a(b + c) + d)(e\bar{g} + g(f + \bar{e})) + (b + c)(h + i).$$

The factored form, in general, is not unique. For example, the expression $abc + abd + cd$ is itself a factored form, but can also be written as $ab(c + d) + cd$ or $abc + (ab + c)d$, both factored forms. Typically we are interested in the *minimum* factored form, which is the factored form containing the least number of literals.

C. Algebraic and Boolean Division

The *product* of two expressions f and g, fg, is a set $\{c_i \cup d_j \mid c_i \in f$ and $d_j \in g\}$ made nonredundant using the standard Boolean operation of containment (e.g., $ab + a = a$).

When f and g have disjoint support, fg is an *algebraic product* (no Boolean operations are needed to obtain the product); otherwise fg is a *Boolean product*. For example, $(a + b)(c + d) = ac + ad + bc + bd$ is an algebraic product, and both $(a + b)(a + c) = aa + ab + ac + bc = a + bc$ and $(a + b)(\bar{b} + c) = a\bar{b} + ac + b\bar{b} + bc = a\bar{b} + bc$ are Boolean products.

The *quotient* of an expression f by another expression g, f/g, is the largest set q of cubes such that $f = qg + r$ where q is the quotient and r is the remainder.

If qg is restricted to an algebraic product, f/g is the (unique) *algebraic quotient*, otherwise f/g is a (nonunique) *Boolean quotient*.

Similarly, if $f/g = q \neq \emptyset$, and q can be obtained using algebraic division, then g is an *algebraic divisor* of f (so is q), otherwise, g is a *Boolean divisor* of f (so is q).

For example, let

$$f = ad + bcd + e$$

$$g = a + bc$$

$$h = a + b$$

using algebraic division:

$$f/g = d \ (g \text{ is an algebraic divisor of } f)$$

using Boolean division:

$$f/h = (a + c)d \ (h \text{ is an Boolean divisor of } f)$$

D. Kernels and Co-kernels

The notion of a kernel of a logic expression was introduced in [3] to provide a means for finding subexpressions common to two or more expressions. Kernels are our bridge between expressions and algebraic factored forms. In this section, we use only algebraic operations (i.e., algebraic product, algebraic division, etc.), but omit the word algebraic for brevity.

We say an expression is *cube-free* if no cube divides the expression evenly (e.g., $ab + c$ is cube-free; $ab + ac$ and abc are not cube-free). Notice that a cube-free expression must have more than one cube.

The *primary divisors* of an expression f are the set of expressions

$$D(f) = \{f/C \mid C \text{ is a cube}\}.$$

The *kernels* of an expression f are the set of expressions

$$K(f) = \{g \mid g \in D(f) \text{ and } g \text{ is cube-free}\}.$$

In other words, the kernels of an expression f are the cube-free primary divisors of f. The cube C used to obtain the kernel $k = f/C$ is called the *co-kernel* of k, and we use $C(f)$ to denote the set of co-kernels of f. For example, define a function x as

$$x = adf + aef + bdf + bef + cdf + cef + g$$
$$= (a + b + c)(d + e)f + g.$$

(For brevity, we often use the factored form to represent an expression.) $a + b + c$ is a kernel corresponding to co-kernels df and ef because it is a cube-free primary divisor obtained by x/df or x/ef. Similarly, $d + e$ is a kernel corresponding to co-kernels af, bf, or cf. $(d + e)f$ is a primary divisor corresponding to x/a, x/b, or x/c. However, $(d + e)f$ is not a kernel because it is not cube-free. Finally, x itself is a kernel because it it cube-free and a primary divisor corresponding to a trivial cube 1. Clearly, $K(x) \subseteq D(x)$.

The following theorem shows an important and useful properties of kernels.

Theorem 1 [3]: f and g have a common multiple-cube divisor d if and only if $\exists \equiv k_f \in K(f), k_g \in K(g)$ such that $d = k_f \cap k_g$.

This theorem, proven in [3], states that two functions f and g have a multiple-cube common divisor if and only if the intersection of a kernel from f and a kernel from g has more than one cube. Thus, it provides a method for detecting if two or more expressions have any common algebraic divisors other than single cubes. This can be done by computing the set of kernels for each logic expression, and forming nontrivial (more than one term) intersections among kernels from different functions. We do not need to compute the set of all algebraic divisors for each expression to determine if there are common multiple-cube divisors. This leads to great run time efficiency since the set of kernels is much smaller than the set of all algebraic divisors. In addition, if the intersection set of all kernels consists of single cubes or is empty, then we need only look for common divisors consisting of single cubes.

For certain operations described in the following sections, it is nearly as effective and frequently more efficient to compute a certain subset of $K(f)$ rather than the full set. This leads to the following recursive definition for the *level* of a kernel. Let

$$K^0(f) = \{k \in K(f) \mid K(k) = \{k\}\}$$

$$K^n(f) = \{k \in K(f) \mid \exists k_1 \in K(k) \text{ such that } k_1 \in K^{n-1}(f)\}$$

then

if $k \in K^0(f)$, then k is a level-0 kernel of f

if $k \in K^n(f)$ and $k \notin K^{n-1}(f)$, then k *is a level* n kernel of f.

According to the definition, a kernel is said to be of level-0 if it has no kernels except itself. Similarly, a kernel is of level n if it has at least one level $n-1$ kernel but no kernels (except itself) of level n or greater. This gives us a natural partition of the kernels:

$$K^0(f) \subset K^1(f) \subset K^2(f) \subset \cdots \subset K^n(f) \subset K(f).$$

For example,

$$x = (a(b + c) + d)(e\bar{g} + g(f + \bar{e})) + (b + c)(h + i)$$

has, among others, the kernels $b + c$ and $a(b + c) + d$ which are level-0 and level-1, respectively, while x itself is a kernel of level 2 since it has level 1 kernels but no level 2 kernels other than itself. Note that if

$$x = j(a(b + c) + d)(e\bar{g} + g(f + \bar{e})) + (b + c)(h + i)$$

then x is a kernel of level 3 since it contains the level 2 kernel

$$(a(b + c) + d)(e\bar{g} + g(f + \bar{e})).$$

There are algorithms to find $K^i(f)$ without generating the full set $K(f)$.

E. Boolean Network

A Boolean network is a technology-independent structure for representing a multilevel logic function. It is a directed acyclic graph (DAG) where each node i is associated with 1) a variable y_i and 2) a representation f_i of a logic function (sum-of-products form and/or factored form).

In the graph, an arc connects node i to node j if $y_i \in sup(f_j)$. The primary inputs $x_1, x_2, \ldots, x_n$ and primary outputs $z_1, z_2, \ldots, z_m$ of a Boolean network corresponds to special nodes in the graph, the source nodes and sink nodes of the DAG. There is no logic function associated with the source or sink nodes.

The fan-in of a node i is the set of all nodes pointing to i. The fan-out of a node i is the set of all nodes which node i points to. The transitive fan-in of a node i is defined recursively as the union of the fan-in nodes of i with the transitive fan-in of its fan-in nodes.

There is one-to-one correspondence between a function in the multilevel logic network and a node in the DAG. Throughout this paper, functions and nodes are used interchangeably.

F. Costs and Values

Each node in the Boolean network is a completely specified Boolean function represented by a sum-of-products form and a factored form. During the synthesis process, we need for guidance a measure of the complexity of the Boolean network. We define the area complexity of a Boolean network as the sum over all of the nodes of the area complexity of each node; the area complexity of a node is the minimum number of literals required to represent the function in a factored form. For example, the function

$$f_1 = abe\bar{g} + abfg + ab\bar{e}g + ace\bar{g} + acfg + ac\bar{e}g + de\bar{g} + dfg + d\bar{e}g$$

with 33 literals can be written in factored form using nine literals:

$$f_1 = (a(b + c) + d)(e\bar{g} + g(f + \bar{e})).$$

so its area complexity is 9 or less.

There are several algorithms (described in Section V-A) which can be used to factor a single logic equation, each with a different performance and quality tradeoff. Because the cost function must be estimated frequently during the synthesis, we desire a fast factoring algorithm for computing the current cost of the network.

To justify the use of the total number of literals in the factored form as an area complexity measure, consider the implementation of f_1 as a CMOS complex gate. It can be implemented as shown in Fig. 3 using nine pairs of n-type and p-type MOS transistors (assuming all signals and their complements are available). Thus, the number of literals in the factored form corresponds to the number of transistors needed to implement the function as a complex gate.

In many cases, it is necessary to further decompose the function into smaller gates. In general, this increases the total number of transistors needed to build f_1. However, if f_1 is decomposed in a manner corresponding to its factored form, then the transistor count increases, but only slightly. For example, the implementation of f_1 as

$$f_1 = hi$$
$$h = a(b + c) + d$$
$$i = e\bar{g} + j$$
$$j = g(f + \bar{e})$$

has a total literal count of 12 (instead of 9). The total literal count has increased by the decomposition, but it remains a good predictor of the area complexity of the function.

Now assume that f_1 is identified as a common factor to several other functions, and is created to reduce the total complexity of the network. Regardless of how f_1 is implemented (either as a single gate, or requiring further decomposition), it has reduced the number of transistors in the network roughly by an amount corresponding to the total number of transistors that have saved in the functions which it feeds minus the number of transistors in f_1. This leads to the following definition for the *area-value* of a node.

To estimate the area value of a factor y, let $FANOUT(y)$ be the set of functions which can be written using y as an algebraic factor, let $N(f, y)$ be the number of times either y or $\bar{y}$ appears in the factored form for f, and let $L(y)$ be the number of literals in the factored form for y. The area value of y is defined as

$$area\_value(y) = \left(\left(\sum_{f \in FANOUT(y)} N(f, y)\right) - 1\right) \cdot (L(y) - 1) - 1.$$

The area value estimates how many literals are saved by introducing the function y into the network, and rewriting each function which can be written in terms of y. To justify this estimate, consider that we can replace each occurrence of y by making a copy of the factored form for y and placing it directly into the factored form for f. Similarly, $\bar{y}$ can be replaced by the dual of the factored form for y. Since the dual of a factored form has the same number of literals as the factored form, we treat the occurrences of y and $\bar{y}$ equally. The -1's in the formula account for the number of literals needed to implement y, the literals needed to use y and $\bar{y}$ in the other functions.

IV. Global Area Optimization

The goal of global area optimization is to minimize the complexity of a set of logic equations thereby minimizing the area needed to implement them. Global techniques allow significant restructuring of the network based on consideration of all of the nodes in the network (as opposed to the local techniques discussed in Section V which consider only a small subset of the nodes at a time). An important consideration for global area optimization algorithms is that they be independent of the particular design style or technology.

The logic equations are represented by a Boolean network. Each node in the network has an associated completely-specified Boolean function represented in both a sum-of-products form and a factored form. The sum-of-products form is useful for manipulating the logic function at the node, while the factored form provides a better area estimate for the node. As mentioned earlier, the count of the number of literals in the factored form of each node is used to estimate the total complexity of the network.

In this section, methods for generating common factors from a set of logic equations are presented (*extraction*). In addition, methods for checking whether an existing function is a factor of another function in the network are also presented (*resubstitution*). Finally, algorithms for reducing the total number of inverters without increasing the size of other functions is presented (*global phase assignment*).

A. Extraction

The most important (and most difficult) step in global area minimization is to identify divisors common to two or more functions which can be used to reduce the total number of literals in the network. Because the size of the set of all algebraic and Boolean divisors is very large, and because an algorithm for efficiently generating only useful Boolean divisors does not exist, we restrict our attention to algebraic divisors in the extraction step.

Our approach is to first identify and extract useful multiple-cube divisors from the functions in the network. This terminates when there are no more multiple-cube divisors of any pair of functions. Then, single-cube divisors (also called *common subcubes*) are extracted until no further single-cube divisors exist.

The basic algorithm for detecting multiple-cube divisors using kernels was given by Brayton and McMullen [3]. We present modifications to the basic algorithm for generating subsets of kernels (in particular, the level-0 kernels and any single level-0 kernel). An interesting observation presented here is that the problems of detecting intersections in a set of kernels, and detecting common subcubes in a set of functions, are computationally equivalent to finding the kernels of an expression.

1) Kernels and Kernel Intersections: The following is an algorithm for computing all the kernels of a function f. Several other methods have been proposed, but this algorithm seems to have withstood all competition so far. The literals in the support of f are numbered from 1 to n, g is a cube-free expression and all literals less than j have been divided.

```
KERNELS(j, g):
  R = ∅
  for (i = j; i ≤ n; i++) {
    if (l_i appears in more than one cube) {
      c = largest cube dividing g/{l_i} evenly
```

```
        if (l_k ∉ c for all k < i) {
            R = R ∪ KERNELS(i+1, g/({l_i} ∪ c))
        }
    }
}
R = R ∪ {g}
return R
```

To use the algorithm for generating all the kernels of f, first factor out the largest cube dividing f evenly, then call KERNELS(1, g). The algorithm works as follows. The argument j in KERNELS is a pointer to the literals already factored out, (all literals less then j have been processed). KERNELS is designed to find all kernels associated with any co-kernel not containing any of the literals l_i for $i < j$. The recursive call to KERNELS restricts the terms to those containing literal i. These are then kernels which have as co-kernel a cube whose literals include l_i and the literals of c, the largest cube factor of g/l_i. The recursion is done only if the cube c has no literals $k \leq i$, since all co-kernels associated with this recursion will involve the literals of c, and if one of these has been factored already, we would just reproduce a kernel and co-kernel already found. This makes the algorithm such that we only process unique co-kernels and gives the algorithm an effective tree-trimming strategy for searching for kernels. The algorithm can also be used to generate all the co-kernels by returning the co-kernels rather than kernels.

The number of kernels of an expression can grow exponentially in the number of literals in the support of the expression. This is particularly evident when the function has symmetric variables, because each permutation of the symmetric variables in a kernel produces another kernel. However, in practice, we find that the set of all kernels is reasonably small. In particular, during the synthesis process, most of the functions in the network are small enough such that all of the kernels can be efficiently generated. However, there are times when we want to find quickly a good kernel of a function (and not necessarily the best kernel). Therefore, we also present an algorithm for generating all of the level-0 kernels (a subset of the set of all kernels), and an algorithm for finding any one level-0 kernel.

The algorithm below for generating level-0 kernels is a simple modification of KERNELS. This is based on the observation that if no kernels of g are found in the **for** loop, then g is a level-0 kernel.

```
L_O_K(j, g):
    R = ∅
    for (i = j; i ≤ n; i++) {
        if (l_i appears in more than one cube) {
            c = largest cube dividing g/{l_i} evenly
            if (l_k ∉ c for all k < i) {
                R = R ∪ L_O_K(i+1, g/({l_i} ∪ c))
            }
        }
    }
    if (R = ∅) R = {g}
    return R
```

In Section V-A, we describe the algorithms quick factoring and quick decomposition which need to find just one level-0 kernel. The algorithm above can be further modified to find any single level-0 kernel quickly:

```
ONE_L_O_K(j, g):
    for (i = j; i ≤ n; i++) {
        if (l_i appears in more than one cube) {
            c = largest cube dividing g/{l_i} evenly
            if (l_k ∉ c for all k < i) {
                return ONE_L_O_K(i+1, g/({l_i} ∪ c))
            }
        }
    }
    return g
```

Once the kernels (or level-0 kernels) of all the functions have been computed, we still have the problem of finding intersections among the kernels from different functions. This can be done efficiently in a manner similar to the generation of the kernels according to the following proposition. Recall that an expression is a set of cubes, and a cube is a set of literals.

Proposition 1: Let f be an expression, let $g \subset f$ be a subexpression of f, and let $C = \cap\, g_i$ be the intersection over all of the cubes of g. Define $A(f)$ as the set of all such cubes C as g ranges over all subsets of f with two or more cubes. Then the set $A(f)$ is the set of all co-kernels of f.

Corollary 1: The subset of $A(f)$, consisting of the cubes generated by the subsets $g \subset f$ of cardinality 2, contains the set of level-0 co-kernels of f.

This proposition gives a new alternate view of the operation of computing the kernels (or co-kernels) of an expression. A co-kernel corresponds to a nonempty intersection of two or more cubes of the expression; the kernel is the result of dividing the expression by this co-kernel. However, the approach to generating the kernels of an expression presented in this proposition can be extremely inefficient—most of the possible $2^{|f|}$ intersections of subsets of f are empty. The beauty of the kernel algorithm presented above is that it avoids explicitly enumerating any empty intersections.

This proposition is at the heart of developing an algorithm for finding intersections between the kernels from multiple functions. This problem is stated as follows. Given a set of kernels

$$K = \{k_1, k_2, \ldots, k_n\} \quad \text{with} \quad k_i = \{c_1, c_2, \ldots, c_{m_i}\}$$

find the set of all kernel intersections $I(K)$.

We first form a new expression $IF(K)$ which corresponds to the set K of the kernels. We associate each distinct cube in $\cup\, k_i$ with a new literal, and each kernel with a new cube which contains all the literals corresponding to the cubes of the kernel. This set of cubes forms a new function which we call $IF(K)$.

Proposition 2: Every element in the set of co-kernels of $IF(K)$ (that is, $C(IF(K))$) corresponds to a unique kernel intersection in $I(K)$.

The following example demonstrates the approach. Given

$$K = \{k_1, k_2, k_3\}$$
$$k_1 = abc + de + fg$$
$$k_2 = abc + de + fh$$
$$k_3 = abc + fh + gh$$

we are to find the intersections of kernels k_1, k_2, and k_3. In this case, the nonempty intersections of the kernels are easily seen to be

$$abc = k_1 \cap k_2 \cap k_3$$
$$abc + de = k_1 \cap k_2$$
$$abc + fh = k_2 \cap k_3.$$

We now demonstrate how to use the Proposition 2 to generate these intersections. The distinct cubes are

$$t_1 = abc$$
$$t_2 = de$$
$$t_3 = fg$$
$$t_4 = fh$$
$$t_5 = gh.$$

From this, we form the single function $IF(K)$ where each cube of $IF(K)$ corresponds to a kernel of K

$$IF(K) = t_1t_2t_3 + t_1t_2t_4 + t_1t_4t_5.$$

For example, the kernels k_1 corresponds to the term $t_1t_2t_3$. The kernels and co-kernels of $IF(K)$ are

$$K(IF(K)) = \{t_2t_3 + t_2t_4 + t_4t_5, t_3 + t_4, t_2 + t_5\}$$
$$C(IF(K)) = \{t_1, t_1t_2, t_1t_4\}.$$

Working backwards, we can interpret each co-kernel of $IF(K)$ as a subset of the cubes from the kernels

$$I(K) = \{t_1, t_1 + t_2, t_1 + t_4\}$$
$$= \{abc, abc + de, abc + fh\}.$$

Hence, to compute the intersections among the kernels of many functions, we first form the function $IF(K)$ and then generate the kernels of this function. Note that we can choose to generate only the level-0 kernels of this function, rather than all of the kernels, again for the sake of efficiency.

2) Multiple-Cube Extraction: Theorem 1 is used to determine whether several functions have any common multiple-cube divisors. Instead of generating all possible divisors of each function, we can compute the kernels of each function and generate the intersections of the kernels. We can rate an intersection by its area value (i.e., the number of literals which can be saved if we were to extract that intersection). The formula for computing the area value of a node can be simplified in this case because the term $N(f, y)$ is always 1 when y corresponds to a kernel of function f. The simplified formula for computing the area value of a kernel intersection is

$$area\_value(k) = (NF(k) - 1)(L(k) - 1) - 1$$

where k is a kernel intersection, $NF(k)$ is the number of functions of which k is a divisor, and $L(k)$ is the number of literals of k in its factored form.

The following greedy extraction algorithm is implemented in MIS. Two parameters are used to control the tradeoff between the quality and run-time efficiency of the results. Parameter k is the level of kernels generated at each step. Parameter n is the number of kernel intersections to use before the set of kernels and their intersections is recomputed. FROM(x) is the set of functions for which x may be a divisor. SUBSTITUTE(f, x) returns the function f after x is substituted into f.

```
KERNEL_EXTRACT(F, k, n):
  repeat {
    K = ∪_{f∈F} K^k(f)
    I = I^0(K)
    for i = 1 to n {
      x = argmax_{y∈I} { area_value(y) }
      if area_value(x) < 0 exit;
      for all f ∈ FROM(x) }
        f = SUBSTITUTE(f, x)
      }
      F = F ∪ {x}
    }
  }
```

The algorithm takes as an input a Boolean network F. First, it generates all of the level-k kernels of all the functions in F. Then, it generates all the kernel intersections corresponding to the level-0 kernels of $IF(K)$. It picks the intersection with the best area value (one at a time, up to the specified parameter n) and substitutes it into all functions where the intersection is from. After n intersections are chosen, the algorithm repeats; if the best intersection at any step has a negative value, then the algorithm terminates. (A kernel intersection which comes from only a single function has a negative value; at this point, there are no multiple-cube divisors common to two or more functions.)

The reason for using the parameter k is to control the size of K so that the intersections can be generated within a reasonable amount of time. The tradeoff is that we may lose some *good* high-level kernels. Likewise, we generate only level-0 kernels of $IF(K)$ in order to reduce the size of the intersection set. Again, the tradeoff is that we may lose some *good* high-level intersections. (However, note that level-0 intersections are ones that maximally fan-out after substitution). The parameter n is used to control the number of times we need to regenerate all of the kernels and their intersections. Clearly, regenerating the kernels and their intersections after a single substitution is wasteful; only a few functions in the network have changed.

However, the reason for recomputing the kernels and their intersections is the following: after an intersection is substituted into the network, the values of some other intersections may change. So, we can only use the same set of kernel intersections up to a certain point where the values of the intersections become inaccurate. Table III shows how the choice of k and n effect the final results for one example. It also indicates that using low level kernels and more intersections in each iteration is nearly as effective and much more efficient. We have not experimented with using higher level intersections.

3) Single-Cube Extraction: Based on Proposition 1, the algorithm for generating kernels can also be used to generate the common subcubes among several functions. Given a set of functions

$$F = \{f_1, f_2, \ldots, f_n\}$$

we can build another function G such that

$$sup(G) = sup(f_1) \cup sup(f_2) \cup \cdots \cup sup(f_n)$$

$$G = \{c \mid \exists\, i \text{ such that } c \in f_i\}.$$

Proposition 3: The set of common subcubes in F is exactly $C(G)$, the co-kernels of the function G.

The generic algorithm for single-cube extraction is:

CUBE_EXTRACT(F):
repeat {
  c = FIND_A_COMMON_CUBE(F).
  $g = \{c\}$
  for all $f \in FROM(c)$ {
    f = SUBSTITUTE(f, g)
  }
  $F = F \cup \{g\}$.
} until no common cube can be found.

The routine FIND_A_COMMON_CUBE returns a common cube in F according to some criteria. Three possibilities come to mind.

1) Generate the set of all co-kernels of the function G, determine the literal savings for each co-kernel, return the co-kernel with the best literal savings.
2) Generate the level-0 co-kernels of G, rank them according to their literal savings, and choose the best one.
3) Generate any single level-0 co-kernel.

1) and 3) were implemented in MIS and comparable results were obtained on most examples even though 1) took, in general, much more time.

Table IV shows how the kernel extraction and cube extraction can be used. By combining them with the *resub*, which is described in the next section, we have a powerful optimization loop.

B. Resubstitution

Because of the heuristics used in the extraction process, we may have missed some common factors. *Resubstitution* is used to check whether an existing function itself is a divisor of other functions. For example, suppose the network is

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

$$y = a + b.$$

Function y itself is a divisor of x. Therefore, it can be used to simplify x. Function x can then be rewritten as:

$$x = y(c + d) + e.$$

This operation is called resubstitution. In particular, it is an algebraic resubstitution since $a + b$ is an algebraic divisor of $ac + ad + bc + bd + e = (a + b)(c + d) + e$.

However, algebraic techniques alone do not exploit all of the Boolean properties of the logic equations. To improve the results, we also perform Boolean resubstitution, which uses Boolean division when trying to substitute one function into another. Boolean resubstitution is capable of providing better results, but in general takes much longer than algebraic resubstitution. As an example, algebraic resubstitution does not simplify the following network:

$$x = (ab + cd)\bar{e}\bar{f} + (ab + ef)\bar{c}\bar{d} + (cd + ef)\bar{a}\bar{b}$$

$$y = ab + cd + ef.$$

However, using Boolean resubstitution, x can be rewritten as

$$x = y(\bar{a}\bar{b} + \bar{c}\bar{d} + \bar{e}\bar{f})$$

for a savings of 11 literals.

TABLE III
TRADEOFF'S IN KERNEL EXTRACTION

k	n	time	no. of literals in factored form	no. of kernels in first iteration	no. of intersections in first iteration
0	1	181.0	760	209	23
	2	93.1	767		
	5	45.9	759		
	10	26.8	773		
999	1	302.8	754	597	196
	2	172.0	754		
	5	98.0	766		
	10	72.3	773		

The example is *example* 3 from Section 7. Time is measured on VAX 8650 running Ultrix 1.2. k is the level of kernels used. n is the number of kernel intersection used before regenerating kernels.

TABLE IV
TYPICAL GLOBAL AREA OPTIMIZATION LOOP

MIS command	explanation	time	no. of literals in SOP form	no. of literals in factored form
rl ex3	read in a network	0.7	1804	1139
cx	extract common cubes	13.8	949	882
asb	algebraic substitution	6.6	666	650
el 0	eliminate nodes with 0 or less value	1.1	816	587
kx 5 0	kernel extract (k=0, n=5)	18.5	636	624
asb	algebraic substitution	3.1	625	613
el 0	eliminate nodes with 0 or less value	0.8	705	559
cx	extract common cubes	4.9	654	608
asb	algebraic substitution	3.1	652	607
el 0	eliminate nodes with 0 or less value	0.8	661	558
asb	algebraic substitution	1.7	656	556
el 0	eliminate nodes with 0 or less value	0.3	656	555

The example is *example* 3 from Section 7. Time is measured on VAX 8650 running Ultrix 1.2.

In general, the substitution of y into x is carried out by dividing x by y and checking if the resulting x is simpler (in factored form). In the Sections II-B-1 and IV-B-2, we present the algorithms for the algebraic and Boolean division which are implemented in MIS.

1) Algebraic Division: The following is a sketch of the algorithm for carrying out algebraic division with complexity of $O(n \log n)$, where n is the total number of terms in f and g.

ALG_DIV(f, g):
U = restriction of f to the literals in g
V = restriction of f to the literals not in g
/* note that $u_j v_j$ is the jth term of f */
$V_i = \{ v_j \in V \mid u_j = g_i \}$
$h = \cap V_i$
$r = f - gh$
return (h, r)

For example, if

$$f = ac + ad + bc + bd + e$$
$$g = a + b$$

then

$$U = a + a + b + b + 1$$
$$V = c + d + c + d + e$$
$$V_1 = v_j \in V \mid u_j = g_1 = c + d$$
$$V_2 = v_j \in V \mid u_j = g_2 = c + d.$$

So, *ALG_DIV*(f, g) returns

$$h = c + d$$
$$r = e.$$

Care is taken to make the algebraic division algorithm as fast as possible since it is used repeatedly in MIS. One way to accomplish this is to numerically encode the cubes of U, V, and g. Then, sorting these numbers, we can efficiently obtain the comparisons required to compute V_i and h, and by keeping track of the indices during the sorting process, we can also determine the remainder r easily.

The following observations are trivial but important in circumventing most of the divisions required for algebraic resubstitution.

The function f_j is **not** an algebraic divisor of f_i (i.e., $f_i/f_j = 0$) if

1. f_j contains a literal not in f_i;
2. f_j has more terms than f_i;
3. for any literal, the number of appearances in f_j exceeds that in f_i;
4. y_i is the transitive fan-in of f_j.

In some cases, we are not interested in the result of division if the quotient (f_i/f_j) is only a single cube. This can be detected sometimes by another useful filter:

5) If for any literal, the number of appearances for f_j equals that for f_i, then (f_i/f_j) is at most a single cube.

2) Boolean Division: Recall that division is defined in terms of multiplication and Boolean multiplication differs from algebraic multiplication in that the functions need not have disjoint support. Thus, given logic functions f, g, h, e, we said that g is a Boolean divisor of f if

$$f = gh + e \quad \text{and} \quad h \neq 0$$

i.e., the Boolean function formed by intersecting g and h and forming the union with e is a cover of f. Extending this to an incompletely specified function $ff = \{ f, d, r \}$, consisting of an ON-set, an OFF-set and a DC-set (don't-care set), division is stated as

$$f = gh + e \pmod{d}$$

meaning that the equivalence is not required on the don't-care set d (in other words, $gh + e$ need only be a cover of ff). We require either that g, h, and e be completely specified functions or are functions which all have d as the don't-care set. Note that if f is any cover of ff, then any factorization of f is also a cover of ff.

In applying Boolean division to multilevel logic minimization or factoring, there are two distinct problems to be solved. The first (and easiest) is, given a logic function g and an incompletely specified function ff, **compute** logic functions h and e (minimal in some sense) such that $gh + e$ is a cover of ff. The second problem is, given ff, **find** a function g such that $gh + e$ is a cover of ff and g, h, and e are minimal in some sense. The reason for minimal conditions on g, h, e is that the point of factoring or substitution is to find a minimal representation for ff. It is not hard to find divisors of ff, but it is hard to find divisors which lead to simple representations.

Theorem 2: Let $f_1 = hx + e$ be a cover of an incompletely specified function $ff = \{ f, d, r \}$. Suppose $\bar{x}g + x\bar{g} \subseteq d$, where g is any function. Then, $f_2 = hg + e$ is also a cover of ff.

The theorem can be proved by checking that the exclusive-OR of f_1 and f_2 is entirely contained in the don't care set.

Theorem 2 provides a solution to the first problem. To Boolean divide g into a completely specified function f, we can first form a don't-care $g\bar{x} + \bar{g}x$, where x is a new variable representing g. Next, we find a cover $hx + e$ (minimal in some sense) of the incompletely specified function $\{ f, x\bar{g} + \bar{x}g, (f \cup (\bar{x}g + x\bar{g}))' \}$. From Theorem 2, $hg + e$ is also a cover. The basic steps of the method for the division are as follows.

1) Use a new variable x to represent g.
2) Form the don't-care set $\bar{x}g + x\bar{g}$.
3) Minimize f using this new don't-care set.
4) Return (f/x, e) where e is the remainder, the terms of f not containing x.

As mentioned earlier, in minimizing multilevel logic, we can have several objectives. We have focused on minimal factored representations. We have seen that a minimal cofactor h and remainder e can be obtained by extending the don't-care set and using a two-level minimizer. However, this may not lead to a minimal fac-

tored representation. We next take one more step to reduce the support of the final representation which heuristically leads to better factorizations.

The procedure BOOL_DIV, shown below, is an implementation of Boolean division using two heuristics. The first is to restrict the function to its minimum literal support similar to that discussed in [4]. This is done by a call to procedure MINLIT passing as arguments the care onset f of ff and the care offset r of ff. MINLIT solves the relevant row covering problem and returns a slightly expanded cover, where the literals not in the minimum support have been eliminated. The second heuristic attempts to make the operation more efficient by first factoring out the algebraic cofactor $h = f/g$, thus reducing the problem to a Boolean division of the algebraic remainder e by the divisor g. Note that the expression hx (where x stands for a new variable) is added to the don't-care set in performing the Boolean division of the remainder e by g.

BOOL_DIV(f, g):
  (h, e) = ALG_DIV(f, g)
  $f = e$
  $DC = x\bar{g} + \bar{x}g + hx$.
  $r = \overline{f \cup DC}$.
  $f = r \cup DC$.
  f = MINLIT(f, r).
  f = EXPAND(f, r).
  f = IRREDUNDANT(f, DC).
  $h = h \cup f/x$.
  $e = f - f/x$.
return(h, e)

In general, Boolean division takes more time than the algebraic division since a Boolean minimization step with a proper don't-care set is involved. Hence, even more than for algebraic resubstitution, we need good filters to decide if Boolean resubstitution should be tried.

C. Phase Assignment

Each function in the Boolean network is represented as a positive logic expression. Also, each intermediate variable is assumed present in both its true and complemented form. In practice, there are many situations where implementing a function or its complement costs the same (e.g., static CMOS), and not all intermediate variables are needed in both phases. Hence, there is an optimization problem of choosing the phase of each intermediate function (i.e., to implement the function or its complement) in order to minimize the total number of inverters needed to implement the network. This is the *Global Phase Assignment* problem.

The following is a simple greedy algorithm for solving the global phase assignment problem. Since the algorithm runs very fast as compared with other algorithms, we call it QUICK_PHASE. F is the set of functions in the Boolean network.

QUICK_PHASE(F):
  repeat {
    x = *argmax*{*INVERTERS_SAVED*(f)}
    if (INVERTERS_SAVED(x)0) ≤ exit
    invert x
  }

INVERTERS_SAVED(f) computes the change in the number of inverters needed in the network if the function f were invented (i.e., complemented). For example, the network

$$f_1 = \bar{a}b$$
$$f_2 = \bar{f}_1 c$$
$$f_3 = \bar{f}_1 \bar{d}$$

requires a total of three inverters in this form (1 each for a, d, and f_1). However, if f_1 is inverted, only two inverters are required:

$$f_1 = a + \bar{b}$$
$$f_2 = f_1 c$$
$$f_3 = f_1 \bar{d}.$$

In static CMOS, each logic function is naturally inverting. This can be taken into account during the computation of INVERTERS_SAVED. However, for simplicity, we continue dealing with positive logic functions in this discussion.

QUICK_PHASE first computes the inverter savings for each function, and then chooses to invert the function which removes the most inverters from the network. If no inverters can be saved by inverting any function, then the algorithm has reached a local minimum and terminates.

One must be careful to compute properly the inverter savings of a function f. We now present the algorithm for INVERTERS_SAVED. The procedure USED(x, F) tests whether literal x is used in some function of F and *FANOUT*(x) and *FANIN*(x) are defined as

$$FANOUT(x) = \{ y \mid x \in y \quad or\ \bar{x} \in y \}$$
$$FANIN(x) = \{ y \mid y \in x \quad or\ \bar{y} \in x \}.$$

INVERTERS_SAVED(x):
  if USED(x, *FANOUT*(x)) and not USED($\bar{x}$, *FANOUT*(x)) *saving* = −1
  else if USED($\bar{x}$, FANOUT(x)) and not USED(x, *FANOUT*(x)) *saving* = 1
  else /* both x and $\bar{x}$ are used in *FANOUT*(x) */
    saving = 0
  for each $y \in$ *FANIN*(x) {
    if not USED($\bar{y}$, *FANOUT*(y)) {
      saving = *saving* − 1
    } else if not USED($\bar{y}$, *FANOUT*(y)-{x}) {
      saving = *saving* + 1
    }
  }
  return *saving*

The algorithm has two parts. It first computes the inverter savings at the output of x. If x is only used in its positive phase, inverting x results in one more inverter. If

x is only used in its negative phase $\bar{x}$, inverting x saves one inverter. If both phases of x are used, inverting x does not change the number of inverters. Computing inverter savings at the inputs of x is more complicated. For each input y of x, if y is only used in its positive phase, then one more inverter is needed. The only case when an inverter between y and x can be saved is when y is used only in its positive phase in all the functions y fans out to except x. In all remaining cases, the number of inverters does not change.

Although QUICK_PHASE is very fast, it often falls into a local minimum far removed from the global minimum. The final result largely depends on the initial state of the Boolean network. To allow for partial hill climbing, we present another algorithm which gives better results at the expense of more CPU time. This algorithm is called GOOD_PHASE (GP).

```
GOOD_PHASE(F):
  unmark all f ∈ F
  repeat {
    best = F
    QUICK_PHASE(F)
    if NUM_INVERTERS(F) < NUM_
       INVERTERS(best) { best = F
    }
    repeat {
      if all f ∈ F are marked {
        return best
      } else {
        x = argmax_{f∈F and f unmarked} {INVERTERS_SAVED(f)}
        invert x
        mark x
      }
    } until NUM_INVERTERS(F) < NUM_
        INVERTERS(best)
  }
```

GOOD_PHASE always keeps the best network found up to the current point (i.e., the network needing the least inverters). The call to QUICK_PHASE initially reaches a local minimum. Then, GOOD_PHASE enters the hill-climbing stage by inverting nodes even though the number of inverters in the network may increase. If, at any point, a solution better than the current best is found, we accept this solution and repeat. The algorithm terminates when all of the functions have been inverted once and no better solution has been found. This is a partial hill-climbing scheme because each function is inverted only once in a greedy order (minimum inverter increase).

Table V shows the difference between GOOD_PHASE and QUICK_PHASE for one example. GOOD_PHASE results in 25 percent fewer inverters, but takes more than 10 times as long to compute a result.

TABLE V
TRADEOFF'S BETWEEN DIFFERENT PHASE ASSIGNMENT ALGORITHMS

	no. of inverters	time
quick phase	31	7.8
good phase	23	89.4

The example is *example* 3 from Section 7. Time is measured on VAX 8650 running Ultrix 1.2. There are total of 56 functions in the network excluding inverters.

V. LOCAL OPTIMIZATION

Local optimizations refer to operations performed on a single node in the Boolean network, or locally in the surrounding neighborhood of the node. MIS uses the local transformations of decomposing large gates into smaller ones (*decomposition*), deriving better implementations of the gates (*factorization*), and simplifying each gate using knowledge of its local environment (*simplification*).

A. Factoring a Gate

In most design styles, an example of which is the complex-gate CMOS design, the implementation of a gate corresponds directly to a factored form of the logic function. Hence, a local optimization is to factor the logic equation of a single gate in order to produce an optimal pull-down (and pull-up network) for the gate. Our optimization criteria is the factored form with the least literals.

Different factoring algorithms have been explored. Each has its own use and and run-time cost. QUICK_FACTOR, the fastest factoring algorithm, is useful to estimate the cost of the implementation. Two other factoring algorithms, GOOD_FACTOR and BOOLEAN_FACTOR, are also implemented in MIS. Even though they are more expensive operations, they occasionally yield better factorizations, and are used to derive the final implementation of each gate.

Algorithms have been presented for exactly solving the problem of determining a factored form with a minimum number of literals [18]. However, the complexity of these exact techniques would appear to make them impractical for all but the smallest functions. Instead, we rely on fast, heuristic algorithms based on kernels to find optimal factored forms.

The generic factoring method is described by the following recursive procedure.

```
GENERIC_FACTOR(f):
  if (|f| = 1) return f
  k = CHOOSE_DIVISOR(f)
  (h, r) = DIVIDE(f, k)
  return GENERIC_FACTOR(k) GENERIC_
    FACTOR(h) + GENERIC_FACTOR(r)
```

The method first chooses a divisor of f and performs the division to obtain $f = kh + r$. At this level in the recursion, k, h, and r are expressions which must be recursively factored. Hence, in the last line, GENERIC_FACTOR is called for each of these expressions. The factored product plus the factored remainder is then formed and returned. Internally, the factored forms can be represented by either series-parallel trees or parenthesized expressions.

With very little overhead, we can modify the generic factoring algorithm to achieve a better result. The idea is in the following variation of the generic factoring schema.

```
GENERIC_FACTOR2(f):
  if (|f| ≤ 1) return f
  k = CHOOSE_DIVISOR(f)
  (h, r) = DIVIDE(f, k)
  if (|h| > 1) {
  k = CUBE_FREE(h)
  } else {
  k = ONE_LITERAL_OF(h)
  }
  (h, r) = DIVIDE(f, k)
  return GENERIC_FACTOR2(k) GENERIC_
  FACTOR2(h) + GENERIC_FACTOR2(r)
```

The heuristic used here is that having chosen the divisor k, we obtain h and are in the process of writing $f = kh + r$. Given that we are going to use factors k and h, we might as well collect everything that can be multiplied by h. Before this is done, we eliminate any literal factors of h (CUBE_FREE) in order to obtain the largest k possible. Then we perform the second DIVIDE to obtain this new k, which must include at least the old k. This can be quite effective if we did not choose the best divisor k initially (perhaps because of run-time considerations).

We can build several variations of factoring by choosing different algorithms for CHOOSE_DIVISOR and DIVIDE. There are two variations of CHOOSE_DIVISOR:

CHOOSE_LEVEL_0_KERNEL – pick any level-0 kernel (fast)
CHOOSE_BEST_KERNEL – compute all kernels and choose best one (slow)

and two variations of DIVIDE:

ALG_DIV - algebraic (weak) division (fast) (refer to Section IV-B1)
BOOL_DIV - Boolean (strong) division (slow) (refer to Section IV-B2).

These provide a spectrum of speed and quality tradeoff's for factoring. We have implemented the following three versions of factoring algorithms.

QUICK_FACTOR uses:

1. CHOOSE_LEVEL_0_KERNEL – pick any level-0 kernel
2. ALG_DIV – perform algebraic division.

QUICK_FACTOR uses the GENERIC_FACTOR2 algorithm. The extra division step in GENERIC_FACTOR2 attempts to improve the quickly chosen divisor. Typically, we find that this improvement step greatly improves the quality of the factored result.

GOOD_FACTOR uses:

1. CHOOSE_BEST_KERNEL – compute all kernels and choose best
2. ALG_DIV – perform algebraic division.

BOOLEAN_FACTOR uses:

1. CHOOSE_BEST_KERNEL – compute all kernels and choose best
2. BOOL_DIV – perform Boolean division.

To illustrate the different results that can be obtained by these various factoring algorithms, consider the function

$$f = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + dg.$$

Quick factoring (QF) and good factoring (GF) give different but similar quality factorings:

$$GF(f) = (c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce$$

$$QF(f) = g(a + d) + (a + b)(c + d + e) + c(e + f) + f(b + d)$$

but QF is much faster because it needs only to determine one level-0 kernel for each factor. However, the next example shows the difference in quality by QF, GF, and BF. For

$$f = bd + cd + be + ce + afd + afe + abg + acg + afg$$

the factored results are

$$QF(f) = (b + c)(d + e) + \big((d + e + g)f + (b + c)g\big)a$$

$$GF(f) = (b + c)(d + e + ag) + (d + e + g)af$$

$$BF(f) = (af + b + c)(ag + d + e).$$

Table VI shows the results of QUICK_FACTOR (QF), GOOD_FACTOR (GF), and BOOLEAN_FACTOR (BF) on a large set of functions. Good factoring is almost three times slower than quick factoring due to the extra time spent computing all of the kernels of the function before choosing a divisor. Boolean factoring is almost four times slower than good factoring due to the extra time spent performing a Boolean division (as opposed to an algebraic division).

B. Local Decomposition of a Gate

The operation *decomposition* is similar to factoring except that each divisor is formed as a new node in the Boolean network and the associated variable is substituted into the function being decomposed. Decomposition is one of the transformations used to break down large functions into smaller pieces, usually at a cost of a few more literals in the network.

For each method of factoring, we have the associated method for decomposition. For example, QUICK_DECOMPOSITION is similar to QUICK_FACTOR in that a single level-0 kernel of the fucntion is generated as the divisor. This kernel is implemented as a separate node,

TABLE VI
TRADEOFF'S BETWEEN DIFFERENT FACTORING ALGORITHMS

	no. of literals in factored form	time
quick factoring	1139	2.8
good factoring	1133	6.9
Boolean factoring	1108	28.9

The example is *example* 3 from Section 7. Time is measured on VAX 8650 running Ultrix 1.2. There are total of 56 functions factored.

and then the kernel, the quotient, and the remainder are all recursively decomposed.

GOOD_DECOMPOSITION generates all of the kernels before identifying the best kernel (i.e, the kernel saving the largest number of literals) to be extracted. Finally, BOOLEAN_DECOMPOSITION generates all of the kernels to choose the best kernel, and then uses Boolean division to divide the function by the kernel.

Decomposition can be combined with algebraic resubstitution to provide a means of finding common subexpressions. The method is

1. Apply quick decomposition to each node of the network.
2. Perform algebraic resubstitution of each node into every other node where possible.
3. Eliminate all single literal functions and all functions with small value (usually 0 or -1)

At the end of decomposition, each of the network cannot be further decomposed; each literal appears only once in each function. Resubstitution identifies, as a special case, nodes in the network which are identical. (The logic function at the node becomes just a single literal.) These trivial functions are eliminated along with the nodes with small value by an elimination step.

The motivation behind this technique for common subexpression elimination is that QUICK_DECOMPOSITION is very fast but still identifies good kernels for factoring each single function well. The kernels used for this become nodes of the Boolean network and resubstitution identifies common ones. Thus, the common divisors identified in this way are ones that are also near best for factoring. Of course, this is not always the best choice, and not all common divisors are found, but the method is very fast and the results are surprisingly good. A typical combination of decomposition, resubstitution, and elimination is shown in Table VII.

C. Simplification

Two-level minimization is a much more developed science than multilevel minimization, and very efficient algorithms exist for finding minimal two-level representations of Boolean functions. Two-level logic minimization plays an important role in multiple-level logic synthesis. Recall that each node in the Boolean network is represented in both a sum-of-products form and a factored form. Before processing the network, we can replace each sum-of-products expression with a minimal, equivalent representation. Two-level minimization can be made more powerful in this context (and slightly less of a procedure)

TABLE VII
TYPICAL LOCAL AREA OPTIMIZATION LOOP

MIS command	explanation	time	no. of literals in SOP form	no. of literals in factored form
rl ex3	read in a network	0.7	1804	1139
qd *	quick decompose each node	1.5	1373	1373
asb	algebraic substitution	13.6	947	947
el 0	eliminate nodes with 0 or less value	1.4	1463	807
asb	algebraic substitution	1.2	1361	902
decomp *	good decompose each node	4.7	895	895
asb	algebraic substitution	3.7	866	866
el -1	eliminate nodes with -1 or less value	1.5	1519	826
asb	algebraic substitution	1.1	1495	828
decomp *	good decompose each node	7.0	923	923
asb	algebraic substitution	4.2	896	896
el -1	eliminate nodes with -1 or less value	1.2	1368	797
asb	algebraic substitution	1.4	1343	790

The example is *example* 3 from Section 7. Time is measured on VAX 8650 running Ultrix 1.2.

by providing the minimizer with various don't-care sets derived from the immediate environment of a node.

We use simplification at several steps in the optimization procedure. First, to reduce our dependence on the initial multiple-level network, we perform a selective elimination on the network in order to produce fewer but larger functions. After this, each node in the network is replaced with a simpler form using SIMPLIFY1 as described below. Later, after common divisors have been identified and extracted, we use SIMPLIFY2 or SIMPLIFY3 to remove redundancy in the network. These algorithms identify redundancy which is local in the sense that it manifests itself over just a few levels of the network.

Our interest is in a minimal form which has the least literals in its factored form. Ideally, we would replace each node with an equivalent (it does not change the input–output behavior of the network) minimal representation. (A network which is minimal in this sense is called a prime and irredundant network [2].) Performing either of these two steps is usually not feasible and certainly not always necessary during each phase of the logic synthesis process. For the most part, we use the minimization methods of ESPRESSO [5] with different don't care sets generated according to how thorough and fast we want to be. Further, we minimize the function for the total number of literals in the sum-of-products form as an approximation to the number of literals in the factored form (since we have no way as yet for minimizing with this later objective in mind).

Several simplification procedures can be built from ESPRESSO. These vary only in the don't-care sets constructed.

```
SIMPLIFY1(f):
  DC = 0
  g = ESPRESSO(f, DC)
  if NUM_LIT(QF(g)) < NUM_LIT
    (QF(f)) { f = g
  }
  return f
```

NUM_LIT counts the number of literals in a given factored form. A node is replaced with the result if it is simpler than the existing representation as measured by the

resulting factored form which is computed using QUICK_FACTOR (QF). We must check the resulting factored form after minimization in order to guarantee that the function has become simpler. It is possible for the number of literals in the sum-of-products form to decrease while increasing the number of literals in a $x = pn^2$ SIMPLIFY1 is the most direct application to a single node f of the network. No don't-care sets (DC) is generated or used in the minimization.

SIMPLIFY2(f):
$DC = \bigcup_{x \in FANIN(f)} (x\bar{v}_x + \bar{x}v\,x)$
g = ESPRESSO(f, DC)
if NUM_LIT($QF(g)$) < NUM_LIT ($QF(f)$) { $f = g$
}
return f

In SIMPLIFY2, v_x is the variable representing node x in function f. The immediate fan-in don't-care set (that is, don't cares which result from the functions which directly fanin to f) is generated and used by the logic minimizer.

The final method uses even more of the don't-care set, including the immediate fan-out don't-care set (that is, don't-cares which result from the functions which directly depend on f).

SIMPLIFY3(f):
$DC_i = \bigcup_{x \in FANIN(f)} (x\bar{v}_x + \bar{x}\,v_x)$
$DC_o = \bigcup_{x \in FANOUT(f)} ((x/f)(x/\bar{f}) + (\overline{x/f})(\bar{x}/\bar{f}))$
g = ESPRESSO(f, $DC_i \cup DC_o$)
if NUM_LIT($QF(g)$) < NUM_LIT ($QF(f)$) { $f = g$
}
return f

VI. Timing Pre-Optimization

Our approach in multiple-level logic optimization is to minimize first the area without concern for the delay. Then at the beginning of a timing optimization iteration, we have a network whose area is minimized (i.e., all the global factors have been extracted out). Given the system timing requirement for a network, the goal of timing optimization is to reduce the delay through the network with minimum area increase.

The timing optimization process iterates over several major operations. First, given the signal arrival time of all primary inputs, the delay of each signal (intermediate variable or primary output) is computed. Then, based on the required time of all the primary outputs, the slack value of each signal (intermediate variable or primary input) is computed. Now, the critical part of the network (which is defined precisely later on) can be identified. In the DAG representation of the network, the critical path of the network can be viewed as a set of directed path from primary inputs to primary outputs. To reduce delay through all the critical path, we find a set of nodes which form the minimum weighted cut-set of all the critical path (to be defined later), and reduce the delay through each node in the set. The whole process repeats until the timing constraints are satisfied, or until it is apparent that they cannot be satisfied.

There are several crucial steps in the timing optimization loop. The first problem is to estimate the delay of the network. Since any timing estimates without actual layout information are necessarily inaccurate, we aim at a delay model which is fast to evaluate and gives good relative measure as well as reasonably accurate absolute measure of the speed of the circuit. The next problem is to derive a weight function for the nodes which can reflect the tradeoff between area and delay. Finally, using the delay estimate and the weight function, the critical path of the network (which is defined precisely later on) can be identified and then restructured.

In the following sections, we describe the algorithms and approaches as applied to static CMOS. With few modifications, these techniques can be applied to other technologies as well. In static CMOS technology, a critical path from a primary input to a primary output consists of alternating falling and rising signals. To get a better delay estimate, the worst-case fall and rise propagation delay of a gate with respect to each input are computed separately. The delay, the required time, and the slacks of all the signals are also computed separately with respect to the falling and rising transitions.

A. Delay Estimate

Given a static CMOS gate (logic function in its factored form) and its local environment (e.g., number of gates it drives), we estimate for each input the worst-case delay through the gate. The delay is due to the charging or discharging of the load capacitance through the pull-up or pull-down network of transistors. As an example of estimating the fall delay from a particular input to the output of a gate, the worst case corresponds to discharging the load capacitance on the longest path through the pull-down network from the output to the ground through a transistor controlling that input. We take the following approach to estimate the delay through each gate.

First, for each input of a gate, we translate the series-parallel transistor network (i.e., the pull-down or pull-up network depending on whether we are estimating the fall or rise delay) into an equivalent chain of series-connected transistors corresponding to the longest path with respect to that input. Then, we parameterize the longest chain by a) the number of transistors in series, b) the transistor width, c) the load capacitance, and d) the average number of transistors in parallel with each of the series-connected transistors. We define

- n number of series-connected transistors,
- w transistor width,
- c load capacitance,
- p average number of transistors parallel with each transistor on the path,

using a simple RC model, where

$$R \propto n/w$$
$$C \propto (npw, w, c).$$

Thus

$$delay \propto RC \propto pn^2,\ n,\ nc/w$$

and with

$$y = n$$

$$z = nc/w$$

the delay is then modeled with a polynomial function of variables x, y, and z,

$$delay = P(x, y, z).$$

The coefficients of the delay equation are determined by a least-square fit of the delay data taken from a large set of simulations as the parameters n, w, p, and c are varied. Experiments have shown that this model gives reasonable estimates of delay for our purposes.

B. Node Weights

The objective of the timing optimization step is to reduce the delay through the network with minimum area increase. Recall that the area value of a node is an estimate of the total area increase if the node were eliminated from the network. In reducing the delay through a network, we often need to eliminate some nodes in order to reduce the number of levels of the network. We therefore want to pick such nodes whose elimination results in the least area increase. Furthermore, we do not have to eliminate a node completely from the network in order to reduce the number of levels. We only need to eliminate the node from the "critical" part of the network. So, the area value of a node needs to be modified to reflect the area increase correctly when the node is to be partially eliminated. This is the weight of a node.

To be precise, let

$$CRITICAL\_FANOUT(f) = \{ g \in FANOUT(f) \mid f \text{ is to be eliminated from } g \}.$$

The weight of a node f is defined as

$$weight(f) = \begin{cases} area\_value(f) & \text{if } CRITICAL\_FANOUT(f) = FANOUT(f) \\ \left(\sum_{g \in CRITICAL\_FANOUT(f)} N(g, f) \right)^* L(f) & \text{if } CRITICAL\_FANOUT(f) \neq FANOUT(f) \end{cases}$$

where $N(g, f)$ is again the number of times either f or $\bar{f}$ appears in the factored form for g, and $L(f)$ is the number of literals in the factored form for f.

C. Optimization Loop

To reduce the delay through the network, we need to identify the critical part of the network. First, associate some timing information with each signal (the output of a function). Let

$R_r(f)$	time at which signal f must be switched from low to high,
$R_f(f)$	time at which signal f must be switched from high to low,
$A_r(f)$	time at which signal f actually switches from low to high,
$A_f(f)$	time at which signal f actually switches from low to high,
$S_r(f)$	slack of the rising signal f (is equal to $R_r(f) - A_r(f)$),
$S_f(f)$	slack of the falling signal f (is equal to $R_f(f) - A_f(f)$),
$D_r(f, x)$	low to high propagation delay of f with respect to input x,
$D_f(f, x)$	high to low propagation delay of f with respect to input x,
$R(f)$	$R_r(f)\ R_f(f)$
$A(f)$	$A_r(f)\ A_f(f)$
$S(f)$	$S_r(f)\ S_f(f)$
$D(f, x)$	$D_r(f, x)\ D_f(f, x)$.

Initially, $R(f), f \in F$, are known for all the primary outputs and $A(f), f \in F$, are known for all the primary inputs. (These can be determined by a timing analysis of the system in which the logic must function.) The following formula can be used to compute $R(f)$, $A(f)$, and $S(f)$ for all signals f in the network efficiently:

$$A_r(f) = \max_{x \in FANIN(f)} \{ A_f(x) + D_r(f, x) \}$$

$$A_f(f) = \max_{x \in FANIN(f)} \{ A_r(x) + D_f(f, x) \}$$

$$R_r(f) = \min_{x \in FANOUT(f)} \{ R_f(x) - D_f(x, f) \}$$

$$R_f(f) = \min_{x \in FANOUT(f)} \{ R_r(x) - D_r(x, f) \}$$

$$S_r(f) = R_r(f) - A_r(f)$$

$$S_f(f) = R_f(f) - A_f(f).$$

Since the slack is the difference between the required and actual time of a signal, the signals with zero or negative slacks are critical. Now, we are ready to identify the critical part of the network. Let F be the network and $CRITICAL(F)$ be the critical part of the network,

$$CRITICAL(F) = \{ f \in F \mid S_r(f) \leq 0, \text{ or } S_f(f) \leq 0 \}.$$

In addition, we define a path in the network F to be

$$PATH(F) = \{ (x_1, x_2, \cdots, x_n) \mid x_i \in F, x_i \in FANIN(x_{i+1}), x_1 \in PI, x_n \in PO \}$$

where *PI* is the set of primary inputs and *PO* is the set of primary outputs. The idea behind all these definitions is that we want to find a minimum weighted set of nodes which cut all the critical paths in the network. The following is a definition for the minimum weighted cut-set:

$$MIN\_WEIGHTED\_CUTSET(F) = \underset{S \subseteq F \text{ and } PATH(F-S)=\varnothing}{argmin} \left\{ \sum_{x \in S} weight(x) \right\}.$$

The delay of the network F can be reduced if the delay through every node x in $MIN\_WEIGHTED\_CUTSET(F)$ is reduced. The following timing optimization loop summarizes our approach.

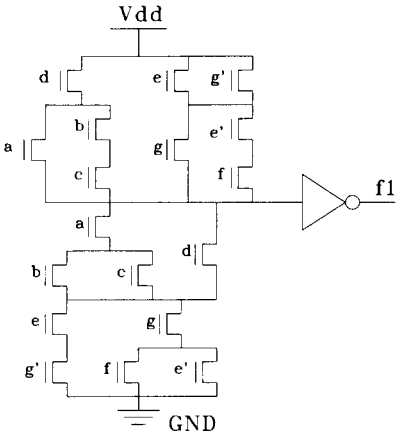
TIMING_OPT(F):
while (actual delay > required delay) {
  compute $A(f)$, $R(f)$, and $S(f)$ for all $f \in F$
  $G = CRITICAL(F)$.
  compute $weight(g)$ for all $g \in G$
  $X = MIN\_WEIGHTED\_CUTSET(G)$.
  $REDUCE\_DELAY(x)$ for all $x \in X$
}

The $REDUCE\_DELAY(x)$ tries to reduce the delay through a gate x by either resizing the transistors in x, refactoring x, decomposing x into smaller gates, or combining several small gates around x into a larger gate.

Because our timing estimates are necessarily inaccurate, we view the result of the optimization not as a precise delay calculation, but rather as a reasonable guide for restructuring the architecture of the network to meet the timing constraints. Also, the timing estimates are reasonable specifications for the module generators to use when synthesizing and placing the gates. More accurate timing estimates and verifications may be employed later in the design cycle when the details of the gate designs and placements are known more precisely.

VII. Results

The algorithms described here have been implemented in a computer program *mis*. *mis* is written in C and has been run on Unix-based workstations (SUN/3, DEC GPX/II, IBM PC/RT) and DEC and IBM mainframes. In the Berkeley synthesis system, *mis* reads a network description from the *Oct* database, and writes the optimized logic network back into the database. To interface to systems and environments where *Oct* is not used, *mis* also reads and writes the standardized logic intermediate format (LIF) files. In the interactive mode, *mis* provides a large set of commands for manipulating a logic network, including the powerful optimizations presented in this paper. A partial list of the commands available in *mis* is given in Fig. 4. *mis* also supports a batch mode, where the optimization is controlled by a "script" of MIS commands. The script can be hand-designed by the user for a particular problem, or one of several standard scripts which have been developed can be used. A sample script is given in Fig. 5



f1 = (a(b+c)+d)(eg'+g(e'+f))

Fig. 3. Implementation of the factored form as a complex gate.

	GLOBAL OPTIMIZATION
kextract	identify common divisors using KERNEL_EXTRACT
cextract	extract single cube factors using CUBE_EXTRACT
eliminate	eliminate nodes below threshold
collapse	collapse the network to two-level
goodphase	minimize number of inverters using GOOD_PHASE
quickphase	Minimize number of inverters using QUICK_PHASE
aresub	resubstitute node to all other nodes (Algebraic)
bresub	resubstitute node to all other nodes (Boolean)
sweep	eliminate 0-fanout or 1-fanin nodes
	LOCAL OPTIMIZATION
bdecomp	Decompose nodes using BOOLEAN_DECOMPOSITION
gdecomp	Decompose nodes using GOOD_DECOMPOSITION
qdecomp	Decompose nodes using QUICK_DECOMPOSITION
bfactor	factor nodes using BOOLEAN_FACTOR
gfactor	factor nodes using GOOD_FACTOR
qfactor	factor nodes using QUICK_FACTOR
simplify0	simplify using fast minimization algorithm
simplify1	simplify using ESPRESSO
simplify2	simplify using ESPRESSO with don't care set
strongd	Boolean (strong) divide one node by another
invert	invert nodes
addinv	add inverters as needed
	TIMING OPTIMIZATION
printdelay	print gate delays
printrtime	print requirecd time of a signal
printatime	print the actual time of a signal
printweight	print node weights
printcutset	print minimum weighted cutset
printcp	print critical pathes
printcs	print critical pathes with slacks
setatime	set actual time of primary inputs.
settime	set required time of primary outputs.
delay	calculate delays
reduce	reduce the delay through a critical node
setslack	set the output slacks
speedup	speed up the network by percentage
	MISCELLANEOUS
backup	backup the current copy of network
lclose	close the current log file
lopen	open a log file
restore	restore the backup copy of the network
source	execute MIS commands in a file
undo	undo last command that changed network
verify	verify the Boolean equivalence of two networks

Fig. 4. Partial list of MIS commands.

A test of the algorithms described in MIS was performed at Advanced Micro Devices on some industrial circuits. The function of each of these circuits was taken from actual chip designs, and had previously been designed and optimized manually. In each case, the logic network was designed starting from a BDSYN description of the behavior of the network.

The target technology was a fixed library of NAND and NOR gates with up to four inputs per gate—no complex gates were used in the design. The cost functions used in

Command	Description
sweep	general 'clean-up' -- constant propogation, eliminate *dead* gates, double-inverter elimination
simplify1 *	Two-level minimization of each node (using Espresso)
aresub	Algebraic resubstitution -- make maximal use of divisors currently in the network
kextract 10 9	Kernel extract to find common divisors
cextract	Cube extract to find common cubes
aresub	Algebraic resubstitution to find any other common factors
eliminate 0	Selective collapse (eliminate nodes with value less than 0)
kextract 5 9	Repeat basic extraction steps
cextract	
aresub	
eliminate 0	Selective collapse
gdecomp *	Break apart any remaining large nodes
aresub	Check for common factors
eliminate 0	Repeat basic extraction step
simplify1 *	Minimize each node
kextract 5 9	
cextract	
gdecomp *	Final decomposition of any large nodes
aresub	

Fig. 5. A typical MIS script.

TABLE VIII
MIS RESULTS COMPARED TO MANUAL DESIGN

Example	no. of inputs	no. of outputs	Manual Design Gate Count	Manual Design Device Count	MIS Design Gate Count	MIS Design Device Count	Ratio of Devices
ex1	8	7	54	245	46	206	0.84
ex2	25	18	95	406	69	306	0.75
ex3	14	14	162	814	445	1965	2.50
ex3b	14	14	162	814	-	1250	1.53
ex3c	14	14	162	814	-	900	1.10

the optimizations performed by MIS are currently oriented towards CMOS complex gate design. In particular, the final output from MIS is in the form of a Boolean network consisting of an arbitrary CMOS complex gate at each node. Hence, a manual translation was performed on the output of MIS to this technology. No optimization or merging of gates was done during the translation, even though it would be highly desirable to do so. The results of this experiment are given in Table VIII.

The results from the first two examples are very encouraging. The difference in the third example is explained, in part, from the fact that MIS is currently unable to use the don't-care information inherent in the multiple-level description of the network. In a second experiment, listed as "ex3b," the network was minimized as a two-level PLA with the same don't-care set used by the designer for his implementation. This minimized PLA was then re-extracted into a multiple-level network using MIS. The resulting network was not mapped into the target technology for a direct comparison; however, it appears to be about two-thirds of the size of the original network produced by MIS. In a third experiment, listed as "ex3c," we used MIS interactively, trying some of the more powerful operators, especially Boolean resubstitution, and obtained a further reduction to an estimated 10-percent increase over the manual design.

Given that the technology mapping was performed in a straightforward manner, we expect this result to be a lower bound on what can be achieved with MIS and, hence, this result is encouraging. We are currently looking at algorithms for optimizing the translation from an optimized Boolean network into a fixed-cell library (e.g., a standard cell library).

VIII. Conclusions and Future Directions

We have presented a software system (MIS) for multiple-level logic optimization and a set of algorithms on which this system is based. Currently, the system is being distributed to a limited number of users. The algorithms manipulate a Boolean network structure yielding good multilevel implementations, minimizing area while taking into account timing constraints. The set of algorithms included in MIS are characterized by different versions trading off speed and quality of results. Operated in the fast mode, MIS can be used to give quick estimates of area and timing for combinational logic groups.

MIS is an evolving system and many directions are being explored to improve the quality of the final network, and the performance of the algorithms. Among others, we are exploring:

1) The use of Rectangle Covering [8] to improve the quality and performance of both the kernel extraction algorithm and the cube extraction algorithm.
2) Algorithms for the optimal mapping of a Boolean network (after technology independent optimization by MIS) into a fixed-library of gates (for example, a standard-cell gate library). As mentioned in the previous section, we believe that the inclusion of a powerful global algorithm for the technology mapping problem will improve the quality of the results produced by MIS.
3) Improving the timing analysis and timing optimization parts of MIS to find the correct set of transformations to be performed on the network in order to best satisfy the timing constraints.
4) The use of a self adapting script with back-trapping capability to be able to have a single universal script.
5) The use of better filters for Boolean resubstitution.
6) Methods for restricting the don't-care set during simplification to only a "useful" subset, especially when the complements used during simplification are estimated to be too large.
7) The incorporation and use of don't-cares in MIS derived from the input specification.

Acknowledgment

Many people have participated in the development of MIS. The authors thank E. Detjens of Phillips Research Labs, and S. Krishna, T. Ma, P. McGeer, Li-Fan Pei, and R. Yung of Berkeley for work on various parts of MIS. R. Segal was responsible for the program BDSYN. N. Phillips, of Digital Equipment Corporation, and P. McGeer were responsible for the N.2 to BDS translator. N. Phillips also provided assistance with the DECSIM simulator and the BDS language.

REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. on the Theory of Switching*, Apr. 1957.

[2] K. Bartlett, R. Brayton, R. Jacoby, G. Hachtel, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multiple-level minimization using implicit don't-cares," in *Proc. Int. Conf. Comp. Des. (ICCD-86)* (Port Chester, NY), Oct. 1986, pp. 552-557.

[3] R. K. Brayton and C. McMullen, "The decomposition and facorization of Boolean expressions," in *Proc. Int. Symp. Circ. Syst. (ISCAS-82)* (Rome), May 1982, pp. 49-54.

[4] R. K. Brayton and C. McMullen, "Synthesis and optimization of multistage logic," in *Proc. Int. Conf. on Comp. Des. (ICCD-84)* (Rye), 1984, pp. 23-28.

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

[6] R. K. Brayton, N. L. Brenner, C. L. Chen, G. De Micheli, C. T. McMullen, and R. H. J. M. Otten, "The Yorktown silicon compiler," in *Proc. Int. Symp. Circ. Syst. (ISCAS-85)* (Kyoto), June 1985.

[7] R. Brayton, E. Detjens, S. Krishna, T. Ma, P. McGeer, L. Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, R. Yung, and A. Sangiovanni-Vincentelli, "Multiple-level logic optimization system," in *Proc. IEEE Inf. Conf. on CAD (ICCAD)* (Santa Clara, CA), Nov. 1986, pp. 356-359.

[8] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multiple-level logic optimization and the rectangular covering problem," *ICCAD-87*, submitted for review.

[9] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBMJRD 25*, pp. 272-280, July 1981.

[10] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBMJRD*, Sept. 1984.

[11] A. J. de Geus and W. Cohen, "A rule-based system for optimizing combinational logic," *IEEE Design & Test*, pp. 22-32, Aug. 1985.

[12] *VAX DECSIM Reference Manual*, Digital Equipment Corp., Dec. 1985.

[13] J. Dussault, C. Liaw, and M. Tong, "A high level synthesis tool for MOS chip design," in *Proc. 22nd Design Automation Conf.* (Albuquerque), June 1984.

[14] *N.2 ISP' Reference Manual*, Endot Corp., 1986.

[15] D. Harrison, P. Moore, R. Spickelmier, and A. Newton, Data management and graphics editing in the Berkeley design environment, in *Proc. IEEE Int. Conf. on CAD (ICCAD)*, 1986, pp. 24-27.

[16] M. E. Hofmann, "Automated synthesis of multi-level combinational logic in CMOS technology," Ph.D. thesis, Univ. of California, 1985.

[17] T. Hoshino, M. Endo, and O. Karatsu, "An automatic logic synthesizer for integrated VLSI design system," in *Proc. Cust. Int. Circ. Conf. (CICC-84)* (Rochester), May 1984, pp. 356-360.

[18] E. L. Lawler, "An approach to multilevel Boolean minimization," *J. ACM*, pp. 283-295, 1964.

[19] D. Patterson and C. Sequin, "A VLSI RISC," *Computer*, vol. 15-9, pp. 8-21, 1982.

[20] D. Patterson *et al.*, "Design decisions in SPUR," *IEEE Computer*, pp. 8-22, Nov. 1986.

*

Robert K. Brayton (M'75-SM'78-F'81), photograph and biography unavailable at the time of publication.

*

Alberto Sangiovanni-Vincentelli (M'74-SM'81-F'83), for a photograph and biography, please see page 693 of the September issue of this TRANSACTIONS.

*

Richard Rudell, for a photograph and biography, please see page 750 of the September issue of this TRANSACTIONS.

*

Albert R. Wang received the B.S. degree in computer science and applied mathematics from the University of California, San Diego, in 1984. He is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley. His current research interests include multiple-level logic synthesis and optimization, multiple-level logic verification, and sequential logic synthesis and optimization.