# Factoring logic functions

by R. K. Brayton

**A factored form is a representation of a logic function that is either a single literal or a sum or product of factored forms. Thus it is equivalent to a parenthesized algebraic expression. It is one of many possible representations of a logic function, but seems to be the most appropriate one for use in multilevel logic synthesis. We give a number of methods for obtaining different factored forms for a given logic function. These methods range from purely algebraic ones, which are quite fast, to so-called Boolean ones, which are slower but are capable of giving better results. One of the methods given is both fast and gives good results, and is useful in providing continuous estimates of area and delay as logic synthesis proceeds. In multilevel logic synthesis, each of the methods given has a use in a system where run-time and quality are traded off. We also formulate the problem of optimal algebraic factorization, and pose its solution as a rectangle-covering problem for which a heuristic method is given.**

## 1. Introduction

A Boolean variable is a symbol labeling a single coordinate of a Boolean space. A literal is a variable or its negation (e.g., $a$ or $a'$). A cube is a product of literals. A disjunctive form is a sum of cubes. A conjunctive form is a product of sums of literals. Each of these represents logic function. This paper is about another form for representing logic functions, factored forms, and procedures for obtaining factored forms from logic functions.

A *factored form* is either a literal or a sum or product of factored forms. For example, each of the following is a factored form:

$a$,
$a'$,
$abc'$,
$abc' + cd$,
$(a(b + c) + b'(a'e + c'(e' + f)))(b'c + efg)$,

where $a$, $a'$, $b$, $b'$, $\cdots$, $g$, $g'$ are literals. The first two expressions are also literals, cubes, and disjunctive and conjunctive forms. The third is a cube and disjunctive and conjunctive form and the fourth is a disjunctive form. Factored forms include all these other forms.

A factored form is a representation of a logic function with some attractive properties. Like Bryant's differential form [1], the factored form represents both a function and, by duality, its complement. For multilevel logic synthesis, it often estimates how the logic function may be implemented better than other representations. For example, the function

$$x = ae + af + ag + bce + bcf + bcg + bde + bdf + bdg,$$

represented here in disjunctive form, has 24 literals, but factored,

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

has only seven literals. The factored form represents better the complexity of this function.

Factored forms are important for logic synthesis systems since they can be used for accurately estimating area and delay if the factoring can be done quickly with good results. Factoring methods are closely related to techniques for finding common subexpressions, which are important in optimally synthesizing logic as a multilevel logic network.

This paper provides some of the details concerning algorithms, implementation, and general strategy for factoring logic functions. Several logic synthesis systems [2–6] have been based on these or similar ideas, sometimes called weak division. The algorithms were developed at IBM Research in Yorktown [7–9], with some later development and refinements at Berkeley [10]. The concepts for factoring, and similar concepts for logic synthesis, have been tested on many sets of logic examples of various types.

In this paper, we give several methods for factoring a logic function. The methods range from those which are extremely fast to those which are slower but give more optimal results. The paper is divided into five sections. We begin by discussing division, since factoring and division are closely related. Section 2 describes two notions of division as applied to logic functions, algebraic division and Boolean division. Section 3 discusses kernels, which are an important subset of the algebraic divisors. Section 4 gives three variations of generic factoring and a new fast method QF which gives good results. QF has become an important method since it provides a means, during logic synthesis, for continually representing all logic functions as factored forms. In Section 5, we relate optimal algebraic factoring to a rectangle-covering problem and provide some heuristics for its solution.

## 2. Algebraic and Boolean division

We say that a logic function $g$ is a *Boolean divisor* of $f$ if

$$f = gh + r,$$

where $h$ and $r$ are logic functions and $gh \neq 0$. Similarly, we say that $g$ is a *Boolean factor* of $f$ if $f = gh$. Thus a divisor is a factor of a subset of $f$.

In the above definitions, any manipulation of $f$ that produces an equivalent logic function is allowed. For example, the function $x = a + bc$ can be expressed as $x = (a + b)(a + c)$, but in order to achieve this, knowledge about logic functions is necessary. On the other hand, if $x = ac + ad + bc + bd$, then expressing $x = (a + b)(c + d)$ requires only algebraic factoring. We have use for both types of manipulations. The true logic manipulations require more time but in principle can achieve superior results. On the other hand, the algebraic manipulations can be made much faster and in many cases give almost as good results. In logic synthesis each kind is used selectively in order to achieve a balance between run-time efficiency and quality of results.

• *Algebraic division*

In what follows, we are careful to distinguish a logic function from any of its many different algebraic representations. We focus partly on disjunctive forms which are minimal with respect to single-cube containment. This means that any term (cube) in the disjunctive form is not contained in any other single term. In practice we normally use disjunctive form representations that are irredundant sums of prime implicants, but these requirements are not strictly necessary. Formally, a disjunctive form which is minimal with respect to single-cube containment is called an algebraic or logic *expression* or sometimes just an expression. Thus an expression is a particular kind of representation of a logic function. Algebraic, not logic, operations on these expressions transform them into algebraic factorizations.

In order for the manipulation of algebraic expressions to remain in the domain of algebraic expressions, *multiplication* of two expressions $g$ and $h$ is defined only when the sets of variables on which $g$ and $h$ depend are disjoint sets. Thus the multiplication of $a + b$ with $a' + c$ is not defined in the algebraic domain since we would have to know that $aa' = 0$ to obtain another algebraic expression. However, $a + b$ multiplied by $c + d'$ is defined and the resulting expression is obtained by simple polynomial expansion:

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

The disjunctive form associated with a product of two expressions is an algebraic expression and is unique. Hence the process of multiplication is invertible and we can extend the notion of division.

We say that a logic expression $g$ is an *algebraic divisor* of a logic expression $f$ if

$$f = gh + r,$$

where $h$ and $r$ are logic expressions and $h$ is not null. Here equality is meant in the sense that the product is defined and that when we expand the expressions of the right-hand side, we obtain exactly the same set of terms as in $f$. Similarly, we say that $g$ is an *algebraic factor* of $f$ if $f = gh$.

Some useful relations between Boolean divisors and algebraic divisors of the function and its complement are given below.

*Lemma 1* A logic function $g$ is a Boolean factor of a logic function $f$ if and only if $fg' = 0$, i.e., $g' \subseteq f'$ ($f \subseteq g$).

*Proof* If $f = gh$, then $g'f = 0$. Conversely, $g'f = 0$ implies that $f = fg + fg' = fg = g(f + r)$, where $r \subseteq g'$, i.e., $f = gh$. □

*Lemma 2* $g$ is a Boolean divisor of $f$ if and only if $fg \neq 0$.

*Proof* Since $f = fg + fg' = (f + k)g + r$, where $k \subseteq g'$, then if $fg \neq 0$, $f = hg + r$, where $gh \neq 0$. Conversely, $f = gh + r$ implies that $fg = gh + gr$. Since $gh \neq 0$, then $fg \neq 0$. □

*Lemma 3* If $f$ is a sum of prime implicants, and $g$ is an algebraic divisor of $f$, then $g'$ is a Boolean divisor of $f'$.

*Proof* $f = gh + r$ implies that $f' = g'r' + h'r'$. If $g'r' = 0$, then $f' = h'r'$ or $f = h + r$, implying that $f$ was not a sum of primes. Thus $g'r' \neq 0$ and hence $g'$ is a Boolean divisor of $f'$. □

*Lemma 4* If $g$ is an algebraic divisor (algebraic factor) of $f$, then $g$ is a Boolean divisor (Boolean factor) of $f$.

*Proof* Trivial.

The first two lemmas show that for any logic function $f$ there are many Boolean factors and divisors; in fact, *any* function containing $f$ is a Boolean factor of $f$ and *any* function not orthogonal to $f$ is a Boolean divisor of $f$. This poses a problem in choosing a best divisor since there are so many. Lemma 4 states that the algebraic divisors are a subset of all the Boolean divisors. Experimentally, the algebraic divisors have provided a reasonably good subset from which to choose. The fact that these can be determined quite quickly, as we shall see, makes them good candidate divisors in a logic synthesis system, for either factoring the logic functions or finding common divisors among many functions.

In general, we face two tasks in using either notion of division for factoring. First, to find a good candidate divisor, and second to effect the division, i.e., to determine, given $g$ and $f$, the coefficient $h$ and remainder $r$ so that $f = gh + r$.

For notation later, we define the *quotient* of $f$ and $g$ (denoted $f/g$) as the largest algebraic expression $h$ such that $f = gh + r$. Thus $f = (f/g)g + r$, and the *remainder* has the property that $g$ is not one of its algebraic divisors.

The following is a sketch of the algorithm for carrying out algebraic division; given $f$ and $g$ it returns the quotient $h$ and remainder $r$:

```
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 ∈ V: u_j = g_i}.
h = ∩ V_i.
r = f − gh.
return (h, r).
```

Care should be taken to make this algorithm as fast as possible since it is a key subroutine of many of the algorithms used in a logic synthesis system. One way to accomplish this with complexity of $O(n \log n)$, where $n$ is the total number of terms in $f$ and $g$, is to numerically encode the cubes of $U$, $V$, and $g$. By sorting these numbers, the comparisons required to compute $V_i$ and $h$ can be made efficient, and by keeping track of the indices during the sorting process, the remainder $r$ is an easy consequence.

- *Boolean division and Boolean procedures*

By restricting attention to algebraic expressions for Boolean functions, very fast methods for transforming networks of logic functions are possible. However, the full power of Boolean function manipulation is needed also.

For instance, the Boolean function

$$f = a'b + a'c + a'd + b'a + b'c + b'd + c'a + c'b + c'd + d'a + d'b + d'c$$

can be factored algebraically as

$$f = (d + c)(a' + b') + (a + b)(c' + d') + c'd + a'b + cd' + ab'.$$

However, $f$ can be expressed more simply as

$$f = (a + b + c + d)(a' + b' + c' + d').$$

Since the two factors above have a common set of variables, the last result could not have been achieved by the algebraic methods.

Boolean procedures are those that use the identities of Boolean algebra such as $aa' = 0$, $aa = a$, and $a + a' = 1$. The two-level logic minimization algorithm, ESPRESSO-II [11], and subprocedures for manipulating logic, such as COMPLEMENT, TAUTOLOGY, REDUCE, etc., are Boolean procedures. Boolean procedures are typically slower than the algebraic procedures and hence must be used discriminantly.

Since, in practice, we are interested in expressing a logic function in its simplest form, we would like a procedure for Boolean division which returns $f = gh + r$, where $h$ and $r$ are as simplified as possible. In this section we discuss some of the algorithms which are used to define such a procedure for Boolean division.

*Minimizing logic functions with different objectives*

The representation resulting from minimizing a logic function $f$ using a heuristic logic minimizer, such as ESPRESSO-II, is dependent on the heuristics and procedures used. Even with exact minimizers, such as McBOOLE [12] or ESPRESSO-EXACT [13], the result is only guaranteed to have a minimum number of terms. Typically, two-level logic minimizers have as their primary objective the minimum number of terms, because the output of the minimizer is usually implemented as a PLA where the area is proportional to the number of terms. However, with multilevel logic as the method of implementation, the number of terms is one of the least important objectives. Rather, minimization objectives such as the number of

- literals
- variables that the function depends on (variable support)
- literals that the function depends on (literal support)
- literals in a factored form

are more directly related to the implementation. The last objective, which is probably the most important one, indicates that even the notion of finding a minimum (in some sense) disjunctive form is not sacrosanct. The middle

two objectives reflect a relation between the least number of variables or literals and good factoring or easier wiring problems. These are called minimum support objectives and are used as heuristics in defining an algorithm for Boolean division.

*Minimum support algorithms*

Let $ff = \{f, d, r\}$ be an incompletely specified logic function, where $f$ is the onset, $d$ the don't-care set, and $r$ the offset. Suppose that $F$, $D$, and $R$ are given cube covers for the logic functions $f$, $d$, and $r$, respectively. Given a cube $c^i \in F$, the *variable blocking matrix* [11] is defined as

$$(B^i)_{jk} = \begin{cases} 1 & \text{if } c^i_k = 1 \text{ and } r^j_k = 0 \quad \text{for } r^j \in R, \\ 1 & \text{if } c^i_k = 0 \text{ and } r^j_k = 1 \quad \text{for } r^j \in R, \\ 0 & \text{otherwise.} \end{cases}$$

The *super-variable blocking matrix* $B$ is composed of all the rows of all the $B^i$. Note that $B$ has $|F| \times |R|$ rows.

A *row cover* of $B$ is a binary vector $v$, $v_k \in \{0, 1\}$, such that $Bv \geq l$, where $l$ is the vector of all 1's.

*Theorem 5* If $v$ is a row cover of $B$ and $v_k = 0$, then there exists a cover for $ff$ which is independent of the variable $x_k$.

*Proof* For any cube $c^i \in F$ define

$$\hat{c}^i = \begin{cases} c^i_p & p \neq k, \\ 2 & p = k. \end{cases}$$

Since $Bv \geq l$ and $v_k = 0$, then $\hat{c}^i$ is orthogonal to all $r^j$, $r^j \in R$. Hence $\hat{c}^i$ is an implicant of $ff$ containing $c^i$. Thus $\{\hat{c}^i\}$ is a cover independent of $v_k$. □

*Corollary 5* If $v$ is a minimum row cover of $B$ ($|v|$ is minimum), then a minimum variable support for $ff$ is the set of variables $\{x_i; v_i = 1\}$.

In the above formulation, the resulting prime irredundant cover for $ff$ may depend on both $x_i$ and $x'_i$, but this is counted only once in determining the minimal variable support. Since each unique literal appearing in a cover may represent a wire which must be routed to the function in an implementation, a minimum literal support may be desirable. This is solved in a similar way.

For each $c^i \in F$, define the *literal blocking matrix* $\hat{B}^i$ as

$$(\hat{B}^i)_{jk} = \begin{cases} 1 & \text{if } c^i_k = 1 \text{ and } r^j_k = 0, \quad r^j \in R, \\ 0 & \text{otherwise;} \end{cases}$$

$$(\hat{B}^i)_{j,n+k} = \begin{cases} 1 & \text{if } c^i_k = 0 \text{ and } r^j_k = 1, \quad r^j \in R, \\ 0 & \text{otherwise.} \end{cases}$$

Let $\hat{B}$ be the *super-literal blocking matrix* composed of all the rows of all the $\hat{B}^i$.

*Theorem 6* If $v$ is a row cover of $\hat{B}$ and $v_k = 0$, $k \leq n$, then there exists a cover of $ff$ independent of $x_k$; if $v_{n+k} = 0$, $k \leq n$, then there exists a cover of $ff$ independent of $x'_k$.

*Proof* The proof is similar to that of Theorem 5.

*Corollary 6* If $v$ is a minimum row cover of $\hat{B}$, then a minimum literal support for $ff$ is the set

$$\{x_k; v_k = 1\} \cup \{x'_k; v_{n+k} = 1\}.$$

In practice, these various row-covering problems can be solved heuristically with very good results. For one of the best such techniques, see [13].

*Boolean division for incompletely specified functions*

For an incompletely specified function $ff = \{f, d, r\}$, Boolean division is defined as

$$f = gh + e \bmod d \quad \text{and} \quad gh \neq 0 \bmod d,$$

meaning that equivalence is not required on the don't-care set $d$. We call $gh + e$ a *cover* of $ff$ if $f(v) = 1$ implies $g(v)h(v) + e(v) = 1$ and $r(v) = 1$ implies $g(v)h(v) + e(v) = 0$. $g$ is a *Boolean divisor* of $ff$ if there exist $h$, $e$ such that $gh \not\subseteq d$ and $gh + e$ is a cover of $ff$. We require that $g$, $h$, and $e$ be either completely specified logic functions or functions with $d$ as the don't-care set. Note that if the completely specified $\hat{f}$ is a cover of $ff$, then any factorization of $\hat{f}$ is also a cover of $ff$.

If there exists a cover of $ff$ that can be expressed as $gh \bmod d$, then we say that $g$ is a *Boolean factor* of $ff$, or that $g$ divides $ff$ evenly.

In applying Boolean division to factoring, the two problems to be solved are the following:

1. Given a logic function $g$ and an incompletely specified function $ff$, compute logic functions $h$ and $e$ such that $gh + e$ is a cover of $ff$ and such that $h$ and $e$ are minimal in some sense.
2. Given $ff$, find a function $g$ such that $gh + e$ is a cover of $ff$ and such that $g$, $h$, and $e$ are minimal in some sense.

The minimal conditions on $g$, $h$, $e$ make these problems interesting. It is not hard to find divisors of $ff$, but it is difficult to find divisors which lead to simple factorizations.

*Theorem 7*

Given $g$ and $ff = (f, d, r)$, define an extended don't-care set $\overline{d} = d + xg' + x'g$, where $x$ is a new variable. Suppose $\hat{f} = xh + e$ is a cover of $\overline{ff}$ (with the extended don't-care set $\overline{d}$), where $h$ and $e$ are independent of $x$ and $x'$. Then $gh + e$ is a cover of $ff$.

*Proof*

We denote the original function by $ff = (f, d, r)$, and the extended one by $\overline{ff} = (\overline{f}, \overline{d}, \bar{r})$, where $\overline{d} = d + xg' + x'g$, $\overline{f} = f\overline{d}'$, and $\bar{r} = r\overline{d}'$. Since $\overline{d}' = d'(xg + x'g')$, then $\overline{f} = fd'(xg + x'g') = f(xg + x'g')$. Similarly, $\bar{r} = r(xg + x'g')$. Suppose $f(v) = 1$; then $\overline{f}(v, x) = xg + x'g'$. Thus $\overline{f}(v, x) = 1$

if $x = g(v)$. Since $\hat{f}(v, x) = xh + e$ is a cover of $ff$, then also $\hat{f}(v, x) = 1$ if $x = g(v)$ and $f(v) = 1$. Thus $f(v) = 1$ implies $g(v)h(v) + e(v) = 1$. Next suppose $r(v) = 1$; then $\hat{r}(v, x) = xg + x'g'$ or $\hat{r}(v, x) = 1$ if $x = g(v)$. Again, since $\hat{f}$ is a cover of $\overline{ff}$, then $\hat{f}(v, x) = xh(v) + e(v) = 0$ if $x = g(v)$ and $r(v) = 1$. Hence $r(v) = 1$ implies that $g(v)h(v) + e(v) = 0$. □

Theorem 7 provides a solution to the first problem, in that $\hat{f}/x$ is the minimized cofactor $h$ given a candidate divisor $g$. This method of division has three basic steps:

1. Form a larger don't-care set $d + xg' + x'g$ expressing that $x \neq g$ is a don't-care condition, where $x$ is a new variable.
2. Minimize $f$ using this new don't-care set with any two-level logic minimizer that allows don't-cares. During the minimization, force $x'$ to be eliminated.
3. Return $(\hat{f}/x, e)$, where $e$, the remainder, is the terms of $\hat{f}$ which do not include $x$.

The procedure BOOL_DIV below uses the heuristic for obtaining the minimum literal support as discussed earlier. This is done by a call to procedure MINLIT passing as arguments the care onset $\overline{f}$ of $\overline{ff}$ and the care offset $\bar{r}$ of $\overline{ff}$. MINLIT solves the relevant row-covering problem and returns a slightly expanded cover, where the literals not in the minimum support have been eliminated.

```
BOOL_DIV(f, g, d):
n = number of variables
g' = COMPLEMENT(g)
        /* add the implied don't-care set to d using */
        /* the n + 1 variable as a new one */
        /* DC = d + g v'_{n+1} + g' v_{n+1} */
DC = APPEND((d, 2),APPEND((g, 0),(g', 1)))
        /* form the care offset */
r = COMPLEMENT(APPEND(f, DC))
        /* form the care onset */
f = COMPLEMENT(APPEND(r, DC))
        /* expand f to its minimum literal support */
        /* removing v'_{n+1} */
f = REMOVE(f, v'_{n+1})
f = MINLIT(f, r)
        /* expand f into primes and make an irredundant */
        /* cover */
f = EXPAND(f, r)
f = IRREDUNDANT(f, DC)
        /* compute f/v_{n+1} */
h = COFACTOR(f, v_{n+1}))
        /* REMAINDER returns terms not multiplying */
        /* v_{n+1} */
e = REMAINDER(f, v_{n+1})
return(h, e)
```

We assume here that $f$ is a given cover of $ff$, $d$ is the don't-care set for $ff$, and $g$ is a given candidate divisor. The first step forms a new don't-care set and the corresponding onset and offset, thus giving a new incompletely specified function $(f, DC, r)$ in the larger Boolean space. Next $v'_{n+1}$ is removed from the onset and the minimum literal heuristic is applied. At this point, any two-level logic minimizer could be applied, but here only the EXPAND and IRREDUNDANT steps are used. Finally, the coefficient of $v_{n+1}$ is returned as $h$, and the remaining terms as $e$.

## 3. Kernels

The notion of a kernel [7] of a logic expression was introduced to provide an efficient means for finding common subexpressions. In this paper, we see that kernels are a bridge between algebraic expressions and factored forms.

A *kernel* of an expression $f$ is defined by the following two rules:

1. A kernel $k$ of an expression $f$ is the quotient of $f$ and a cube $c$; $k = f/c$.
2. A kernel $k$ is "cube-free." ($k$ cannot be rewritten as $k = dg$, where $d$ is a nontrivial cube and $g$ is an expression.)

For example, suppose that

$$f = abc + abde.$$

Then $f/a = bc + bde$ is the quotient of $f$ and the cube $a$, but it is not cube-free since the cube $b$ is a factor of $f/a$,

$$f/a = b(c + de).$$

However, $f/ab = c + de$ is cube-free and hence a kernel.

Since no single cube is cube-free, a kernel must consist of at least two cubes. Also, since the universal cube, 1, is a cube and $f/1 = f$, then if $f$ is cube-free, $f$ is considered one of its own kernels.

Associated with each kernel is a cube, called its *co-kernel*, which is simply the cube divisor used in obtaining the kernel. Since the same kernel may be obtained in several ways by dividing $f$ by different cubes, the co-kernel of a kernel is not unique.

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.

For example,

$$x = (a(b + c) + d)(eg' + g(f + e')) + (b + c)(h + i)$$

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

$y = j(a(b + c) + d)(eg' + g(f + e')) + (b + c)(h + i)$

is a kernel of level 3 since it contains the level-2 kernel

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

with co-kernel $j$.

The following is an algorithm for computing all the kernels of an expression $f$:

```
KERNELS(f):
c = largest cube factor of f
K = KERNEL1(0, f/c)
if (f is cube-free){
        return f ∪ K
        }
return K

KERNEL1(j, g):
R = g
for (i = j + 1; i ≤ n; i++){
        if (l_i appears in only one term) continue
        c = largest cube dividing g/l_i evenly
        if (l_k not in c for all k ≤ i)
            R = R ∪ KERNEL1(i, g/(l_i ∩ c))
        }
return R
```

The algorithm works as follows. The argument $j$ in KERNEL1 is a pointer to the literals already factored out (all literals $\leq j$ have been processed). KERNEL1 is designed to find all kernels associated with any cube divisor *not* containing any of the literals $l_i$ for $i \leq j$. The recursive call to KERNEL1 restricts the function being processed to the subset of terms containing literal $i$. This computes the 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 it only processes unique co-kernels. Also, it gives the algorithm a very effective tree-trimming strategy for searching for kernels.

A few of the important properties of kernels are discussed below. We use the notation $K(f)$ to refer to the set of all kernels of $f$.

*Theorem 8 [7]* If two expressions $f$ and $g$ have the property that any $k_f \in K(f)$ and any $k_g \in K(g)$, which implies that $k_g$ and $k_f$ have at most one term in common, then $f$ and $g$ have no common algebraic divisors with more than one term.

This theorem is used for detecting whether 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) subkernels among kernels from different functions. If this set of subkernels is empty, then we need only look for divisors consisting of single cubes (which is an easier task). Thus we do not need to compute the set of all algebraic divisors for each expression to determine whether there are common nontrivial algebraic divisors. This leads to great run-time efficiency since the set of kernels is much smaller than the set of algebraic divisors, and, secondly, in the algorithm for computing kernels, the cube-free property of kernels provides a very effective means to trim the search tree for kernels.

The next theorem leads to very fast methods for factoring, for finding the prime factorization of a single expression, and for finding the greatest common divisor of several expressions. We use it in this paper for factoring.

*Theorem 9* If $p$ is a kernel of $f$ and is prime (cannot be factored algebraically), then $p$ is a kernel of exactly one of the prime factors of $f$.

*Proof* Suppose $f = \Pi f_i$. Let $d$ be a co-kernel of $p$. We can uniquely write $d = \Pi c^i$, where support $(c^i) \subseteq$ support $(f_i)$. Since $p = f/d = \Pi(f_i/c^i)$, then $p$ can be prime only if $c^i$ is a cube of $f_i$ for all but one $i$. Thus $p = f_i/c^i$ and since $p$ is cube-free, then $p$ is a kernel of $f_i$. □

*Corollary 9* Let $k$ be a level-0 kernel of $f$. Then $k$ is a kernel of exactly one of the algebraic prime factors of $f$.

*Proof* A level-0 kernel is prime. □

To help understand this result, consider the expression

$(a(b + c) + d)(eg' + g(f + e'))$.

This factors into two prime factors $a(b + c) + d$ and $eg' + g(f + e')$. Note that the level-0 kernel $b + c$ occurs in only one of the prime factors, while the level-1 kernel $(b + c)(f + e')$ spans several of the prime factors. We see later how this can be used to obtain a very fast effective method for factoring logic expressions.

Several algorithms (such as quick factor QF) will need to find just one level-0 kernel. This is obtained by a trivial modification to algorithm KERNEL1:

```
ONE_LEVEL_0_KERNEL(g):
if (|g| ≤ 1) return 0
if ((L = LITERAL_COUNT(g)) ≤ 1) return g
for (i = 1; i ≤ n; i++){
        if(L(i) ≤ 1) continue
        c = largest cube dividing g/l_i evenly
        return ONE_LEVEL_0_KERNEL(g/(l_i ∩ c))
        }
```

This algorithm terminates on finding the first level-0 kernel. Here LITERAL_COUNT returns a vector giving the

number of times each literal appears in a given expression. If all counts are $\leq 1$, then $g$ is a level-0 kernel which is returned and the algorithm terminates. Otherwise, the first literal with a count greater than one is chosen. It and all literal factors are divided out and the algorithm recurs until no literal appears more than once.

## 4. Factoring

In this section, we discuss three algebraic methods for factoring logic expressions and two Boolean methods.

• *Generic factoring*

Most algebraic or Boolean factoring methods can be described by the following recursive procedure:

```
GFACTOR(f):
If ( | f | ≤ 1) return f
k = CHOOSE_DIVISOR(f)
(h, r) = DIVIDE(f, k)
return (GFACTOR(k) GFACTOR(h) + GFACTOR(r))
```

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

Several variations of factoring can be obtained by choosing different algorithms for CHOOSE_DIVISOR and DIVIDE. The simplest method merely selects the best literal divisor (CHOOSE_LITERAL) and DIVIDE performs single literal division (ALG_DIV). This provides a very fast but suboptimal method for factorization. We call this LF, for "literal factorization."

As an example of literal factorization, consider the expression

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

CHOOSE_LITERAL might choose $k = a$ and ALG_DIV would return $h = c + d + e + g$ and $r = bc + bd + be + bf + ce + cf + df + dg$. Continuing in this way, LF might obtain (depending on which literals are chosen in sequence)

$$\mathrm{LF}(x) = a(c + d + e + g) + b(c + d + e + f) + c(e + f) + d(f + g).$$

We can do better by replacing CHOOSE_DIVISOR with CHOOSE_KERNEL, which computes all kernels of $f$ and returns the one with greatest "value." This is a slower but better-quality factorization and will be called XF. Continuing with the example,

$$\mathrm{XF}(x) = (c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce.$$

Although XF is slower because it takes longer to compute the kernels and to choose a best one, it always gives better factorizations than LF (in the example, 14 versus 16 literals).

Replacing CHOOSE_DIVISOR with CHOOSE_KERNEL and DIVIDE with BOOL_DIV leads to the procedure BF (for Boolean Factorization). This applied to the above example gives the same results as XF. However, for

$$x = ab' + ac' + ad' + a'b + bc' + bd' + a'c + b'c + cd' + a'd + b'd + c'd,$$

we obtain

$$\mathrm{BF}(x) = (a + b + c + d)(a' + b' + c' + d'),$$

whereas

$$\mathrm{LF}(x) = a'(b + c + d) + b'(a + c + d) + c'(a + b + d) + d'(a + b + c),$$

$$\mathrm{XF}(x) = a'(b + c + d) + (a + b)(c' + d') + c(b' + d') + c'd.$$

Note that the result of BF is not an algebraic factorization.

These three methods, LF, XF, BF, use a generic recursive factoring scheme GFACTOR and two different methods for selecting a divisor,

CHOOSE_LITERAL—pick the best literal (very fast),
CHOOSE_KERNEL—compute all kernels and choose best one (slow),

and the two variations of DIVIDE.

ALG_DIV—algebraic (weak) division (fast),
BOOL_DIV—Boolean (strong) division (slow).

These provide a spectrum of speed and quality of results for factoring.

• *Quick factoring*

A variation on the generic factoring schema above is

```
GFACTOR2 (f):
If (| f | ≤ 1) return f
k = CHOOSE_DIVISOR(f)
(h, r) = DIVIDE(f, k)
If (h is not a cube){h = CUBE_FREE(h)}
      else {h = ONE_LITERAL_OF(h)}
(k, r) = DIVIDE(f, h)
return GFACTOR2(k) GFACTOR2(h) + GFACTOR2(r)
```

The heuristic used here is that having chosen the divisor $k$, we obtain $h$ and are in the process of forming the partial factorization $f = kh + r$. Given that $h$ will be used as a factor, we might as well collect everything that can be multiplied by $h$. If $h$ is not prime, the same could be said about the factors of $h$. However, we cannot afford to factor

$h$, except to eliminate any literal factors of $h$ (CUBE_FREE). Then we perform the second DIVIDE to obtain a new $k$, which must include at least the old $k$. This scheme is designed for situations where the best divisor $k$ was not chosen, perhaps because of run-time considerations.

A fast variation of factoring is obtained then by replacing CHOOSE_DIVISOR with ONE_LEVEL_0_KERNEL, which determines a single level-0 kernel, and DIVIDE with ALG_DIV. This leads to the procedure QF (for quick factor).

In the example

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg,$$

QF and XF give different but similar-quality factorings,

$$\mathrm{XF}(x) = (c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce,$$

$$\mathrm{QF}(x) = g(a + d) + (a + b)(c + d + e) + c(e + f) + f(b + d),$$

but QF is much faster because it only needs to determine one level-0 kernel for each factor. However, for

$$x = abeg' + abfg + abe'g + aceg' + acfg + ace'g + deg' + dfg + de'g + bh + bi + ch + ci,$$

$$\mathrm{QF}(x) = (a(g(e' + f) + eg') + i + h)(b + c) + d(g(e' + f) + eg'),$$

but XF obtains the better result (13 literals versus 16 literals),

$$\mathrm{XF}(x) = (a(b + c) + d)(eg' + g(f + e')) + (b + c)(h + i),$$

by working harder to find the best kernel to be used at the top level of the recursion.

In more recent implementations of logic synthesis algorithms [10], QF has become an important tool because it is so fast and effective. QF is used continuously during synthesis to estimate area and delay.

• *Factoring using duality*

The final method of factoring simply recognizes that a factored form can be obtained by factoring the complement of a function by any of the methods discussed, and then using duality or DeMorgan's law to obtain a factoring of the function. For example, suppose we have factored

$$F' = \mathrm{FACTOR}(x') = (ab' + a'b)(cd(a' + e) + c'(b'e + f)) + bce.$$

By using DeMorgan's law, a factored form for $x$ is obtained as

$$\mathrm{DUAL}(F') = ((a' + b)(a + b') + (c' + d' + ae')(c + (b + e')f')(b' + c' + e').$$

The procedure, outlined below, is denoted by DF for DeMorgan Factoring:

DF($f$):
$f'$ = COMPLEMENT($f$)
$F'$ = FACTOR($f'$)(using LF, QF, XF, or BF)
return DUAL($F'$)

## 5. Optimal algebraic factoring and rectangle-covering problems

Rectangles and rectangle coverings of Boolean matrices have several applications in logic synthesis [14]. We discuss one of these here, namely, optimal algebraic factoring.

• *Rectangle coverings*

A *rectangle* of a Boolean matrix $B$, $B_{ij} \in \{0, 1\}$ is a subset of rows and columns $(R, C)$ such that $i \in R, j \in C$ implies $B_{ij} = 1$.

A *prime rectangle* $P = (R, C)$ of $B$ is one not strictly contained in another rectangle of $B$.

We say that $B$ is *covered* by the set of rectangles $\{(R^k, C^k)\}$ if

$$B_{ij} = 1 \quad \text{implies} \quad i \in R^k, j \in C^k \quad \text{for some rectangle } k.$$

A covering need not be disjoint, so that a 1 in $B$ may be covered by several rectangles. In general we are interested in the weighted covering problem where we assign each rectangle $(R, C)$ a weight defined by a weight function, $w(R, C)$. The weight of a cover $\{(R^k, C^k)\}$ is defined as the sum

$$\sum_k w(R^k, C^k).$$

The rectangle-covering problem is to find a cover of $B$ with minimum weight.

One straightforward way to approach the rectangle-covering problem is to formulate it as a row-covering problem. We first find all prime rectangles of $B$. (Because kernels and prime rectangles are related [14], we can use a procedure like KERNEL for finding all prime rectangles of $B$.) Now construct another Boolean matrix $M$ which has a row for each 1 in $B$ and a column for each prime rectangle. $M_{ij} = 1$ if the prime $P^j$ covers the 1 in $B$ associated with row $i$ of $M$. Otherwise $M_{ij} = 0$. Since we want to find a covering of least weight, each column of $M$ is given a weight $W_j$ equal to the weight of the associated prime rectangle $P^j$; $W_j = w(P^j)$. By solving

Minimize: $W^T x$
such that: $Mx \geq 1$,

we obtain a minimum prime rectangle covering of $B$ of least weight.

• *Heuristic coverings*

Because of the obvious complexity of finding an optimum rectangle cover, and because an optimum cover may include

nonprime rectangles, we resort to heuristics. The following is a greedy procedure for determining a good rectangle cover. It proceeds in two phases. In the first, we attempt to find an optimal prime rectangle cover. In the second phase, we attempt to decrease each of these rectangles in an optimal way to obtain a cover with smaller rectangles. If the weight of a rectangle decreases with smaller size, then the reduced covering of nonprime rectangles is better. During the iteration, as primes are chosen greedily, an increasing number of 1's in $B$ are marked as "don't-care" points, indicating that the 1 is covered but it is permissible for another rectangle to cover the same point. However, the weight function should be defined such that there is no advantage or disadvantage in covering a don't-care point:

```
RECTANGLE_COV(B):
P = 0
while (there are "cares" in B){
    P_i = prime rectangle of greatest value.
    Mark as don't-care any 1 in B associated with P_i
    P = P ∪ P_i,
    }
P = IRREDUNDANT(P, B)
return P = REDUCE_R(B, P)
```

Similarly to the ESPRESSO-II procedure for logic minimization, IRREDUNDANT selects a subset of $P$ which still covers $B$. REDUCE_R maximally reduces each of the rectangles in the cover by deleting rows or columns of the prime rectangle to decrease its weight. This can be done if the other rectangles jointly cover a row or column of a rectangle:

```
REDUCE_R(B, P):
for (i = |P| − 1 to 1){
    P̂ = P − {P_i}.
    P_i = rectangle of least weight covering P_i − P̂.
    }
return {P_i}
```

Note that in RECTANGLE_COV we have stated that 1's, which are "associated" with 1's of $B$ covered, are marked as don't-care. Later we will have a general notion of a 1 of $M$ being associated with a rectangle. In some applications this means simply that it is covered by the rectangle, but in other applications it is important to be able to have 1's indirectly associated with each other. The procedure RECTANGLE_COV is analogous to the EXPAND, IRREDUNDANT, and REDUCE sequence of ESPRESSO-II [11], with REDUCE being necessary if smaller rectangles have less weight. This operation can be iterated, as done in ESPRESSO-II, by restricting the first part to select and expand to prime each rectangle of a given rectangle covering. These are made irredundant and reduced, with the reduced rectangles becoming the input to the first part for re-expansion. Iteration would continue until no decrease in weight is obtained.

We will see how the RECTANGLE_COV procedure can be applied to optimal algebraic factoring. This type of factorization can be extended to multiple functions [14].

• *Optimal algebraic factorization using rectangle covering and overlap*

An optimum algebraic factorization is one with the least number of literals. In general, it is obtained by allowing one or more terms to be repeated; i.e., a term may be produced by more than one product. As an example,

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + de + df + dg$$

can be factored optimally as

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

Note that in expanding this, the term $de$ is produced twice. However, each of the products combines expressions with disjoint variables and hence they are algebraic multiplications.

The idea of allowing repeated terms (*overlap*) is a natural consequence of rectangle covers with overlapping rectangles.

An optimum factorization is a sum of products of sums of products, etc. At the top level, each product represents a subset of the terms of $f$ which can be algebraically factored. One way to obtain an optimum factoring is to choose these subsets correctly and then to recursively optimally factor each of their prime factors.

We say that a subset of terms of $f$ is a *product* of $f$ if it can be factored algebraically into two or more prime factors. A *maximal product* of $f$ is a product not contained in another product of $f$. For example, for

$$x = ac + ad + bc + bd + ec + ed + ef,$$

the subset $ac + bd + ad + bc$ is a product of $x$ but it is not maximal since

$$(a + b)(c + d) \subseteq (a + b + e)(c + d) \subseteq x.$$

However, both $e(c + d + f)$ and $(a + b + e)(c + d)$ are maximal products of $x$.

An optimum algebraic factorization of $f$ is a sum of products, but in general each product need not be a maximal product. However, each product is contained in some maximal product. Thus an appropriate set of maximal products could in theory be reduced in an optimal way, thereby obtaining an optimal factorization. We relate these ideas to a rectangle-covering problem using Theorem 10 below, which relates maximal products to kernels. In turn, kernels are related to rectangles of an appropriate Boolean matrix.

*Definition* A *subkernel* of $f$ is any expression with at least two terms obtained as the maximal set of terms common to each kernel of any set of kernels of $f$.

For notation, if $\{e_i\}$ is a set of expressions, then $S(\{e_i\})$ is the maximal set of terms common to all the expressions:

$$S(\{e_i\}) = \{t^j;\ t^j \in e_i \text{ for all } i\}.$$

Thus a subkernel is $S(\{k_i\})$ if it has at least two terms.

*Theorem 10* Let $s$ be a maximal product of $f$. Then $s$ can be written as

$$s = c\prod_i q_i,$$

where $c$ is a cube and each $q_i$ is a cube-free subkernel.

*Proof* Since $s$ is a product, it can be written as

$$s = cgh,$$

where $c$ is a cube and $g$ and $h$ are either cube-free expressions or 1. Let $h = \{h^i\}$. If $cg \neq S(\{f/h^i\})$, since $cg \subseteq S(\{f/h^i\})$, then $S(\{f/h^i\})h$ is a product of $f$ containing $s = cgh$, contradicting that $s$ is maximal. Thus $cg = S(\{f/h^i\})$ and

$$s = cS(\{f/ch^i\})h.$$

Similarly for $h$, so

$$s = cS(\{f/ch^i\})S(\{f/cg^i\}).$$

Hence, $s$ can be written as a cube times the product of cube-free subkernels of $f$. □

Recall the KERNEL procedure of Section 3. During this procedure, we record a kernel defined as a cube-free quotient $f/c$ for some cube $c$, the co-kernel. Each of the co-kernels is unique by the nature of the KERNEL procedure.

We build a Boolean matrix $M$ for an algebraic expression $f$ as follows. Each row corresponds to a unique co-kernel, and each column corresponds to a cube, $d^j$, of any kernel found. The $d^j$ associated with the columns are made unique. $M_{ij} = 1$ if cube $d^j$ is a term of kernel $k^i$; otherwise $M_{ij} = 0$. Since the cubes $d^j$ are unique, then a prime rectangle $(R, C)$ of $M$ corresponds to the subkernel of the set of kernels associated with the rows $R$.

*Theorem 11* A rectangle $(R, C)$ of $M$ is prime if and only if $q = \{d^j;\ j \in C\}$ is a subkernel of $f$.

*Proof* Assume $(R, C)$ is prime. Then $R$ and $C$ are maximal and $S(\{k^i;\ i \in R\}) = \{d^j;\ j \in C\}$. Conversely, if $q = S(\{k^i;\ i \in R\})$, then $C = \{j;\ d^j \in q\}$ is maximal, hence $(R, C)$ is prime. □

As an example, with

$$x = ade + af + bcde + bcf + g,$$

then $M$ is

| | $a$ | $bc$ | $de$ | $f$ |
|---|---|---|---|---|
| $a$ | 0 | 0 | 1 | 1 |
| $bc$ | 0 | 0 | 1 | 1 |
| $de$ | 1 | 1 | 0 | 0 |
| $f$ | 1 | 1 | 0 | 0 |

Here, the second kernel (row 2) was obtained by dividing $x$ by $bc$ (the second cube on the left). This kernel is $de + f$.

For optimal factoring, we order the terms of $f = \{t^l\}$ and define a mapping

$$m_{ij} = \begin{cases} k & \text{if } c^i d^j = t^k, \\ 0 & \text{otherwise.} \end{cases}$$

Two 1's in $M$ are *associated* if $m_{ij} = m_{kl}$.

In the example, if the terms of $x$ are ordered

| *term number* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| *terms* | *ade* | *af* | *bcde* | *bcf* | *g* |

then the terms associated with the Boolean matrix are

| | $a$ | $bc$ | $de$ | $f$ |
|---|---|---|---|---|
| $a$ | 0 | 0 | 1 | 2 |
| $bc$ | 0 | 0 | 3 | 4 |
| $de$ | 1 | 3 | 0 | 0 |
| $f$ | 2 | 4 | 0 | 0 |

i.e., $m_{13} = 1$ since $(a)(de) = ade = t^1$, $m_{14} = 3$ since $(de)(bc) = bcde = t^3$, etc., and (4, 1) is associated with (1, 4).

A rectangle $(R, C)$ of $M$ is associated with (or covers) a subset of terms of $f$:

$$T(R, C) = \{t^l : t^l = m_{ij},\ i \in R, j \in C\}.$$

Note that no term is covered more than once by a single rectangle; i.e., if $m_{ij} = m_{kl}$ and $i, k \in R, j, k \in C$ for some rectangle $(R, C)$, then $i = k, j = l$. An entry $M_{ij} = 1$ in $M$ is said to be *associated* with a rectangle $(R, C)$ if $m_{ij} \in T(R, C)$. Note that this is different from an entry being covered by a rectangle. For example, entry $M_{32}$ is associated with rectangle $(\{1, 2\}, \{3, 4\})$ since

$$m_{32} = 3 \in T(\{1, 2\}, \{3, 4\}) = \{1, 2, 3, 4\}.$$

In applying the procedure RECTANGLE_COV to $M$ with $m$ used to associate entries in $M$, it is necessary to give a weight to a rectangle $(R, C)$. In the greedy procedure in the first part, we want to choose a prime rectangle $P_i$ with two objectives: first, to cover as many care points as possible with the next choice of $P_i$; second, to choose a rectangle with least weight.

This motivates the following weight function. First, let the "cost" of a cube, $c(c^i)$, be equal to the number of literals in

it. Suppose $(R, C)$ is a prime rectangle. Let $\hat{R} \subseteq R$ be the set of rows $R$ which contain at least one care point, and similarly for $\hat{C} \subseteq C$. The cost of $(R, C)$ is then defined as

$$\hat{c}(R, C) = \sum_{i \in \hat{R}} c(c^i) + \sum_{j \in \hat{C}} c(d^j).$$

Here, $\{c^i\}$ and $\{d^j\}$ are the cubes of the co-kernels and kernels, respectively. We sum over only those rows and columns which contain "care" points, since we do not want to penalize prime rectangles for covering don't-care rows or columns.

In the example, if all points are care points, the cost of $(\{1, 2\}, \{3, 4\})$ is the number of literals in the factorization represented by the rectangle, $(a + bc)(de + f) = 6$. However, if points (1, 4) and (2, 4) are don't-care, then the cost is 5, since $(a + bc)(de + f)$ can be implemented as $(a + bc)de$.

Finally, the weight of $(R, C)$ is defined as

$$w(R, C) = \frac{\hat{c}(R, C)}{\text{number of care points covered by } (R, C)},$$

i.e., $w(R,C)$ is the average cost of each care term covered by $(R, C)$.

With these notions of the weight of a rectangle, and of which of the 1's in $M$ are associated with a rectangle, we can apply RECTANGLE_COV to obtain a near-optimal factorization of $f$. This is discussed later in this section after we explore how to find a rectangle with least weight.

• *Choosing a rectangle of least weight*

A heuristic for finding the rectangle of small weight is composed of three algorithms: GREEDR, GREEDC, and PING_PONG. Each receives as input a matrix $B$, $B_{ij} \in \{0, 1, 2\}$, where 2 indicates a don't-care point. A "cost" for each row and column is also given by the vectors, $w^R$, $w^C$. GREEDR and GREEDC are also given an index, $k$, of a row or column, respectively:

GREEDR($B$, $k$, $w^R$, $w^C$):
$R = \{k\}$
$C = \{j; B_{kj} \neq 0\}$
$P = (R, C)$
while ($|C| > 1$){
  $l = \underset{l \notin R}{\text{argmin}} \dfrac{w_l^R}{\sum_{B_{lj}=1, j \in C} (w_j^C)^{-1}}$
  $R = R \cup \{l\}$
  $C = \{j; B_{ij} \neq 0$, for all $i \in R\}$
  if ($w(R, C)$ is best yet) $P = (R, C)$
  }
return $P$

In the above procedure, we start the rectangle with the given row $k$, and the columns are simply the nonzero (1's or 2's) columns of row $k$. Next we choose a row $l$ not in $R$ which minimizes the quotient shown. The sum in the denominator is maximized by choosing a row with many care columns ($B_{lj} = 1$) in common with $C$, each of small weight. Then $l$ is added to $R$, and $C$ is updated in a way ($B_{ij} \neq 0$) which allows don't-care points in the rectangle. If the present rectangle is the best seen so far, it is recorded. This is iterated until $C$ has only one column. During GREEDR, a sequence of prime rectangles is observed starting with one of maximal width and ending with one of maximal height.

The procedure GREEDC is the same as GREEDR except that the roles of rows and columns are interchanged.

The procedure PING_PONG below alternates between GREEDR and GREEDC until a rectangle is repeated:

PING_PONG($B$, $w^R$, $w^C$):
$k = \underset{k}{\text{argmin}} (w_k^R) \Big/ \sum_{B_{kj}=1} (w_j^C)^{-1}$
$P = P_1 = 0$
while ($P_1 \notin P$){
  $P_1 = (R, C) = \text{GREEDR}(B, k, w^R, w^C)$
  $P = P \cup P_1$
  $j = \underset{j \in C}{\text{argmin}} (w_j^C) \Big/ \sum_{B_{ij}=1} (w_i^R)^{-1}$
  $P_1 = (R, C) = \text{GREEDC}(B, j, w^R, w^C)$
  $P = P \cup P_1$
  $k = \underset{k \in R}{\text{argmin}} (w_k^R) \Big/ \sum_{B_{kj}=1} (w_j^C)^{-1}$
  }
return $P_1$.

• *Optimal factoring, OF*

Finally, a near-optimal factoring procedure is the following:

OF($f$):
Build Boolean matrix $M$ and term map $m$ using kerneling procedure on $f$.
Compute $w^R$, $w^C$ from co-kernels $\{c^i\}$ and kernel cubes $\{d^i\}$.
$B = M$
$P = 0$
while ($B_{ij} = 1$ for some $ij$){
  $P_1 = (R, C) = \text{PING\_PONG}(B, w^R, w^C)$
  $P_1 = \text{EXTEND}(P_1, B, w^R, w^C)$
  $P = P \cup P_1$
  for (all $ij$){
    if ($i, j$ associated with $P_1$)$B_{ij} = 2$
    }
  }
$P = \text{IRREDUNDANT}(P, M, m)$
$P = \text{REDUCE\_R}(M, m, P)$
for ($i = 1$; $i \leq |P|$){
  $f_i = \{c^j; j \in R^i\}$
  $g_i = \{d^j; j \in C^i\}$
  }
return $(\sum \text{OF}(f_i) \times \text{OF}(g_i))$

OF uses PING_PONG to determine a near-optimal selection of the first level of factorization, i.e., the first set of products in the sum of products representation. After each rectangle, $P_1$, has been selected by PING_PONG, we recognize that it may not be the best rectangle. Hence, similar to what is done in GFACTOR2, this rectangle is used to try to find a better one by the procedure EXTEND. This examines the expression determined by the columns of $P_1$ to see if it is a product. If it is a product, then each factor will be represented by a subset of columns since the kerneling process used to form the Boolean matrix, $M$, found all kernels. The associated rectangle of each factor is examined to see if it is better than $P_1$, and if so, it replaces $P_1$.

An irredundant subset of $P$ is selected (IRREDUNDANT) and each of these is reduced (REDUCE_R). Each of the factors associated with $R^i$ and $C^i$ is recursively optimally factored by applying OF to the expressions $f_i$ and $g_i$ formed by the co-kernels of the rows and cube kernels of the columns of the reduced rectangles.

## Acknowledgments

Many of the ideas presented here were conceived and implemented jointly with Curt McMullen. A book on logic synthesis is planned, and many ideas are also due in part to intense interactions with my co-authors Gary Hachtel, Curt McMullen, Rick Rudell, and Alberto Sangiovanni-Vincentelli. Also, Albert Wang has contributed through the implementation of MIS and through close collaboration this year at Berkeley. The logic "team" in Course 290 at Berkeley, Alessandro Cagnola, Ewald Detjens, Suresh Krishna, Tony Ma, Patrick McGeer, Li-Fan Pei, Nathan Phillips, Russell Segal, Nicholas Weiner, Robert Yung, and Tiziano Villa, provided inspiration to work harder on these ideas. Motivation for developing factoring was greatly aided by experience with its use in the Yorktown Silicon Compiler [15] and in the design of a 32-bit microprocessor [16], where N. Brenner, C. L. Chen, G. DeMicheli, J. Jess, M. Mack, R. Otten, L. van Ginneken, J. Katzenelson, and Y. Yamour have contributed. Earlier interactions with various people in IBM, such as G. Marinen, G. Machol, W. Griffin, R. Kilmoyer, J. Davis, and N. Portuoando, were also very helpful.

## References

1. R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, 1986.
2. K. A. Bartlett and G. D. Hachtel, "Library Specific Optimization of Multi-Level Combinational Logic," *ICCAD85*, October 1985.
3. R. K. Brayton and C. T. McMullen, "Synthesis and Optimization of Multi-Stage Logic," *ICCD84*, October 1984.
4. A. de Geus and W. Cohen, "Optimization of Combinational Logic Using a Rule-Based Expert System," *J. Design & Test*, to appear.
5. D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel, "SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic," *Design Automation Conference*, June 1986.
6. J. F. M. Theeuwen and P. T. H. M. van Paassen, "Automatic Generation of Boolean Expressions in NMOS Technology," *ICCAD85*, November 1985.
7. R. K. Brayton and C. T. McMullen, "The Decomposition and Factorization of Boolean Expressions," *ISCAS Proceedings*, April 1982.
8. R. K. Brayton, J. Cohen, G. D. Hachtel, B. Trager, and D. Y. Y. Yun, "Fast Recursive Boolean Function Manipulation," *ISCAS Proceedings*, April 1982.
9. R. K. Brayton, C. L. Chen, C. T. McMullen, R. H. J. M. Otten, and Y. Yamour, "Automated Implementation of Switching Functions as Dynamic CMOS Arrays," *CICC84*, May 1984.
10. R. K. Brayton, A. Cagnola, E. Detjens, S. Krishna, A. Ma, P. McGeer, Li-Fan Pei, N. Phillips, R. Rudell, R. Segal, A. Wang, N. Weiner, R. Yung, T. Villa, A. R. Newton, A. Sangiovanni-Vincentelli, and C. H. Sequin, "Multiple-Level Logic Optimization System," *ICCAD86*, November 1986.
11. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
12. M. R. Dagenais, V. K. Agarwal, and N. C. Rumin, "The McBoole Logic Minimizer," draft, 1984.
13. R. L. Rudell, "Exact Minimization of Multiple-Valued Functions for PLA Optimization," *ICCAD86*, November 1986.
14. R. K. Brayton, "Algorithms for Multi-Level Logic Synthesis and Optimization," *NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design*, L'Aquila, Italy, July 1986.
15. R. K. Brayton, N. L. Brenner, C. L. Chen, G. DeMicheli, C. T. McMullen, and R. H. J. M. Otten, "The YORKTOWN Silicon Compiler," *ISCAS'85*, June 1985.
16. R. K. Brayton, N. L. Brenner, C. L. Chen, G. DeMicheli, J. Katzenelson, C. T. McMullen, R. H. J. M. Otten, and R. L. Rudell, "A Microprocessor Design Using the Yorktown Silicon Compiler," *ICCD85*, October 1985.

*Received October 2, 1986; accepted for publication January 16, 1987*

**Robert K. Brayton** *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Brayton received the B.S. degree from Iowa State University, Ames, in 1956 and the Ph.D. in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1961. While at M.I.T., he was a research assistant in artificial intelligence and developed the first LISP compiler. He has been at the IBM Thomas J. Watson Research Center since 1961, spending the years 1966 at M.I.T., 1976 at Imperial College, London, and 1985–86 at Berkeley as a visiting professor. He is currently the manager of the mathematical algorithms groups consisting of logic design, mathematical programming, general mathematical analysis, and parallel computing. Dr. Brayton is a Fellow of the American Association for the Advancement of Science and of the Institute of Electrical and Electronics Engineers, and winner of the IEEE Best Paper Award (Circuit and Systems Group); he has received two IBM Outstanding Innovation Awards and an IBM Patent Award. From 1970–72, he was a NSF Chautauqua Lecturer on Mathematical Models and Computing. He has been a member of the NSF Advisory Panel in Mathematics, the IEEE Circuits and Systems ADCOM, and subcommittees on nonlinear networks, large-scale systems, and computer-aided design. Dr. Brayton is the author of two books, *Computer Aided Design: Sensitivity and Optimization* with R. Spence, and *Logic Minimization Algorithms for VLSI Synthesis* with Hachtel, McMullen, and Sangiovanni-Vincentilli. His fields of interest have been nonlinear networks, sparse matrices, numerical analysis, stability theory, computer-aided design, character recognition, and optimization. His most recent research has been in the construction and design of a silicon compiler and in related problems of logic synthesis and minimization, state assignment, and high-level design languages.