VLSI DESIGN
1995, Vol. 3, Nos. 3-4, pp. 267-287
Reprints available directly from the publisher
Photocopying permitted by license only

Division-Based Versus General Decomposition-Based Multiple-Level Logic Synthesis

FRANK VOLF, LECH JÓŹWIAK and MARIO STEVENS
Eindhoven University of Technology, Faculty of Electrical Engineering, P.O. Box 513, 5600 MB Eindhoven, the Netherlands

During the last decade, many different approaches have been proposed to solve the multiple-level synthesis problem with different minimum functionally complete systems of primitive logic blocks. The most popular of them is the division-based approach. However, modern microelectronic technology provides a large variety of building blocks which considerably differ from those typically considered. The traditional methods are therefore not suitable for synthesis with many modern building blocks. Furthermore, they often fail to find global optima for complex designs and leave unconsidered some important design aspects. Some of their weaknesses can be eliminated without leaving the paradigm they are based on, other ones are more fundamental. A paradigm which enables efficient exploitation of the opportunities created by the microelectronic technology is the general decomposition paradigm. The aim of this paper is to analyze and compare the general decomposition approach and the division-based approach. The most important advantages of the general decomposition approach are its generality (any network of any building blocks can be considered) and totality (all important design aspects can be considered) as well as handling the incompletely specified functions in a natural way. In many cases, the general decomposition approach gives much better results than the traditional approaches.

Key Words: *Logic Synthesis, Decomposition, Digital Circuits, FPGA Synthesis, ASIC Design, VLSI Design*

1. INTRODUCTION

The term **logic synthesis** refers to all transformations in the design of digital hardware in which binary data are involved. In this paper, we will only consider a subset of logic synthesis methods, namely methods for the transformation of a multiple-output binary function into a (near-)optimal multiple-level network of primitive logic blocks. The term **primitive logic block** refers to any binary function that can be mapped one-to-one onto a primitive hardware building block in a certain technology. A **primitive hardware building block** is the smallest hardware unit considered that is used to implement binary functions in a certain technology. Examples of primitive hardware building blocks are the gates in the library of standard-cell implementations or configurable logic blocks (clbs) for Xilinx FPGA [54] implementations.

Up to 1980, very special cases of the multiple-level logic synthesis received the most attention namely, the transformation of a binary function into an optimal two-level network (e.g. AND-OR-NOT, OR-AND-NOT, NAND-NAND, NOR-NOR and AND-EXOR implementations), and the transformation into multiple-level EXOR, AND-EXOR, AND-OR-NOT or MUX networks. This interest resulted from the fact that these networks could be easily modelled, minimized and mapped one-to-one on networks of typical primitive hardware building blocks provided by the electronic technologies of that time, for example, on those as in TTL and ECL technologies and on PLAs or PALs.

Multiple-level networks often allow a more compact implementation of combinational logic in comparison with two-level networks. They enable the separate implementation of common sub-expressions and sharing them among multiple functions or sub-functions. However, many functions do not result in compact AND-EXOR, AND-OR-NOT or MUX networks. On the other hand, modern microelectronic technology provides us with a huge number of various primitive hardware building blocks which can be used for obtaining more compact networks. Exploitation of these abilities requires new appropriate multiple-level synthesis methods. The introduction of a new generation of FPGAs [49] has recently generated very strong stimulus for research in multiple-level logic synthesis. The internal structure of the FPGAs is in fact a programmable multiple-level network and therefore, these devices require the use of multiple-level

logic synthesis techniques in order to exploit their abilities. Unfortunately, the synthesis of general multiple-level networks is much more complicated than the synthesis of two-level logic. The main reason for this is the difficulty in defining the nature of the "optimal solution" in the multiple-level synthesis problem. For example, in the case of two-level AND-OR-NOT logic, the "optimal solution" is the solution with the minimal number of product terms, which is a relatively good measure for the complexity of the implemented network. In multiple-level logic, the structure of the logic is less uniform and can be considerably more complex than two-level logic. Also, the design decisions have a much more substantial impact on the many factors that decide the total quality of a multiple-level logic network: area, speed, power dissipation, testability etc. Furthermore, these factors are no longer simple functions of the implementation structure as it was the case for two-level logic.

During the last decade many different approaches have been proposed to solve the multiple-level logic synthesis problem. The most important of them are the following:

- algebraic and Boolean division techniques [2][7][8][10][42][45],
- multiple-level BDD and other decision graph approaches [6][13][14][37],
- algorithms based on the minimisation of the communication complexity between blocks [23],
- multiple-valued logic based approaches [36][43][52],
- methods based on spectral analysis techniques ([17] contains an overview),
- methods based on the iteration of gate transforms and gate reductions, the so-called transduction methods [40].

In recent years a number of papers have been published which implicitly or explicitly use a new concept of general structural decomposition as a synthesis paradigm (e.g. [15][26][27][30][32]). The distinctive feature of these methods is that they are all special cases of a general full-decomposition as presented in section 3. In the general full-decomposition approach, an incompletely specified multiple-output binary function is decomposed into a network of communicating subfunctions (logic blocks) in such a way that this network realizes the specified behaviour, satisfies specified constraints and optimizes given objectives. Decomposition decisions are based on analysis of the structure of the information streams in the function and the relations between this structure and the specified con-straints and objectives. The constraints imposed by hardware building blocks and their possible interconnections are innately taken into account. This approach has a number of interesting properties, including the following:

- In many multiple-level synthesis approaches Boolean expressions are used to describe functions. Very often these approaches use only a limited set (minimum functionally complete set) of Boolean operators (e.g. AND-OR-NOT) and not the full set of operators implemented by a certain library of hardware building blocks. To implement the minimised expression, a transformation step called **technology mapping** must be performed in order to transform the expression into a network of hardware building blocks. If the repertoire of primitive logic blocks offered by a certain technology library differs substantially from the set of Boolean operators used during synthesis, the work completed during synthesis is almost futile, because the real synthesis must be performed during the technology mapping. The synthesis methods based on general decomposition integrate the technology mapping phase into the synthesis: a network of logic blocks, that can be mapped one-to-one onto a network of primitive hardware building blocks, is constructed.
- The internal structure of Xilinx FPGAs and similar fine granularity FPGAs is in fact a programmable multiple level logic block structure which can be innately modelled using the theory in Section 3. Therefore methods for the synthesis of these types of FPGAs can be relatively easily constructed using this theory.

This paper aims to present a comparative analysis of the general decomposition-based and division-based multiple-level logic synthesis approaches. We will investigate the properties of the decomposition-based logic synthesis methods by introducing the general full-decomposition concept, presenting and discussing the existing decomposition methods and comparing the decomposition methods with the classical multiple-level synthesis methods based on the division of Boolean expressions. We have chosen the division-based algorithms as our reference, because they are by far the most popular ones (as measured by the number of publications on this subject).

The remainder part of this paper has been organised as follows: Sections 2 and 3 contain introductions to the theory and the most important results obtained in division-based and decomposition-based logic synthesis, respectively. In Section 4, a comparison between these two classes is presented. Some concluding remarks can be found in Section 5.

2. DIVISION-BASED LOGIC SYNTHESIS

The fundamentals for division-based multiple-level logic synthesis were introduced by Robert K. Brayton in 1982 [7][8][10]. In this section, we will review the most important aspects of Brayton's original method and

discuss its advantages, disadvantages and a few extensions to this method.

2.1 Basic Theory

The division-based multiple-level logic synthesis is based on manipulation of Boolean expressions. The theoretical framework for these expressions consists of binary Boolean algebras. These binary Boolean algebras are well known [5][21][47] and therefore, we will not repeat them here.

A **Boolean variable** is a single coordinate in a Boolean space. A **literal** is a Boolean variable or its complement. A **cube** c is a set of literals such that $x \in c \Rightarrow \bar{x} \notin c$, for example {a, b} represents the cube ab; {a, $\bar{a}$} is not a cube. Two trivial cubes 0 and 1 exists; they are defined as the Boolean functions 0 and 1 respectively. A **Boolean expression** is a set of cubes. We will not write expressions as set of cubes, but we will use the well known sum-of-product term representations. The translation between these two notations is straightforward. A Boolean expression is called **non-redundant** if no cube of the expression contains another cube properly. For example the expression a + ab is redundant because $\{a\} \subseteq \{a,b\}$. The expression $a+\bar{a}b$ is non-redundant. The **support sup(f)** of an expression f is $sup(f) = \{x \mid \exists c \in f : x \in c \vee \bar{x} \in c\}$. Less formally, the support is the set of Boolean variables which appear either complemented or uncomplemented in the expression f. Two functions f and g are said to have a **disjoint support** if and only if $sup(f) \cap sup(g) = \emptyset$.

The expressions describe the structure of a multiple-output Boolean function. Parentheses are used to specify the multiple-level character of the expressions; such a parenthesised expression is called a **factored form** and the process of obtaining a factored form of a Boolean function is called **factorisation.** An **incompletely specified single-output function** $\mathcal{F}$ is represented by a triple of completely specified single-output functions: $\mathcal{F}$ (f,d,r), where f represents the on-set, i.e. all input patterns for which $\mathcal{F}$ evaluates to 1. Similarly, d and r are representations of the don't care-set and the off-set, respectively. The functions f, d and r should be multiple exclusive and they should form together the complete Boolean space (i.e. all vectors from the Boolean input space should belong to precisely one of these functions). The **product of two expressions** f and g (denoted $f \cdot g$) is defined as $f \cdot g = \{c_i \cup d_j \mid c_i \in f \wedge d_j \in g\}$. The product set is made explicitly non-redundant after calculating the product. If f and g have a disjoint support, then $f \cdot g$ is called the **algebraic product**, otherwise $f \cdot g$ is called the **Boolean product**. A completely specified function g is a **Boolean divisor** of an (incompletely specified) function $\mathcal{F} = (f,d,r)$, if two completely specified functions h and i exist so that $g \cdot h \not\subseteq d$ and $f \subseteq g \cdot h + i \subseteq f + d$, where $g \cdot h$ represents the Boolean product. h is called the quotient and i is called the remainder of the division, similar to the names used in number calculus. However in our case, h and i are not necessarily unique. For example, the function $f = ac + \bar{a}b + bc + d + \underline{e}$ where the underlined cube is a cube from the don't care set, has a Boolean divisor (a + b), with a quotient $(\bar{a} + c)$ and a rest expression d, as $(a + b)(\bar{a} + c) + d = ac + \bar{a}b + bc + d \subseteq f$. Because (a + b) and $(a + \bar{c})$ have intersecting input supports, the product is a Boolean product.

The disadvantage of the Boolean division approach is that the set of Boolean divisors is usually very large and therefore good divisors cannot be easily found. Consequently, an alternative approach has been considered. It is based on the fact that the sum-of-product term representation for two-level functions is almost canonical and efficient common algebraic factors can be identified as being common factors of the product terms. The idea is motivated by the fact that manipulations of sum-of-product terms are in most cases quickly performed (many algebraic operations have linear time complexity). The disadvantage of this idea is that it does not guarantee optimal solutions. Brayton uses an alternative approach based on this idea. In this approach, the incompletely specified function $\mathcal{F}$ is minimised to obtain a two-level minimal representation of the on-set f of $\mathcal{F}$ (using the two-level minimiser Espresso [9]) and algebraic division is used to manipulate f. A completely specified function p is an **algebraic divisor** of a completely specified function f if q and r exist, such that $p \neq 0$, $q \neq 0$ and $f = p \cdot q + r$, where $p \cdot q$ the algebraic product. q is once again called the **quotient.** In the remainder of this article we will denote the quotient as q = f/p; r is called the **remainder** of the division; q and r need not be unique. However, if we define q as the *largest set* for which $f = p \cdot q + r$ then q and r are unique. This special type of algebraic division is called: **weak division**. For example (a + b) is an **algebraic divisor** of the function f = (a + b)(c + d + e) + g. Both q = c + d, r = ae + be + g and q = c + d + e, r = g are valid quotient/remainder pairs for f divided by (a + b). If the calculations are restricted to **weak division** then the second pair is the unique quotient/remainder pair. Note that in $f = p \cdot q + r$, the names of the divisor and the quotient are arbitrary, if p is a divisor and q is the associated quotient, then we can equally call q the divisor and p the quotient.

A cube c is said to (algebraically) **divide expression f evenly** if and only if $\forall d \in f : c \subseteq d$. An expression is said to be **cube free** if no cube divides the expression evenly (e.g. ab + c is cube free, but ab + ac is not cube-free, since cube a divides ab + ac evenly). The **primary divisors** of a Boolean function f are the elements of the set D(f) defined as: D(f) = {f/c | c is a cube}. The set of **kernels** of a Boolean function f is the set: $K(f) = \{g \mid g \in D(f) \wedge g \text{ is cube free}\}$. In other words, the kernels

of an expression f are the cube-free, primary divisors of f. The cube c used to obtain kernel k (k = f/c) is called the **co-kernel** of k, and we use C(f) to denote the set of co-kernels of f. A kernel $k \in K(f)$ is said to be of **level n** if $k \in K^n(f)$ and $k \notin K^{n-1}(f)$, where $K^0(f) = \{k \in K(f) | K(k) = \{k\}\}$ and $K^n(f) = \{k \in K(f) | \exists l \in K(k): l \in K^{n-1}(f)\}$. For example, for the function $f = a \cdot (b + c) + d$ the set of divisors is $D(f) = \{a;\ b + c;\ a(b + c) + d;\ 1\}$. The set kernels for this function are: $K^0(f) = \{b + c\}$ and $K^1(f) = \{b + c;\ a(b + c) + d\}$.

Theorem 1: Functions f and g have a **common multiple cube divisor** d if and only if

$$\exists_{k_f \in K(f)} \ \exists_{k_g \in K(g)} \ d = k_f \cap k_g \ [7].$$

This theorem states that two functions only have a multiple cube divisor if an intersection of a kernel of f and a kernel of g has more than one cube. It is the fundamental theorem used in the factorisation algorithms presented in the next sections.

2.2 Standard Factorisation Algorithm

The algorithm discussed below was introduced by R.K. Brayton in 1982 ([7][8][10]). We will call it the **standard factorisation algorithm**, because all other factorisation algorithms are based on it. Each method presented in this paper will be characterised by sketching an outline of the major steps and by discussing the impact of these steps on the quality of the result. The standard factorisation method can be characterised as follows:

- The fundamentals of the method are based on the notion of **kernel/co-kernel pairs**. This notion is easy to comprehend, which makes easy reasoning of the synthesis process. Although the set of kernels can become very large, it is considered to be reasonably small for most practical synthesis problems. Kernels form a very small subset of all algebraic divisors of a function. Algebraic divisors are a special subset of another group of divisors: the Boolean divisors. Therefore, the set of kernels of a Boolean expression is a very small sub- set of all possible divisors and can be too restrictive to find near optimal solutions.
- The input to the algorithm is the two-level, locally minimised on-set f' of an incompletely specified function $\mathcal{F}(f,d,r)$ obtained using the two-level minimiser Espresso [9]. This makes incompletely specified multiple-output functions much easier to handle, but the loss of the don't cares before the algorithm actually starts will almost certainly lead to a less satisfactory implementation.
- In the type of factorisation problems presented here, two optimisation criteria are generally considered. Firstly, the area occupied by the implementation of the circuit and secondly the maximal speed of the implementation of the circuit. The area of the implementation is usually split into two components: the active area (area used for active elements (transistors)) and the routing area (the area used for wiring). The maximum speed of the implementation is usually determined by its **critical path**; this is defined as the worst case response time of any output to a change in one or more of its inputs.

  In the standard factorisation algorithm, the number of literals is used as the optimisation criterion. The number of literals is a quite accurate measure for the implementation of AND-OR-NOT based Boolean expressions, because each literal is an input of a gate and the difference between the minimum and the maximum number of inputs for a gate of a certain technology is usually small (typical 2 or 3). Although this measure is reasonably accurate for the active area of a physical implementation, it is not adequate enough for the routing area or for the delay of the circuit. Since the **routing area** of complex multiple-level logic circuit can be much larger than the active area and heavy parenthesised expressions increase the delay, the number of literals can be a weak selection mechanism for large logic circuits. It is important to realise that the number of literals is only an adequate measure for the active area if we use an AND-OR-NOT based technology. If more complex gates are possible (like the AND-NOR-21 gate: $f = a \cdot b + c$) then a non-trivial technology mapping is required as this will transform a set of Boolean expressions with minimum literal count to a gate network using the minimum amount of area. This problem is known to be NP-hard. The same problem can occur when the Boolean expression uses operators with too many inputs: for example, the expression $f = a \cdot b \cdot c \cdot d \cdot e \cdot h \cdot i \cdot j$ probably requires technology mapping, because not many technologies support 8-input AND gates. Furthermore, for FPGAs which have an internal structure of programmable networks of look-up tables (like Xilinx FPGAs), the number of literals is in fact completely irrelevant because these look-up tables can implement any function which has a limited number of inputs and outputs (regardless of how many literals are needed to describe this function). The general full-decomposition theory presented in section 3 does not have this disadvantage as in this theory, functions are modelled as networks of arbitrary building blocks.
- Each kernel/co-kernel pair obtained using theorem 1 is applied to all functions and the gain in the number

of literals obtained by this pair is calculated. The kernel/co-kernel pair with the highest literal gain is selected and applied to all functions. Although this approach is fast and easy to understand, we consider it to be a weak point of the algorithm. Firstly, the selection is based on the momentary gain of the number of literals of a kernel. It does not predict the total gain in the number of literals that can be expected by choosing this kernel. This is important because the choice of a kernel may block other kernel/co-kernel pairs and, after choosing the currently best kernel, no good kernel/co-kernel pairs may remain. The second objection is that the selection algorithm is greedy. A greedy search strategy should only be used for solution spaces which are more or less continuous and regular. If the solution space is very rough, a greedy algorithm often finds a local instead of a global optimum. The solution space of logic synthesis is very rough [28] and therefore greedy algorithms are not suitable. A third disadvantage is that the selected kernel is globally applied. This means that a kernel/co-kernel pair with a large gain in some functions is not only applied to the functions where it results in a gain in the number of literals, but also to functions where it is a very bad choice, and therefore may block good kernels for that function. All these factors together provide evidence to show that to guarantee a near-optimal gain in the number of literals, a more sophisticated kernel selection algorithm should be used.

- The kernel/co-kernels of Brayton are not **algebraically compatible**. This means that after choosing and applying a certain kernel, other kernel/co-kernel pairs may no longer be valid. The quality factors of the other kernel/co-kernel pairs can also change. Therefore, after each kernel selection the kernel/co-kernel set needs to be recalculated. This is a rather expensive operation and therefore Brayton tries to select a few kernels/co-kernels before rebuilding the kernel/co-kernel set, which means that the algorithm works with somewhat inaccurate estimations and extra checks are required to assess whether a kernel/co-kernel pair is still feasible.
- The algorithm continues to select the kernel/co-kernel with the highest gain in the number of literals, until the gain is smaller than a threshold value X. Then, all single-cube divisors which have a literal gain larger than X are extracted. If no more multiple-cube or single-cube divisors can be found then the threshold value X is decreased and the extraction process is continued. The choice of the sequence of values of X is determined by experiments (empiric data).

2.3 Lexicographical Factorisation

The **lexicographical factorisation** algorithm [1][2][3][45] was developed in the Laboratoire Conception de Systèmes Intégrés of the Institut National Polytechnique de Grenoble. It aims to improve Brayton's method by removing some of the disadvantages previously mentioned. Its basic idea is to find and use an order of the input variables in the factorised expression. This approach leads to a multiple-level random logic implementation with an improved routing factor compared to the standard factorisation algorithm of Brayton. In the lexicographical factorisation, the variables are factored out in order of appearance in a certain variable ordering. Suppose we have a function $f = abc + abd + ae + b\bar{c} + be + \bar{c}\bar{d}$ and an input order {a, b, c, d, e}, then the factorised form of f with respect to this order is $f = a(b(c + d) + e) + b(\bar{c} + e) + \bar{c}\bar{d}$. The construction of the input order is based on kernel/co-kernel pairs. The **precedence relation** induced by the kernel/co-kernel pair (k, c) states that the variables in co-kernel c precede the variables of kernel k. For example, function $f = \bar{a}\bar{c}(\bar{b}d + b\bar{d})$ has a kernel/co-kernel pair $(\bar{b}d + b\bar{d}, \bar{a}\bar{c})$. The precedence relation is: a and c precede b and d. A **factorisation (k, c) is compatible with a reference order**, if and only if its precedence relation is respected by the reference order. For example, $\bar{b}a(c + d)$ is compatible with all reference orders in which b and a precede c and d. These reference orders are: {a, b, c, d}, {b, a, c, d}, {a, b, d, c} and {b, a, d, c}. Two factorisations (k_1, c_1) and (k_2, c_2) are **lexicographically compatible**, if and only if at least one input order exists with which they are both compatible. Suppose we have two elementary factorisations on some function: $(ce + d, \bar{a}b)$ and $(\bar{d} + f, \bar{c}\bar{e})$. These factorisations are compatible because they are both compatible with the reference order {a, b, c, e, d, f}.

The lexicographical factorisation method can be characterised as follows:

- The **lexicographical algorithm**, like standard factorisation, has the locally two-level minimised Boolean functions as its inputs. The input-order is created in the first step. Part of the input order can be imposed externally to account for external factors (e.g. late arrival times of some input). If this enforced input ordering is not complete, then the following input ordering algorithm is used to complete the order. A list of all kernel/co-kernel pairs is first constructed. The pairs are sorted with respect to their global gain in the number of literals. The pair with the largest gain in the number of literals and which not violates the input order is selected and the input order is updated with regard to the precedence

relation of the selected kernel/co-kernel pair. These steps are repeated until no more compatible kernel/co-kernel pairs can be found.

- The lexicographical factorisation algorithm uses a **greedy selection algorithm** to find a good input order. The constructed input order may not be the best one (in addition to the fact that any input order is a large restriction on the set of kernels). Therefore, a more sophisticated algorithm may be necessary to find a near optimal input order. Related to this problem is the fact that lexicographical factorisation uses the number of literals to estimate the quality of an implementation.
- One of the main advantages of lexicographical factorisation over standard factorisation is the following theorem proven in [45]:

  Theorem 2 Lexicographical compatible kernel/co-kernel pairs are algebraically compatible.

  Lexicographical factorisation therefore does not require the recalculation of the set of kernels/co-kernel pairs after one pair is selected from the set. The result is that lexicographical factorisation is a much faster algorithm when compared with standard factorisation.

- The factorisation is then performed and respects the variable ordering just created. Since the variable ordering is known, factorisation is very simple. Negated variables are factored out immediately after the non-negated variable. In the final step, common sub-expressions are identified and implemented as sub-functions. Because of the input order, the search for common sub-expressions is very efficient. It is performed as the last step, because high priority is given to internal simplifications of the expressions and this results in a low number of wires and short wires.
- The lexicographical factorisation results in implementations which have a much smaller routing area compared to the standard factorisation algorithm of Brayton. Respecting the variable ordering can however result in an increase of the active area. Many good kernels can not be used because the input order is very restrictive. Therefore, the method produces only good results for circuits with a high routing factor (the amount of active area used is larger than the active area obtained using Brayton's method). Most large circuits are known to have a large routing factor and experiments ([45]) have shown that the lexicographical factorisation algorithm produces better results in less time for large circuits.
- Unfortunately, only external inputs are accounted for in this method. The method does not explicitly try to avoid long lines which result from internal sub-function creation (although implicitly this problem is partially solved because the variable ordering keeps related sub-functions close to each other).

2.4 Concurrent Decomposition Algorithm

This method was introduced by Janusz Rajski and Jagadeesh Vasudevamurthy of the McGill University in Montreal, Canada [41][42][50]. It is based on testability preserving transformations and the factorised multiple-level network is fully tested by a complete test set derived from the original two-level circuit. The characteristic feature of the concurrent decomposition method is that it limits itself to the use of double cube divisors (i.e. kernels with only two cubes), single cube divisors with only two literals and the complements of these single and double cube divisors. It has been found that these divisors (in spite of their simplicity) can be used to synthesise circuits with small area and short delay times. Furthermore, by restricting the calculations to double cube divisors and single cube divisors with two literals, the calculations of these divisors are now polynomial time operations (whereas the calculation of the set of kernels can require an exponential amount of time). A **double cube divisor** is a cube-free multiple cube divisor with only two cubes. The set D(f) of all double cube divisors is defined as $D(f) = \{d | \forall_{i,j}: 1 \leq i \leq n,\ 1 \leq j \leq n,\ i \neq j:\ d = \{c_i \setminus (c_i \cap c_j), c_j \setminus (c_i \cap c_j)\}\}$ where n is the number of cubes of f and c_i represents cube i of f. $(c_i \cap c_j)$ is called the **base** of double cube divisor d. Note that the definition of a double cube does not exclude empty bases (i.e the situation for which c_i and c_j are disjoint). The complexity of the construction of D(f) is $O(n^2)$.

Given the function f = ade + ag + bcde + bcg, the double cube divisors of f are

- de + g obtained from the cubes ade and ag or from the cubes bcde and bcg.
- a + bc obtained from the cubes ade and bcde or from the cubes ag and bcg.
- ade + bcg obtained from the cubes ade and bcg.
- ag + bcde obtained from the cubes ag and bcde.

The set of double cube divisors D(f) is represented by a number of subsets $D_{x,y,s}(f)$, where x represents the number of literals in the first cube, y the number of literals in the second cube and s the number of literals in the support of f. Without loss of generality one can assume $x \leq y$. Note that $\max(x,y) \leq s \leq x + y$. The special sets $D_{exor}(f)$ and $D_{exnor}(f)$ denote the EXOR and the EXNOR double cubes respectively; $D_{2,2,2}(f) = D_{exor}(f)\ D_{exnor}(f)$. $S_x(f)$ is used to denote the set of single cube divisors of f with exactly x literals. A feature of concurrent factorisation is that it does not only take elements of D(f) and S(f) as divisors, but also uses the complement of these divisors. In [42] an important theorem is formu

lated that describes the cases in which the complement of a double cube divisor is also a divisor:

Theorem 3: Let f and g be two Boolean functions. If a) $d_i \neq \bar{s}_j$ for every $d_i \in D_{1,1,2}(f)$, $s_j \in S_2(g)$ and b) $d_i \neq \bar{d}_j$ for every $d_i \in D_{exor}(f)$, $d_j \in d_{exnor}(g)$ and c) $d_i \neq \bar{d}_j$ for every $d_i \in D_{exnor}(f)$, $d_j \in D_{exor}(g)$ and d) $d_i \neq d_j$ for every $d_i \in D_{2,2,3}(f)$, $d_j \in D_{2,2,3}(g)$ and e) $d_i \neq \bar{s}_j$ for every $d_i \in D_{1,1,2}(g)$, $\bar{s}_j \in S_2(f)$

then f has neither a complement double cube divisor, nor a complement single cube divisor in g.

This theorem is quite important from the practical viewpoint: by checking for a complement divisor of a $d \in D(f)$, we have to search for them only in a small subset of $D_{x,y,s}(f)$ or $S_2(f)$. As in standard factorisation, a theorem for finding divisors among two functions exists [42].

Theorem 4: Expressions f and g have a common multiple cube divisor if and only if $D(f) \cap D(g) \neq 0$.

Finding multiple cubes based on the sets D(f) and D(g) leads to a higher run-time efficiency because the set of double cube divisors is much smaller when compared to the set of all kernels. The method can be characterised as follows:

- The double cube divisors are constructed by an $O(n^2)$ algorithm, where n is the number of terms. Each double cube divisor d is stored in the appropriate $D_{x,y,s}$ and its implementation cost is calculated. If d has a single cube complement (which can be easily checked using theorem 3) then the cost for the single cube complement is also taken into account in the implementation cost. The quality of a Boolean expression is estimated by the number of literals in the expression. The quality factor specifies the gain in the number of literals which results from choosing both this double cube divisor and its complement. Double cube divisors are extracted separately for each function. The extraction of double cube divisors is only done once for the entire factorisation process. A similar algorithm is used to select all single cube divisors.
- The divisor selection algorithm is a **greedy algorithm** selecting the (single or double cube) divisor (and possibly its complement divisor) with the largest **gain in the number of literals**. Our objections to this measure have already been discussed in section 2.2 and hold for concurrent decomposition. The set of double cube divisors is **not algebraically compatible**. However, because the complexity of the double cube generation algorithm is quadratic, this algorithm is **much faster** when compared to the standard factorisation (which occasionally can have exponential time complexity).
- In order to preserve testability the algorithm has to be used on **single-output functions** and not on multiple-output functions. This means that good sub-expressions cannot be shared among different output functions. Furthermore, the transformation of a multiple-output function to a set of single-output functions can increase the number of product terms by at most a factor equal to the number of outputs in the multiple-output function. In an attempt to overcome this problem, the single-output Boolean functions are searched for common parts prior to executing the concurrent decomposition algorithm. The common parts are implemented as sub-functions. It must be noted that the search for common parts involves common product terms not common subcubes, i.e. the intermediate variables occur in the functions as product terms with this intermediate variable as the only variable in the product term and they are not literals of a cube.
- The algorithm preserves testability but this is a severe restriction. In [42] a number of rules are specified which should be fulfilled in order to preserve testability. These rules state that single cube extraction, double cube extraction and concurrent extraction on single-output functions preserve testability. Similar rules state that it is very difficult to preserve testability in multiple-output circuits (i.e. Boolean expressions with common sub-expressions). Therefore, the concurrent decomposition methods act on single-output functions. It seems reasonable to expect further gain in the number of literals if the testability condition is dropped and common sub-functions and common double cube divisors are allowed.

2.5 Example of Division Based Synthesis Methods

In this section the division based algorithms that are presented are illustrated by an example. Given the function w(a, b, c, d, e, f, g) defined as w(a, b, c, d, e, f, g) = abce + abd + $\bar{a}$ef + $\bar{b}$ef + efg. w is a single output completely specified function and is minimal with respect to the number and the size of the terms. The standard factorisation algorithm presented in section 2.2 applied to w results in the function x(a, b, c, d, e, f, g): x(a, b, c, d, e, f, g) = ab(ce + d) + ef($\bar{a}$ + $\bar{b}$ + g) (see also Figure 1a). The standard factorisation has obtained a solution with 10 literals. The lexicographical factorisation algorithm cannot find this solution because the used co-kernel/kernel pairs are incompatible: the pair (ce + d, ab) requires (among others) the precedence relation a precedes e, whereas the second co-kernel/kernel pair of x ($\bar{a}$ + $\bar{b}$ + g,ef) requires the precedence relation e precedes a, which is incompatible with the first relation. The lexicographical factorisation finds the following factori sation that respects the variable ordering {d, e, c,

f, a, b, g}: $y(a, b, c, d, e, f, g) = d \cdot y_1 + e(c \cdot y_1 + f(\bar{y}_1 + g))$ with $y_1 = a \cdot b$ (See Figure 1b). The function also requires 10 literals. As it can be seen, a subfunction y_1 has been introduced. Because the lexicographical factorisation algorithm uses NANDs as an internal representation, it was able to identify ab and $\bar{a} + \bar{b}$ as common subexpressions. If this realisation is compared with standard factorisation then it shows the properties of lexicographical factorisation: each input is only once connected to a gate and therefore the routing complexity for the inputs is reduced. It can also been seen that the method does not try to minimise wires for subfunctions as the subfunction y_1 is routed globally. It is however possible to specify a threshold on the gain of the number of literals for sub-functions. If the gain of a certain sub-function is larger than the threshold, the subfunction is created and applied globally (expecting a large active area gain, but small extra area for wiring), otherwise it is implemented locally (costing extra active area, but negligible routing area). Furthermore, the lexicographical factorisation algorithm needs more gates than standard factorisation (increase of active area). The real power of the lexicographical factorisation algorithm can only be shown on large examples, where the gain in active area and the extra routing area for sub-functions is compensated by a large reduction in the routing area for the inputs. We refer to the benchmark results in [45] to illustrate the effectiveness of the lexicographical factorisation for large circuits. Using lexicographical factorisation, the partial input order can also be enforced. Suppose we want the inputs a,b and c to be extracted first (because these inputs have late arrival times due to external circumstances), the input ordering is first completed as {a,b,c,d,e,f,g} and the following factorisation is then found: $z = ab(ce + d) + (\bar{a}z_1 + \bar{b}z_1 + gz_1)$ and $z_1 = ef$ (see Figure 1c) which requires 13 literals. It should be noted that the second part of z cannot be written as $z_1(\bar{a} + \bar{b} + g)$ without violating the precedence relation. Also, it should be noted that the critical path of a and b has reduced from 6 gates to 2 gates at the expense of a somewhat more complex routing and an increase in active area (more inputs per gate). The concurrent factorisation algorithm searches explicitly for the complement of the kernel $\bar{a} + \bar{b}$, whereas in lexicographical factorisation this equivalence was implicitly found. Concurrent factorisation finds the same factorisation as lexicographical factorisation (function y) (see Figure 1b). As it can be seen only double cube divisors and cubes with two literals are used.

3. GENERAL DECOMPOSITION-BASED MULTIPLE-LEVEL LOGIC SYNTHESIS

In this section, we will present a theory of general full-decomposition for combinational machines and give an overview of the decomposition-based methods for multiple-level combinational logic synthesis. Basic definitions are presented in sub-section 3.1, the theory of general full-decomposition can be found in sub-section 3.2 and some special decomposition cases are the topics of sub-section 3.3.

3.1 Basic Definitions

A (completely specified) **combinational machine** M is an algebraic system defined by:

$$M = (I, O, \lambda)$$

where:

I - a finite non-empty set of inputs,
O - a finite non-empty set of outputs,
λ - the output function $\lambda: I \rightarrow O$.

The design requirements do not always completely specify a machine for example, certain input values may never occur due to external constraints or due to realizing the machine in such a way that some of the input values of the realization are not used for implementing the inputs of the originally specified machine. From the behavioural viewpoint, the designer does not care what will be the output value for such an input value. In all such situations one talks about so called "don't care" conditions. "Don't cares" are commonly denoted by "-". In order to account for them, the combinational machine definition should be slightly modified by changing the definition of the machine function λ. For the single-output machine: $\lambda: I \rightarrow O \cup \{-\}$. For the multiple-output machine: $\lambda = [\lambda_i]$, $\lambda_i: I \rightarrow O_i \cup \{-\}$ and $O = [O_i]$. A combinational machine without "don't cares" will be referred as **completely specified** and with "don't cares" as **incompletely specified**. Machine $M' = (I', O', \lambda')$ is a **realisation** of machine $M = (I, O, \lambda)$ (see Figure 2) if and only if the relations $\Psi: I \rightarrow I'$ (a function) and $\Theta: O' \rightarrow O$ (a surjective partial function) exist, so that $\forall x \in I$: $\lambda(x) = \Theta(\lambda'(\Psi(x)))$. The machine composed as a structure consisting of Ψ, M' and Θ and being the **realisation structure** for M defined by M' will be denoted by str(M'). It is possible to prove, that if M' is a realisation of M then for all possible inputs, the outputs produced by machine M and its realisation M' are identical after renaming.

Partitions and partition pairs, originally introduced by Hartmanis [19] are useful for modelling information and information flows inside and between machines. Let S be any set of elements. **Partition** π on S is defined as follows: $\pi = \{B_i | B_i \subseteq S$ and $B_i \cap B_j = 0$ for $i \neq j$ and $\bigcup_i B_i = S\}$ i.e. a partition π on S is a set of disjoint subsets of S whose set union is S. For a given $s \in S$, the **block of a partition π containing s** is denoted as $[s]\pi$ while $[s]\pi = [t]\pi$ denotes that **s and t are in the same block of π**. Similarly, the block of a partition π

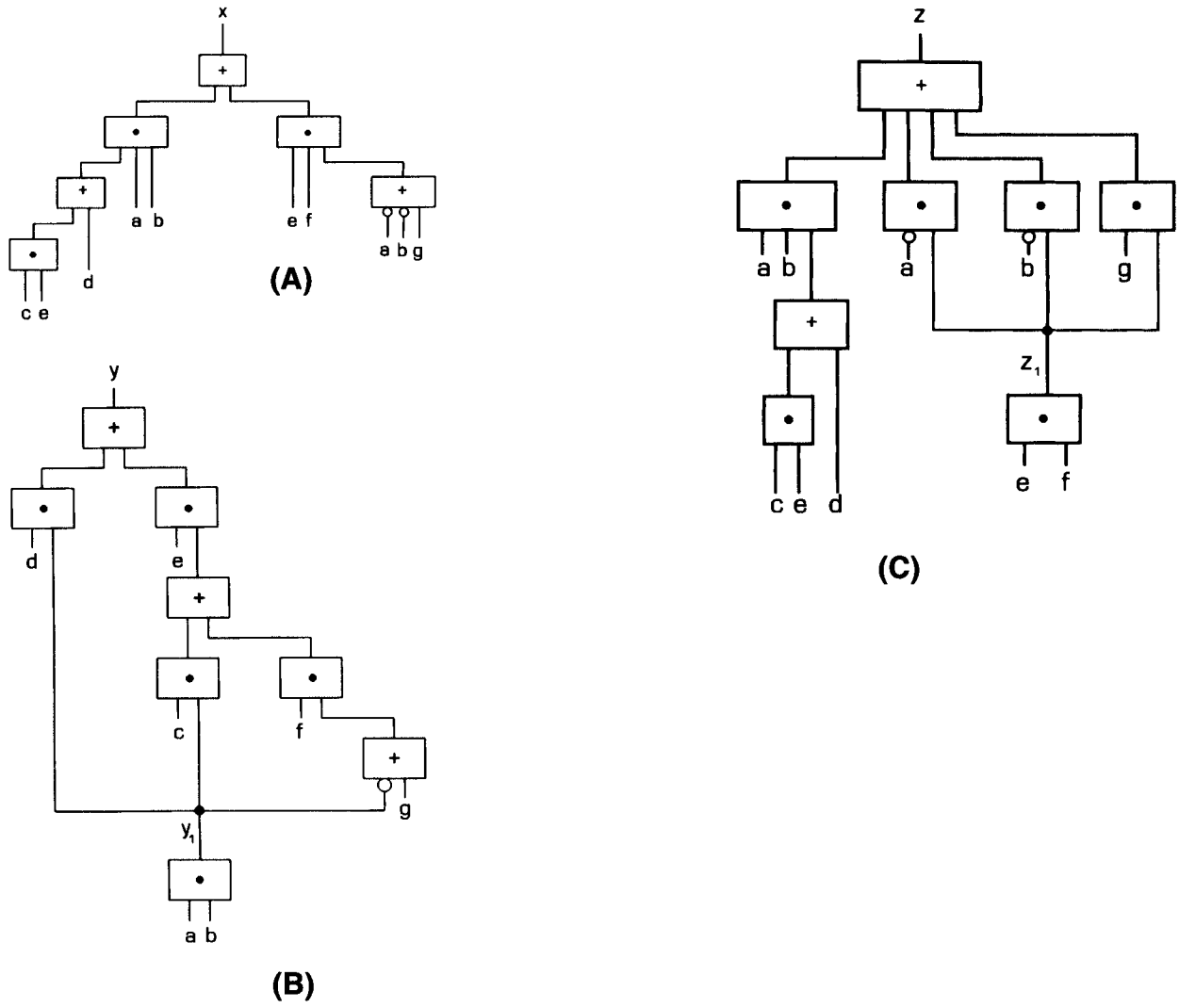

FIGURE 1 Different division based realisations of function w. (a) Standard factorisation. (b) Lexicographical factorisation without predefined input order and concurrent factorisation algorithm. (c) Lexicographical factorisation with partial defined input order {a,b,c}.

containing S', where $S' \subseteq S$, is denoted by $[S']\pi$. The partition containing each element of S in a separate block is called a **zero partition** and denoted by $\pi_S(\mathbf{0})$. The partition containing all the elements of S in one block is called an **identity partition** and is denoted by $\pi_S(\mathbf{I})$. Let π_1 and π_2 be two partitions on S. The **partition product** $\pi_1 \cdot \pi_2$ is the partition on S such that $[s]\pi_1 \cdot \pi_2 = [t]\pi_1 \cdot \pi_2$ if and only if $[s]\pi_1 = [t]\pi_1$ and $[s]\pi_2 = [t]\pi_2$. The partition sum $\pi_1 + \pi_2$ is the partition on S such that $[s]\pi_1 + \pi_2 = [t]\pi_1 + \pi_2$ if and only if a sequence: $s = s_0$, $s_1, \ldots, s_n = t$, $s_i \in S$ for $i = 1..n$, exists for which either $[s_i]\pi_1 = [s_{i+1}]\pi_1$ either $[s_i]\pi_2 = [s_{i+1}]\pi_2$, $0 \leq i \leq n-1$. From the above definitions, it follows that the blocks of $\pi_1 \cdot \pi_2$ are obtained by intersecting the blocks of π_1 and π_2, while the blocks of $\pi_1 + \pi_2$ are obtained by uniting all the blocks of π_1 and π_2 which contain common elements. π_2 is greater than or equal to π_1: $\pi_1 \leq \pi_2$ if and only if each block of π_1 is included in a block of π_2. Thus $\pi_1 \leq \pi_2$ if and only if $\pi_1 \cdot \pi_2 = \pi_1$ if and only if $\pi_1 + \pi_2 = \pi_2$. Any partition π on S can be interpreted as an equivalence relation defined on S with the equivalence classes being the blocks of π. Using this interpretation, the partition π gives information about the elements of S with precision to the equivalence class. With this information, it is possible to distinguish elements from different classes although it is impossible to distinguish elements from the same class. The partial ordering relation $\leq$ denotes the fact that if $\pi_1 \leq \pi_2$ then π_1 (and so the associated equivalence relation) provides information about elements of S, that is at least as precise as information given by π_2 (and its associated equivalence relation). A zero partition provides complete information

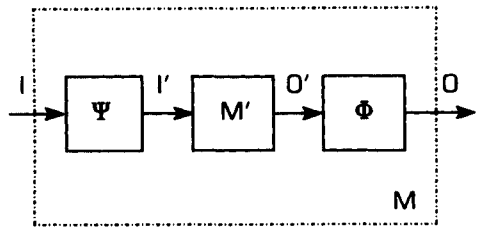

FIGURE 2 Realisation of machine M by machine M'.

TABLE I
Boolean function for example 1

I	$x_1\ x_2\ x_3$	$y_1\ y_2$	O
a	0 0 0	0 1	x
b	0 0 1	1 1	z
c	0 1 0	0 0	w
d	0 1 1	1 0	y
e	1 0 0	1 0	y
f	1 0 1	1 1	z
g	1 1 0	0 0	w
h	1 1 1	0 0	w

about elements of S and an identity partition gives no information. The partition product can be interpreted as a product of the appropriate equivalence relations introduced by these partitions; it represents the combined information about the elements of S provided by both relations together. The partition sum can be interpreted as a sum of the appropriate equivalence relations introduced by these partitions and it represents the combined abstraction of both relations.

Example

In Table I, the function table of an incompletely specified Boolean function is presented. The function has 3 input bits (x_1, x_2, x_3) and two output bits (y_1 and y_2). Each input combination has been labelled with a unique name (a, b, c, d, e, f, g and h) and hence we can use these symbolic names in the following considerations. Likewise, the output combinations have been labelled by w, x, y and z. These functions can be written as the following completely specified machine: $M(I,O,\lambda)$ such that $I = \{a,b,c,d,e,f,g,h\}$, $O = \{w,x,y,z\}$ and $\lambda: I \rightarrow O$ as specified in Table I. Machine M_r: $M_r(I_r, O_r, \lambda_r)$, with $I_r = \{1,2,3,4,5,6,7,8\}$, $O_r = \{\alpha,\beta,\gamma,\delta\}$, $\lambda_r: I_r \rightarrow O_r$, $\lambda_r = \{(1,\beta);(2,\delta);(3,\alpha);(4,\gamma);(5,\gamma);(6,\delta);(7,\alpha);(8,\alpha)\}$ is a realisation of M, since the mappings $\Psi: I \rightarrow I_r$, $\Psi = \{(a,1);(b,2);(c,3);(d,4);(e,5);(f,6);(g,7);(h,8)\}$ and $\Theta: O_r \rightarrow I$, $\{(\alpha,w);(\beta,x);(\gamma,y);(\delta,z)\}$ satisfy the relation $\forall x \in I: \lambda(x) = \Theta(\lambda_r(\Psi(x)))$. For example, take x = f: $\lambda(f) = z$ (see Table I) and $\theta(\lambda_r(\Psi(f)) = \theta(\lambda_r(6)) = \theta(\delta) = z$. Verification of this relation for other combinations can be performed by the reader as an exercise. For machine $M(I,O,\lambda)$ as defined above $\pi_I(0) = \{\overline{a;b;c;d;e;f;g;h}\}$ is the zero input partition and $\pi_I(I) = \{\overline{a,b,c,d,e,f,g,h}\}$ is the input identity partition. Let $\pi_1 = \{\overline{a,b};\ \overline{c,d};\ \overline{e,f,g,h}\}$ and $\pi_2 = \{\overline{a,b};\ \overline{c,e};\ \overline{d,f,g,h}\}$ be partitions on set I. The product $\pi_1 \cdot \pi_2 = \{\overline{a,b};\ \overline{c};\ \overline{d};\ \overline{e};\ \overline{f,g,h}\}$ denotes the combined information of π_1 and π_2, for example in π_1 the symbols c and d are equivalent and hence partition π_1 cannot distinguish between these two symbols. π_2 can make the distinction between c and d (because they are in different blocks of π_2). The product $\pi_1 \cdot \pi_2$ represents the partition that makes the union of the distinctions of π_1 and π_2 and combines in one block only those elements which are equivalent in both partitions. Similarly, the sum $\pi_1 + \pi_2 = \{\overline{a,b};\ \overline{c,d,e,f,g,h}\}$ represents the information about distinctions present in both partitions. Finally, for the partitions $\pi_3 = \{\overline{a,b};\ \overline{c,d};\ \overline{e,f,g,h}\}$ and $\pi_4 = \{\overline{a,b};\ \overline{c,d,e,f,g,h}\}$, $\pi_3 \leq \pi_4$, because π_3 makes all distinctions that π_4 makes. This can also be checked by the definition of $\leq$: $\pi_3 \leq \pi_4 \Leftrightarrow \pi_3 \cdot \pi_4 = \pi_3$ and $\pi_3 \cdot \pi_4 = \{\overline{a,b};\ \overline{c,d};\ \overline{e,f,g,h}\} = \pi_3$. $\leq$ is a partial relation: therefore it need not be defined for all pairs of partitions: $\pi_5 = \{\overline{a,b};\ \overline{c,d};\ \overline{e,f,g,h}\}$ and $\pi_6 = \{\overline{a,c,e};\ \overline{b,f};\ \overline{d,g,h}\}$ are not related by $\leq$ since $\pi_5 \cdot \pi_6 = \{\overline{a,b};\ \overline{c,d};\ \overline{e,f,g,h}\} \cdot \{\overline{a,c,e};\ \overline{b,f};\ \overline{d,g,h}\} = \{\overline{a};\overline{b};\overline{c};\overline{d};\overline{e};\overline{f};\overline{g};\overline{h}\} = I$.

Given $M = (I,O,\lambda)$, let π_I be a partition on I and let π_O be a partition on O. (π_I, π_O) is an **I-O partition pair** if and only if $\forall A \in \pi_I \overline{\lambda(A)} \subseteq C$, $C \in \pi_O$ (where: $\overline{\lambda(A)} = \{\lambda(x) | x \in A\}$) ; i.e (π_I, π_O) is an I-O partition pair if and only if each block of π_I unambiguously determines the block of π_O in which the output is contained. If (π_I, π_O) is a partition pair then π_I is called the first partition of the pair and π_O is called the second partition of that pair. Let π_I be a partition on I. The **minimal second partition** which forms an I-O partition pair with π_I as a first partition will be denoted $\mathbf{m}_{I\text{-}O}(\pi_I)$. The **maximal first partition** which forms an I-O partition pair with π_O as a second partition will be denoted $\mathbf{M}_{I\text{-}O}(\pi_O)$. It can be proved [19] that:

$$m_{I\text{-}O}(\pi_I) = \Pi \{\pi_j | (\pi_I, \pi_j) \text{ is a I-O partition pair}\}$$

$$M_{I\text{-}O}(\pi_O) = \Sigma \{\pi_j | (\pi_j, \pi_O) \text{ is a I-O partition pair}\}$$

For a given π_I, $m(\pi_I)$ describes the largest amount of information which can be computed about the output of M knowing the block of π_I which contains the input. $M(\pi_O)$ describes the least amount of information which must be known about the input of M, in order to be able to compute the information about the output with precision to π_O. π'_I is **an input partition induced by an output partition** π_O' (notation: $\pi'_I = ind(\pi'_O)$) if and only if: $\forall x,y \in I$: if $[\lambda(x)]\pi'_O = [\lambda(y)]\pi'_O$ then $[x]\pi'_I = [y]\pi'_I$. In other words, if π'_I is an input partition induced by an output partition π_O' and, if it is known that the present output y of M is contained in a block $C \in \pi'_O$, then

it is known that current input I of M is contained in a block $B \in \pi_I'$, where block B is unambiguously indicated by block C. It is possible to prove that π_I' is an input partition induced by an output partition π_O' if and only if $\pi_I' \geq M_{I\text{-}O}(\pi_O')$, i.e. the smallest input partition induced by a certain π_O' is $\pi_I' : \pi_I' = M_{I\text{-}O}(\pi_O')$.

For the purpose of a bit decomposition (in which the input/output *bits* are appropriately distributed instead of the input/output *symbols*), the concepts of bit partitions has been introduced [25]. Let $B = \{b_1, b_2, \ldots, b_{|B|}\}$ be a set of (input or output) bits. Let $T = \{t_1, t_2, \ldots t_{|T|}\}$ be a set of (input or output) symbols. Each input/output bit $b_k \in B$, introduces a two-block partition $\pi_T(b_k)$ on the set of symbols (bit value patterns) T (in the case of incompletely specified machines on the subset of T for which the value of this bit is specified). One block of $\pi_T(b_k)$ contains the symbols for which bit b_k has the value 0 and the other block contains the symbols for which b_k has the value 1. The product of the partitions $\pi_T(b_k)$ for all the bits b_k: $b_k \in B$ will unambiguously define the set of all input/output symbols, i.e. it will be a zero partition. A partition π_B on the set of bits B $\pi_B = \{b_1, b_2, \ldots, b_k, (b_{k+1}, \ldots, b_{|B|})\}$ is called a **bit-partition**. In a bit-partition the important bits (for distinguishing between certain symbols) $b_1, \ldots, b_k$ are kept in separate blocks and the don't care bits $b_{k+1}, \ldots, b_{|B|}$ are kept in a single block called a **don't care block** (denoted by $\mathbf{dcb}(\pi_B)$). The product $(\cdot)$ and sum $(+)$ operations as well as the ordering relations $(\leq)$ for bit partitions are defined in the same way as for "normal" partitions with the following supplement: the product of a block (important or don't care) with important blocks is an important block in the product partition; whereas the sum of a block (important or don't care) with a don't care block is a don't care block in the sum partition. The **zero bit-partition** is defined as a bit partition with an empty don't care block. The **identity bit-partition** is defined as a bit-partition with all elements in the don't care block. π_T is a **symbol partition induced by a bit partition** π_B ($\pi_T = \text{ind}(\pi_B)$) if and only if $\pi_T \geq \prod_{b_k \in (B - \text{dcb}(\pi_B))} \pi_T(b_k)$. π_B is a **bit partition induced by a symbol partition** π_T ($\pi_B = \text{ind}(\pi_T)$) if and only if $\forall b_k \in (B - \text{dcb}(\pi_B)) : \pi_T(b_k) \geq \pi_T$. If $\pi_T = \text{ind}(\pi_B)$ then, having π_B the blocks of π_T can be computed. If $\pi_B = \text{ind}(\pi_T)$ then, having the block of π_T the values of all the important bits from π_B can be computed.

Example (continued)

The function from Table I and its associated machine description M(I,O,λ) is again used. Given the partition $\pi_I = \{\overline{a,c}; \overline{b,f}; \overline{d,e}; \overline{g,h}\}$ on set I and partition $\pi_O = \{w,x; \overline{y,z}\}$ on set O. (π_I, π_O) is a I-O partition pair and this can be easily checked by checking all blocks of π_I following the definition (e.g. $\overline{\lambda\{a,c\}} = \{x,w\}$ which is a subset of a block of π_O etc.). $m_{I\text{-}O}(\pi_I) = \{\overline{w,x}; \overline{y}; \overline{z}\}$ represents the maximal information about the output of M that can be calculated using π_I. Similar $M_{I\text{-}O}(\pi_O) = \{\overline{a,c,g,h}; \overline{b,d,e,f}\}$ represents the minimal input information that is necessary as a input to calculate π_O. Let $\pi_O' = \{\overline{w,x}; \overline{y}; \overline{z}\}$ then $\{\overline{a,c,g,h}; \overline{b,d,e,f}\}$ and $\{\overline{a,c,g,h}; \overline{b,f}; \overline{d,e}\}$ are both induced input partitions of π_o'. The function from Table I as a binary function instead as a symbolic function will now be considered. Bit-partitions on the inputbits can be made. The set with the input bits is called X, i.e. $X = \{x_1, x_2, x_3\}$. In fact, an input bit represents a symbolic partition which contains in the first block all symbols for which this bit is 0 and in the second block, all symbols for which this bit is 1: $\pi_X(x_1) = \{\overline{a,b,c,d}; \overline{e,f,g,h}\}$, $\pi_X(x_2) = \{\overline{a,b,e,f}; \overline{c,d,g,h}\}$ etc. A bit partition on X is the partition $\pi_X = \{x_1, x_2, (x_3)\}$. Given any partition τ, τ is a symbol partition induced by π_X if and only if: $\tau \geq \prod_{b_x \in (X - \text{dcb}(\pi_x))} \pi_X(b_x)$, which can be calculated as follows: $\tau \geq \pi_X(x_1) \cdot \pi_X(x_2) \Rightarrow \tau \geq \{\overline{a,b,c,d}; \overline{e,f,g,h}\} \cdot \{\overline{a,b,e,f}; \overline{c,d,g,h}\} \Rightarrow \tau \geq \{\overline{a,b}; \overline{c,d}; \overline{e,f}; \overline{g,h}\}$. Similarly, an example of a bit partition induced by the symbol partition $\{\overline{a,c,e}; \overline{b,d,f,h}; \overline{g}\}$ is the partition $\tau_X = \{\overline{x_3}, \overline{(x_1,x_2)}\}$. In this case τ_X is the only possible bit partition.

3.2 General Full-Decomposition

A theory of general decomposition of sequential machines is presented in [29]. In this paper we are concerned with the synthesis of combinational logic however, a combinational machine is merely a special case of a sequential machine with one state and a trivial next-state function. Therefore, the general decomposition theory can also be applied to our problem. A special case of the general full-decomposition theory [29] related to combinational circuits is presented below.

In a general full-decomposition of a combinational machine $M = (I,O,\lambda)$ we need to find a composition of n cooperating partial machines $M_i = (I_i, O_i, \lambda_i)$ as well as the mappings $\Psi : I \rightarrow \times I_i$ and $\theta : \times O_i \rightarrow O$ in order that the composition of M_i together with the mappings Ψ and θ realize machine M. The implementation of the general decomposition model requires three components: the input coder (pre-processor) Ψ, the simultaneously working communicating component machines (main processors) M_i and the output decoder (post-processor) θ. The component machines, input coder and the output decoder are implemented as combinational circuits. The model is general and it contains all elements necessary for the construction of circuit networks which implement combinational circuits: parallel processing elements with possibilities for information exchange between them; divergent pre-processing elements for abstracting and splitting information and representing it in the appropriate form; and convergent post-processing elements for

joining and combining information from parallel processors and representing it in the appropriate form. The full-decomposition can be characterised by the type of connections between the component machines and by the type of input/output encoding/decoding. In a general composition, each partial machine can use (partial) output information from another in order to compute its own output. However, two special cases of a general composition are possible: a parallel and a serial composition. In a parallel composition, no connections exist between the partial machines. Each partial machine is able to compute its own output independently. In a serial composition, machines are ordered and only the component machines M_i, $i \geq j$, can use information from the machines M_j in order to compute its own output. The formal definition for a general composition is given below.

A **general composition** of n combinational machines M_i: $GC(\{M_i\}, \{Con_i\})$, consists of the following objects:

(1) $\{M_i=(I^*_i, O_i, \lambda^i), I^*_i = I_i \times I'_i, 1 \leq i \leq n\}$, a set of machines referred to as **component machines**

(2) $\{Con_i: \times O_j \rightarrow I'_i, 1 \leq i,j \leq n\}$, a set of surjective functions referred to as **connection rules** of the component machines.

A general composition is said to be in **canonical form** if and only if the connections rules Con_i compute the vector values and have the following form: $Con_i(y_1,\ldots,y_n) = (Con_{1,i}(y_1),\ldots, Con_{n,i}(y_n))$, i.e a (partial) output information j, $1 \leq j \leq n$, is separately transmitted to the input of a certain machine i, $1 \leq i \leq n$, i.e. without combining it with a (partial) output information from other partial machines k, $1 \leq k \leq n$, $k \neq j$. A general composition is said to be in **maximally pre-processed form**, if the connection rules Con_i compute the scalar values i.e. information transmitted from various partial machines to a certain machine is combined prior to connection to the input of this machine. Of course, the compositions **in partially pre-processed form** lying between the two above extremes, are also possible. A general composition GC of n combinational machines defines the **general composition machine** $M_{GC}(GC) = M_{GC}(\{M_i\}, \{Con_i\}) = (I_{GC}, O_{GC}, \lambda_{GC})$ with $I_{GC} = \times I_i$, $O_{GC} = \times O_i$, λ_{GC}: $I_{GC} \rightarrow O_{GC}$, $\lambda_{GC} = \times \lambda_i(x_i, Con_i(y_1,\ldots,y_n))$. We will not distinguish between the general composition and the composition machine it defines, unless this can lead to misunderstanding.

The combinational machine $str(M_{GC})$ is a **general full-decomposition** of machine M if and only if the general composition machine M_{GC} realizes M (see Figure 3 for the case of two partial machines). In a similar way, formal definitions for parallel and serial compositions and decompositions can be introduced. In [29] the following theorem has been proved.

Theorem 5 A combinational machine M has a general full-decomposition if and only if n partition doubles (π_I^i, π_I^{*i}) exist and they satisfy the following conditions:

(1) $\pi^i_I \cdot \pi'^i_I \leq \pi^{*i}_I$, where: $\pi'^i_I \geq \prod_{i=1\ldots n} \pi^{*i}_I$

(2) $\prod_{i=1\ldots n} \pi^i_I \leq \pi^{*i}_I$

(3) $\left(\prod_{i=1\ldots n} \pi^{*i}_I, \pi_O(0)\right)$ is an I-O partition pair.

A special case of theorem 5 for two combinational machines is presented below. For simplicity in presentation we will use this case in the sequel of the paper.

Theorem 6 A combinational machine M has a **general full-decomposition with two partial machines and without local connections** if and only if two partition doubles (π_I, π^*_I) and (τ_I, τ^*_I) exist that satisfy the following conditions:

(1) $\pi_I \cdot \tau'_I \leq \pi^*_I$ and $\tau_I \cdot \pi'_I \leq \tau^*_I$, where $\pi'_I \geq \pi^*_I$ and $\tau'_I \geq \tau^*_I$

(2) $\pi_I \cdot \tau_I \leq \pi^*_I$ and $\pi_I \cdot \tau_I \leq \tau^*_I$

(3) $(\pi^*_I \cdot \tau^*_I, \pi_O(0))$ is an I-O partition pair.

Proof

There follows only an outline of the proof in order to show how to construct a decompositional realization structure that is based on partition doubles (π_I, π^*_I) and (τ_I, τ^*_I). Let $M_1 = (\pi_I \times \tau'_I, \pi^*_I, \lambda^1)$ and $M_2 = (\tau_I \times \pi'_I, \tau^*_I, \lambda^2)$ be the two machines for which the following conditions are satisfied:

(4) (π_I, π^*_I) and (τ_I, τ^*_I) satisfy the conditions of Theorem 6,

(5) $\forall A1 \in \pi_I \quad \forall B2' \in \tau'_I$: $\lambda^1(A1, B2') = [\{x | x \in A1 \wedge x \in B2'\}]\pi^*_I$

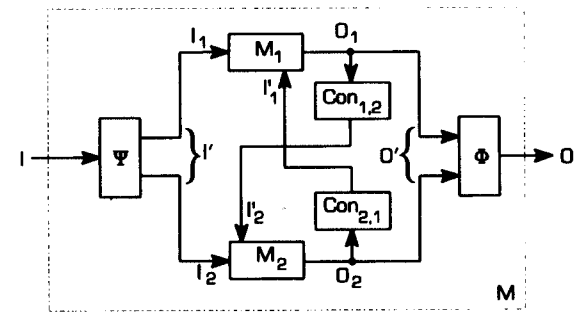

FIGURE 3 General full-decomposition of machine M_i into two component machines M_1 and M_2 without local connections.

(6) $\forall A2 \in \tau_I \quad \forall B1' \in \pi'_I: \quad \lambda^2(A2,B1') = [\{x | x \in A2 \wedge x \in B1'\}]\tau^*_I$

Since condition (1) of theorem 6 is satisfied, $\lambda^1(A1,B2')$ and $\lambda^2(A2,B1')$ are unambiguously defined, i.e. M_1 and M_2 are two well-defined deterministic combinational machines which are able to compute their outputs from their inputs. Let $Con_{1,2}$ and $Con_{2,1}$ be two functions defined as follows: $Con_{1,2}: \pi^*_i \rightarrow \pi'_I$ and $Con_{2,1}: \tau^*_i \rightarrow \tau'_I$. $Con_{1,2}(B1) = [B1]\pi'_I$ and $Con_{2,1}(B2) = [B2]\tau'_I$. Since $\pi'_I \geq \pi^*_I$ and $\tau'_I \geq \tau^*_I$, $Con_{1,2}$ and $Con_{2,1}$ are two well defined functions which are able to unambiguously compute their values from their arguments. It is clear that the above-detailed construction of machines M_1 and M_2 and functions $Con_{1,2}$ and $Con_{2,1}$ is a general composition of machines M_1 and M_2 without local connections. Since condition (2) is satisfied, it is possible to construct the general composition of M_1 and M_2 as a legal composition, i.e. the exchanged information can be computed (directly or indirectly) from the primary input information of partial machines.

Let $\Psi: I \rightarrow \pi_I \times \tau_I$ be a function, $\theta: \pi^*_I \times \tau^*_I \rightarrow O$ be a surjective partial function, and

(7) $\Psi(x)$: $([x]\pi_I, [x]\tau_I)$, and

(8) $\theta(B1,B2) = B1 \cap B2$ if $B1 \cap B2 \neq \varnothing$.

Since, $(\pi^*_I \cdot \tau^*_I, \pi_O(0))$ is an I-O partition pair (3), the output of the original machine M can be unambiguously computed by θ from the outputs of the partial machines M_1 and M_2. $\Psi(x)$ unambiguously computes the inputs of M_1 and M_2. Therefore, the general composition of the machines M_1 and M_2 as defined above, realizes the output behaviour of machine M. Construction of the decompositional realisation structure following theorem 6 is strictly analogous. By repeated use of the general full-decomposition model or its special cases, all functionally correct combinational circuit structures can be constructed.

Example

In Table II, the specification of a completely specified function f is given. Our goal is to implement the function using a minimum number of look-up tables with two inputs and one output. All three factorisation algorithms described in Section 2 find the following implementation f_1 of f: $f_1 = \bar{x}_2\bar{x}_3\bar{x}_4 + x_2(x_3\bar{x}_4 + \bar{x}_1x_4)$. These functions must then be mapped on two-input one-output gates. The only possible technology mapping that can be performed without repeating the synthesis, i.e. without destroying the structure obtained from the factorisation process, is presented in Figure 4a. It results in a circuit with 7 look-up tables. The prototype of an algorithm that is currently being developed by the authors is able to find the solution with 4 look-up tables as presented in Figure 4b. Since, inputs symbols are already binary encoded, we have chosen to use an input coder which achieves a direct distribution of a subset of the input bits to each machine C and E (see Figure 4c). Because one-output gates, are used, we can only implement π^*_C and π^*_E when they only have two blocks. The following two partition doubles satisfy theorem 6: (π_C, π^*_C) and (π_E, π^*_E), where $\pi^*_C = \{\overline{0,1,2,3,4,5,6,7,8,10,12,14};\overline{9,11,13,15}\}$ and $\pi^*_E = \{\overline{0,5,6,7,8,13,14,15};\overline{1,2,3,4,9,10,11,12}\}$. π_C is the input symbol partition induced by the inputs x_1 and x_4, i.e. $\pi_C = ind(\{\bar{x}_1;\bar{x}_4;(\overline{x_2,x_3})\}) = \{\overline{0,2,4,6};\overline{1,3,5,7};\overline{8,10,12,14};\overline{9,11,13,15}\}$. Similarly: $\pi_E = ind(\{\bar{x}_2;\bar{x}_3;\bar{x}_4;(\bar{x}_1)\}) = \{\overline{0,8};\overline{1,9};\overline{2,10};\overline{3,11};\overline{4,12};\overline{5,13};\overline{6,14};\overline{7,15}\}$. Since the decomposition in Figure 4c is a parallel decomposition, conditions (1) and (2) in theorem 6 are satisfied trivially (there is no information flow from one machine to the other) and hence π'_I and τ'_I are identity partitions. We then need to show that the partition doubles satisfy condition (3) of theorem 6. This is relatively easy: $\pi^*_C \cdot \pi^*_E = \{\overline{0,5,6,7,8,14};\overline{1,2,3,4,10,12};\overline{13,15};\overline{9,11}\}$, inspection of Table II for the blocks of this product partition, shows that the product indeed forms an I-O partition pair with $\pi_O(0)$. In a second iteration of the algorithm, block E is further serially decomposed into two blocks (Figure 4d). The partition doubles (π_A, π^*_A) and (π_B, π^*_B) where, $\pi_A = ind(\bar{x}_3;\bar{x}_4;(\overline{x_1,x_2})) = \{\overline{0,4,8,12};\overline{1,5,9,13};\overline{2,6,10,14};\overline{3,7,11,15}\}$,

$\pi^*_A = \{\overline{0,4,8,12};\overline{1,2,3,5,6,7,9,10,11,13,14,15}\}$,

$\pi_B = ind(\bar{x}_2;(\overline{x_1,x_3,x_4})) = \{\overline{0,1,2,3,8,9,10,11};\overline{4,5,6,7,12,13,14,15}\}$, and

$\pi^*_B = \pi^*_E = \{\overline{0,5,6,7,8,13,14,15};\overline{1,2,3,4,9,10,11,12}\}$ realize the behaviour of block E.

TABLE II
Boolean function for example 2

I	x_1 x_2 x_3 x_4	f
0	0 0 0 0	1
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	0
4	0 1 0 0	0
5	0 1 0 1	1
6	0 1 1 0	1
7	0 1 1 1	1
8	1 0 0 0	1
9	1 0 0 1	0
10	1 0 1 0	0
11	1 0 1 1	0
12	1 1 0 0	0
13	1 1 0 1	0
14	1 1 1 0	1
15	1 1 1 1	0

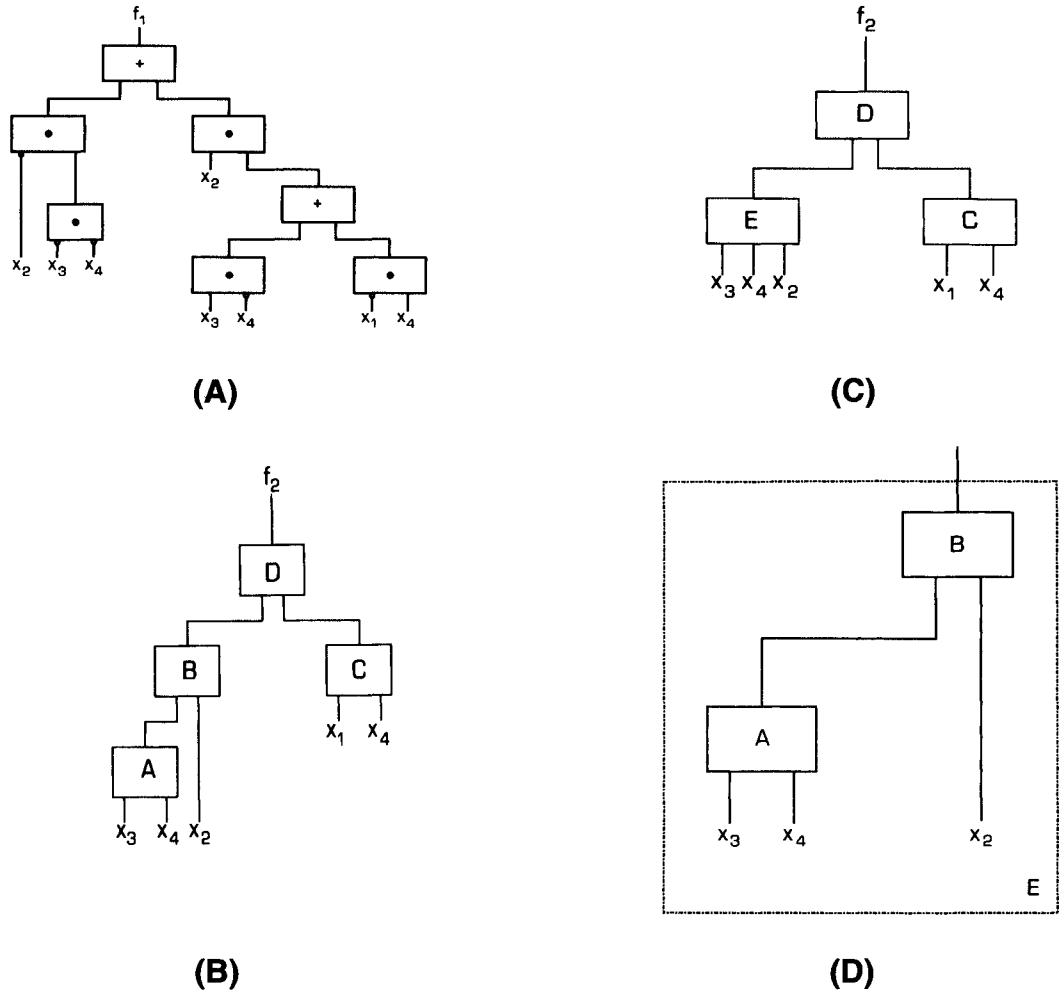

FIGURE 4 Comparison of division-based and general full-decomposition based realisations for function f. (a) Division-based gate-network (b) General-full decomposition based gate-network (c) Step 1 of the General-full decomposition based algorithm (d) Step 2 of the General-full decomposition based algorithm

Since machine B uses output information from machine A, but machine A does not use any output information from machine B, theorem 6 is slightly simplified:

(1) $\pi_A \leq \pi^*_A$ $\pi_B \cdot \pi'_A \leq \pi^*_B$ where $\pi'_A \geq \pi^*_A$,

(2) $\pi_A \cdot \pi_B \leq \pi^*_A$ and $\pi_A \cdot \pi_B \leq \pi^*_B$

(3) $\pi^*_A \cdot \pi^*_B \leq \pi^*_E$

It should be noted that condition (3) is slightly modified. Since we are building a subfunction for implementing π^*_E the partitions π^*_A, π^*_B and π^*_E are all partitions on I and the partition pair property is reduced to the $\leq$ property. It must be shown that the partition doubles satisfy all these conditions. As a result of the fact that each block of π_A is included in a block of π^*_A: $\pi_A \leq \pi^*_A$. Since all output information of machine A is used as input information for machine B, it can be assumed that $\pi'_A = \pi^*_A$ and condition (1) is then satisfied whenever condition (2) is also satisfied. For condition (2) we need to first calculate the product $\pi_A \cdot \pi_B$ = $\{\overline{0,8};\overline{1,9};\overline{2,10};\overline{3,11};\ \overline{4,12};\overline{5,13};\overline{6,14};\ \overline{7,15}\}$. Using these results, it is easy to see that condition (2) is then satisfied. Finally, for condition (3) we need to calculate $\pi^*_A \cdot \pi^*_B$ = $\{\overline{0,8};\overline{4,12};\overline{1,2,3,9,10,11};\ \overline{5,6,7,13,14,15}\}$. With this product partition, it is obvious that condition (3) is satisfied and hence the decomposition is correct. In Table III, the function tables for the different logic blocks of the implementation of f_2 of f are presented. It is interesting to note that this approach uses different gates: OR, NAND,

EXOR and the gate $\bar{a}b$. All these gates are innately and directly obtained from decomposition without the use of (non-trivial) technology mappings.

3.3 Special Cases of General Full-Decomposition

Today, none of the methods that have been published have been able to produce near optimal solutions for a multiple general full-decomposition. All the published results relate to special cases of the presented model. In this section, a number of special cases will be discussed.

3.3.1 Input-bit parallel full-decomposition

In parallel decomposition, no information flows between the partial machines, and therefore the partitions π'_O and τ'_O in theorem 5 are reduced to $\pi_0(I)$. In input-bit decomposition, the input decoder Ψ is reduced to the appropriate distribution of the input bit lines and this results in the replacement of the input partitions π_I and τ_I by the bit-partitions π_{IB} and τ_{IB}. In this way, the following theorem was obtained from theorem 5.

Theorem 7 A combinational machine M has a non-trivial **input-bit parallel full-decomposition with two component machines** (see Figure 5) if and if only two partition doubles (π_{IB}, π^*_I) and (τ_{IB}, τ^*_I) exist that satisfy the conditions:

(1) $\pi_I \leq \pi^*_I$ and $\tau_I \leq \tau^*_I$, where $\pi_I = \text{ind}(\pi_{IB})$ and $\tau_I = \text{ind}(\tau_{IB})$.

(2) $(\pi^*_I \cdot \tau^*_I, \pi_O(0))$ is an I-O partition pair.

A well-known and extensively studied special case of the input-bit parallel full-decomposition is the input-encoder problem. In this case, the output decoder θ is implemented as a PLA and the input encoders M_i have multiple exclusive sets of input bits. The problem is often modelled using multiple-valued logic [43][44][46]. The concept of multiple-valued logic is in fact very similar to that of partition theory. The general function of two-bit encoders is to replace the inputs a, $\bar{a}$, b and $\bar{b}$ of the PLA with the signals $a + b$, $a + \bar{b}$, $\bar{a} + b$ and $\bar{a} + \bar{b}$. It can be proved that the size of the new PLA (without the size of the two-bit encoders which generate these signals from a and b) cannot be larger than the size of the original PLA. However, the total size (the size of the new PLA and the encoders) can be larger. Fortunately in many practical cases, considerable gain in the total area can be obtained. The major problem with this method is the choice of pairs of inputs. In [43][46] a heuristic algorithm is presented which tries to find sub-optimal pairs of inputs. The drawback of this method is that it ignores interactions between pairs of inputs.

TABLE III
Functions tables for the blocks in Figure 4b

x_3 x_4	A	A x_2	B (E)	x_1 x_4	C	B C	D (f_2)
0 0	0	0 0	0	0 0	1	0 0	0
0 1	1	0 1	1	0 1	1	0 1	1
1 0	1	1 0	1	1 0	1	1 0	0
1 1	1	1 1	0	1 1	0	1 1	0

A similar, but more sophisticated and general way to solve the input-encoder problem was presented by Ciesielski et al. [15][52]. He implements the input encoders using PLAs. The encoders can have any number of input signals and some of the input bits may be directly fed to the output decoder. Characterisation:

- In the first step, a set of inputs is partitioned into a number of disjoint subsets. Two heuristic approaches are presented for these: the first is based on integer programming whereas the second is based on a modified min-cut algorithm.
  Benchmark results from a large and varied set of machines show that the integer programming approach is better, but the graph-partitioning approach is faster.
- A classical multiple-valued minimization is then used to find the best implementation of the PLA Θ.
- The results that have been presented (in the form of benchmarks) are very promising, the only drawback of this approach is that the input sets may not intersect.

Another special case of the input-bit parallel decomposition was considered by W. Wan et al. [51]. Given a certain incompletely specified multiple output function $f(x_1,\ldots,x_n)$, the method presented in [51] searches for two disjoint subsets A and B of all inputs of f (i.e. $A,B \subset \{x_1,\ldots,x_n\}$), a multiple output function F and k single output functions $g_1,\ldots,g_k$ in order that the function F $(g_1(A),g_2(A),\ldots,g_k(A), B)$ realizes the "care" behaviour of f. This approach differs from the input encoder problem presented earlier in this section because the input sets used for the encoder blocks g_k are not required to be mutually disjoint. The decomposition is targeted towards Xilinx FPGAs. The input set A is limited to 4 elements hence the functions g_k have no more than 4 inputs. Because g_k can implement any function of 4 variables with the same cost, the goal of the decomposition is to implement as much functionality as possible into the function g_k and make the function F as simple as possible. If the number of inputs of F is too large, the decomposition algorithm can be iteratively applied to F. Unfortunately, it has been impossible to characterise and evaluate this method more precisely, because further

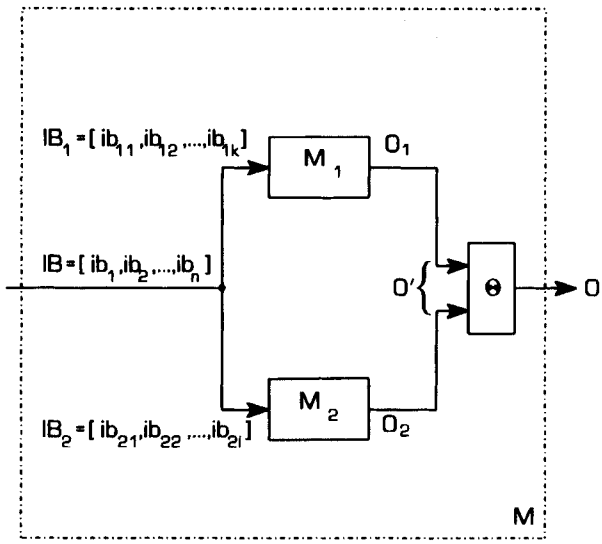

FIGURE 5 Input-bit parallel full-decomposition of M into two component machine M_1 and M_2.

details of the method, other than the outline sketched in [51], were not available to us. However, the benchmark results presented in [51] show that the method performs well compared to previous methods that have been presented for synthesis on Xilinx logic blocks.

Łuba et al. introduced yet another special case (see Figure 6) [31], where one of the component machines (M_2) is replaced by an identity function. The first step of this algorithm is to find the inputs IB_2 which have to be fed directly to output decoder θ. The search algorithm tries to find the best set of inputs, so that the number of inputs of output decoder θ does not exceed a user specified bound (for Xilinx clbs this bound is set to 4). The algorithm then tries to find an implementation for machine M_1 using a minimum number of possible inputs (i.e. $IB_2 \cup IB_3$ contains a minimum number of elements). First a disjoint decomposition is used (i.e IB_3 has no elements). If this fails, inputs from IB_2 are added to IB_3 until machine M_1 can be constructed. Unfortunately, no heuristics are described and no results on large benchmark sets are presented, therefore it is impossible to estimate the efficiency of this method for large circuits. However, the results that are presented are very promising.

In recent years, a number of methods for the more general input-bit parallel decomposition problem have been presented. In [22][23][24], the set of input bits is partitioned in two disjoint subsets. The goal of this method is to minimize the number of bits needed for communication between M_1 and M_2 and output decoder Θ. Although this method does not explicitly use the partition theory it can be easily formulated with this theory. The strength of this method is that it allows the estimation of communication complexity without having to construct the machines M_1 and M_2 and the output decoder Θ. Characterisation:

- Good heuristic solutions for the calculation of communication complexity without the actual need to construct the blocks. This algorithm can have a linear complexity for circuits with low communication complexities.
- Heuristic procedures for the partitioning of the input sets exist.
- Benchmark results on large examples are relatively good.

3.3.2 Bit-parallel full-decomposition

In the bit-parallel full-decomposition, both the input decoder and the output decoder are reduced to the appropriate distribution of the input/output bit lines (see Figure 7). The theorem for this type of decomposition can be obtained from theorem 7 by replacing the output partitions π_O and τ_O with bit-partitions π_{OB} and τ_{OB}.

Theorem 8 A combinational machine has a non-trivial **bit parallel full-decomposition with two component machines** (see Figure 7), if two partition doubles (π_{IB}, π_{OB}) and (τ_{IB}, τ_{OB}) exist that satisfy the conditions:

(1) $\forall_{ob_k \in OB\text{-}dcb(\pi_{OB})} (\pi_I, \pi_O(ob_k))$ are I-O partition pairs, where $\pi_I = ind(\pi_{IB})$.

(2) $\forall_{ob_k \in OB\text{-}dcb(\tau_{OB})} (\tau_I, \tau_O(ob_k))$ are I-O partition pairs, where $\tau_I = ind(\tau_{IB})$.

(3) $\pi_{OB} \cdot \tau_{OB} = \pi_{OB}(0)$.

Solutions to this decomposition problem have been presented independently in [27], [30] and [20]. In [27] the problem is called the output decomposition problem. An output decomposition consists of partitioning the set of Boolean functions (outputs) into a number of disjoint subsets, each implemented by a separate component

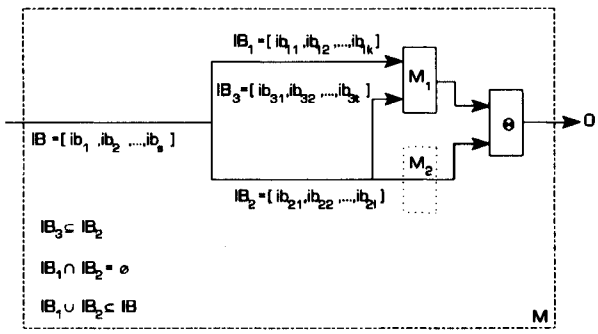

FIGURE 6 Input-bit parallel decomposition as proposed by Łuba et al.

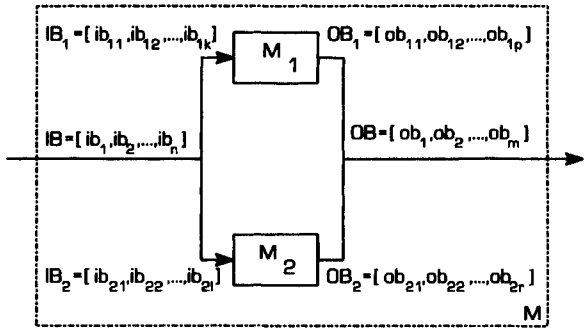

FIGURE 7 Bit-parallel full-decomposition.

machine. The output decomposition in [27] aims at partitioning a multiple-output function into a minimal number of limited programmable logic building blocks (such as PLAs, PALs, etc.) and in minimizing the number of interconnections between the blocks. The problem is modelled as a multi-dimensional constrained optimization problem with constraints imposed on the number of inputs, outputs and terms. It is solved by a special multi-dimensional packing algorithm.

- First, the information processing structure of the original combinational machine and its relation to the characteristics of building blocks are analyzed. From information about the correlations between the input, term and output variables as well as information about the constraints, the expected minimum number of building blocks and the expected number of input bits, output bits and terms per building block are computed. The expected values show how difficult it is to satisfy each of the constraints with a given number of blocks and indicate the amount of attention that must be paid to each of the constraints during the partitioning process. The active input bits and terms for each single-output function are also computed. Based on this information, affinities (from the viewpoint of a certain partitioning problem) between each two (single or multiple-output) functions can be computed.
- With the above information, a limited number of near optimal solutions are constructed in parallel by performing a multi-dimensional packing while using a beam-search algorithm. Since the decision making during the search is based on uncertain information, the search is guided by the heuristic elaborations of the rule of minimizing the uncertainty of choices. At each step, the decisions are taken which ensure the highest certainty of achieving the optimal solutions and, under this condition, the decisions that minimize the uncertainty of information for the future choices. Information that is used directly for decision making consists of relations between the characteristics of single-output functions and constraints imposed by (partially constructed) building blocks and, correlations between the single-output functions and functions in (partially constructed) blocks.
- Published experimental results show that the method is very effective; in almost all cases it was able to find the global optimum in reasonable time even for very complex functions (e.g. a function with 131 inputs, 253 terms and 91 outputs (cpio) or a function with 45 inputs, 428 terms and 43 outputs (apex1)). The search algorithm has a number of parameters which enable a trade-off between the quality of solutions and the required computation time.

A similar method was published few months later in [20]. It uses less information about the original multiple-output function than the method presented in [27] and elaborates information less precisely. An interesting concept not present in the method published in [27] is that of relaxing the term constraints and dynamically processing the terms.

Another approach to bit parallel full-decomposition is presented in [30]. In this paper the problem is referred to as parallel decomposition. Characterisation:

- The actual parallel decomposition is preceded by argument reduction. Argument reduction is a technique which minimizes the number of inputs of a Boolean function, as opposed to the classical minimisation which aims at finding a minimum number of product terms. It is used to find function representations which use minimum number of input variables. This process is similar to term reduction which finds the minimum number of product terms.
- A parallel decomposition algorithm uses the results of the argument reduction and constructs two-block parallel decompositions. Unfortunately in [30], only the idea of parallel decomposition is presented with no algorithms and heuristics. Only few results of experiments are shown however, these are very promising.

In a later paper [32], this decomposition method is combined with the input-bit parallel decomposition method mentioned in the previous section. This already allows for the construction of complex networks of blocks, but it is not yet a general full-decomposition in its most complete form. In this paper, Łuba stated that the bit parallel full-decomposition should be used as a preliminary step to the more general input-bit parallel decomposition procedure. A heuristic is then presented

which shows how these two different decomposition approaches should be alternated and which parameters should be used to tune these algorithms. Some parts of the method use exhaustive or greedy algorithms and the selection of parallel or serial decomposition is empirical. Results presented for quite small circuits show that this algorithm works reasonably well for ACTEL cells and very well for Xilinx cells. The question is, however, whether the algorithm will work effectively and efficiently for complex circuits.

4. COMPARISON OF THE APPROACHES

In the previous two sections, we presented the concepts of division-based and general decomposition-based multiple-level logic synthesis and discussed a number of synthesis algorithms that use these concepts. From this discussion it should be clear that the division-based approach is a very special case of the general decomposition approach, limited to decompositions with partial machines and decoders exclusively implemented with AND, OR and NOT or NAND or NOR gates. Therefore, the methods which are based on division can easily be transformed to equivalent decomposition methods. An example is presented in [35] where division-based synthesis is used to preform a special input-bit parallel decomposition. For special cases of logic implementations in the form of exclusively AND-OR-NOT, NAND or NOR networks, the solutions with division-based synthesis may be appropriate, under the condition that they take into account all the important objectives and constraints and involve effective and efficient algorithms.

The general decomposition approach has a number of advantages over the division-based approach. The main advantage of general full-decomposition is its general character. The general decomposition model and theorem enable modelling and construction of all possible combinational network structures, while the traditional logic synthesis methods, including the division-based methods, model circuits in terms of special minimum or almost minimum functionally complete systems. Such a functionally complete system is able to express each function, but it models functions as networks composed of exclusively special sub-functions which are included in a certain functionally complete system (e.g. AND-OR-NOT, NAND, NOR, EXOR-AND, MUX), while the general decomposition approach models them in terms of all possible sub-functions. If a certain element library includes more types of primitive gates than those included in a certain minimum functionally complete system or includes look-up tables, technology mapping is necessary. The network synthesized using exclusively the gates from the minimal functionally complete system must be mapped into the network composed of any elements from the library. If the repertoire of sub-functions offered by a certain implementation technology differs substantially from the set of gates provided by a given minimal functionally complete system, the work done by a traditional synthesis method is almost futile. Since the initial network is constructed without any regard to future implementation, to guarantee a possible to implement or optimal solution, the technology mapping must again perform synthesis using the previously synthesised network as only a functional specification. Using decomposition-based synthesis this problem does not exist: the synthesis process constructs a network of functional blocks, which are in one-to-one correspondence with physical hardware blocks.

A further advantage of decomposition-based synthesis is its total character. During the decomposition, attention is paid not only to the active elements (operators) but to all the elements and aspects which can influence the quality of the results (i.e. inputs, outputs, interconnects and functionality) and to their interrelations. In the division-based approach all these aspects, except for active elements, are completely ignored. The only exception is the lexicographical factorisation [45], which takes the interconnections into account by accepting a predefined input ordering during synthesis. Of course, it is possible to improve or further develop division-based methods by taking into account all these elements and their relations to the actual objectives and constraints, but it will not enlarge their range of application to circuits substantially different from AND-OR-NOT circuits.

Another important aspect is the use of don't cares in incompletely specified functions. In division-based synthesis, incompletely specified functions are first minimized using a two-level minimizer and then the actual synthesis is performed. In this way all don't cares are removed and hence the design freedom is drastically reduced. The synthesis problem is transformed to this of finding an implementation of a completely specified function (without considering influence of the don't cares on realisation of the actual design). These don't cares are lost for ever. Also, the multiple-level structure itself can introduce don't cares [5][11]. It has been shown that very complex and time-consuming techniques are necessary to effectively use don't cares in division-based synthesis [4]. The decomposition approach does not require prior two-level minimisation and innately uses the freedom given by don't cares in order to optimize the resulting network structure (see for example [31]).

In Table IV, some synthesis results are presented that compare the division based synthesis with the general decomposition based synthesis. The results are taken from [34]. The goal of the experiments was implementation with the minimum number of primitive logic

blocks being 5-inputs, 2-outputs look-up tables. Table IV compares the number of clbs needed to implement a number of benchmark circuits from the MCNC logic synthesis benchmark-set [53]. In the second column (marked Łuba) the results of the general full decomposition method described in [34] are presented. The remaining columns present the results of different division based algorithms. These algorithms use (modified versions of) one of the factorisation algorithms presented in section 2 to minimise the two-level representation, followed by a technology mapping phase. The goal of the technology mapping phase is to transform the AND-OR-NOT network obtained by division into a feasible network with a minimum number of clbs. A feasible network is a network where blocks are only used if they do not violate the constraints imposed on them (in our cases: the blocks can implement any Boolean function that uses no more than 5 different input bits and 2 different output bits). In the third column (labelled MIS-PGA) the results are presented for the method described in [38][39]. This method heavily depends on the classical kernel/co-kernel minimisation and tries to map the minimum network to a feasible structure. In column 4, the results for the same set of benchmarks are presented for the ASYL system [48]. The ASYL system implements, among others, the lexicographical factorisation algorithm. In [48] a very simple but effective modification of the lexicographical factorisation algorithm is presented, and this is able to construct feasible networks. The technology mapper Chortle [18] (column 5 of the table) presents a method for the technology mapping of multiple AND-OR-NOT networks (obtained using any division based synthesis algorithm) into a network of clbs.

As the table shows, the general decomposition approach gives results, which are in all cases better than the division based approaches. These dramatic improvements are obtained because of two major reasons:

- In the general decomposition approach, we deal with partitions representing any function with a certain number of inputs and outputs and not with AND-OR-NOT based functions. Therefore, the synthesis process automatically finds feasible networks using the set of all available primitive logic blocks, without the need of technology mapping.
- The general decomposition approach can handle incomplete specified functions, whereas the division based synthesis uses two-level minimised functions (where the design freedom is removed a priori).

TABLE IV
Experimental results for 5-input, 2-output lookup tables

Benchmark Name	Łuba [35]	MIS-PGA [38]	ASYL [48]	Chortle [19]
rd84	5	10	17	35
rd73	4	6	30	16
misex 1	7	11	13	11
z4	3	5	4	3
5xp1	8	18	24	24
9sym	4	7	8	51

5. CONCLUSIONS

In this paper, we have discussed and compared multiple-level logic synthesis methods based on division of Boolean expressions, and logic synthesis methods based on the theory of general full-decompositions. It is clear that the synthesis based on division can be considered as a special case of a general decomposition-based synthesis. To date, more research work has been done in the field of division-based synthesis, but preliminary results obtained from the general decomposition-based methods show their large potential and have revealed a number of very promising properties. In section 2, we also showed that the known division-based algorithms have a number of weaknesses which can be improved without leaving the division paradigm. In section 4, we showed that division-based synthesis has a number of fundamental weaknesses which require a more general solution, i.e. a general full-decomposition. Recently, substantial progress has been made in developing algorithms for many special cases of the general decomposition. However, a lot of work has still to be performed in order to efficiently exploit the opportunities created by modern microelectronic technology.

ACKNOWLEDGEMENTS

We would like to thank Wil Brant and Ad Chamboné for preparing the figures.

Xilinx is a registered trademark of Xilinx, Inc.

References

[1] P. Abouzeid, K. Sakouti, G. Saucier, F. Poirot: Multilevel synthesis minimizing the routing factor, 27th ACM/IEEE Design Automation Conference, June 1990, pp. 365–368.

[2] P. Abouzeid, G. Saucier, F. Poirot: Lexicographical Factorisation Minimizing the Critical Path and the Routing Factor for Multi-level Logic, in: Logic and Architecture Synthesis, Proceedings of the IFIP TC10/WG 10.5 Workshop, P. Michel and G. Saucier (editors), Elsevier Science Publishers, Amsterdam, 1991, pp. 219–228.

[3] P. Abouzeid, L. Bouchet, K. Sakouti, G. Saucier, P. Sicard: Lexicographical Expression of Boolean Function for Multilevel Synthesis of High Speed Circuits, in: VLSI Logic Synthesis and Design, R.W. Dutton (red.), IOS, Amsterdam, 1991, pp. 31–39.

[4] K.A. Bartlett, R.K. Brayton, G.D. Hachtel, R.M. Jacoby, C.R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang: Multilevel Logic Minimization Using Implicit Don't Cares, IEEE transactions on computer-aided design, vol. 7, no. 6, June 1988, pp. 723–740.

[5] G. Boole: An Investigation of the Laws of Thought, Dover Publications, New York, 1854.

[6] K.S. Brace, R.L. Rudell, R.E. Bryant: Efficient Implementation of a BDD Package, proceedings 27th ACM/IEEE Design Automation Conference, 1990, pp. 40–45.

[7] R.K. Brayton, C. McMullen: The decomposition and factorisation of Boolean expressions, proceedings international symposium on circuits and systems, 1982, pp. 49–54.

[8] R.K. Brayton, J.D. Cohen, G.D. Hachtel, B.M. Trager, D.Y.Y. Yun: Fast recursive boolean function manipulation, proceedings international symposium on circuits and systems, 1982, pp. 58–62.

[9] R.K Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli: Logic Minimization Algorithms for VLSI Synthesis, Kluwer Academic Publishers, Dordrecht, 1984.

[10] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli: MIS: A Multiple-Level Logic Optimization System, IEEE Transactions on computer-aided design, vol. 6, no. 6, november 1987, pp. 1062–1081.

[11] R.K. Brayton, F. Somenzi: Boolean Relations and the Incomplete Specification of Logic Networks, in: VLSI 89, G. Musgave and U. Lauther (editors), Elsevier Science Publishers, North-Holland, 1990, pp. 231–240.

[12] R.K. Brayton, G.D. Hachtel, A.L. Sangiovanni-Vincentelli: Multilevel Logic Synthesis, Proceedings of the IEEE, vol. 78, no. 2, February 1990, pp. 264–300.

[13] R.E. Bryant: Graph-based Algorithms for Boolean Function Manipulation, IEEE Transactions on computers, vol. C-35, no. 8, August 1986, pp. 677–691.

[14] R.E. Bryant: Symbolic Boolean Manipulations with Ordered Binary Decision Diagrams, Carnegie Mellon University, School of Computer Science, CMU report: CMU-CS-92-160.

[15] M.J. Ciesielski, S. Yang: PLADE: A Two-Stage PLA decomposition, IEEE transactions on computer-aided design, vol. 11, no. 8, August 1992, pp. 943–954.

[16] M. Damiani, G. De Micheli: Observability Don't Care Sets and Boolean Relations, IEEE International Conference on Computer-aided Design, November 1990, pp. 502–505.

[17] B.J. Falkowski, I. Schäfer, M.A. Perkowski: Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions, IEEE transactions on computer-aided design, vol. 11, no. 10, October 1992, 1207–1226.

[18] R. Francis, J. Rose, Z. Vranesic: Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs, 28th ACM/IEEE Design Automation Conference, 1991, pp. 227–233.

[19] J. Hartmanis, R.E. Stearns: Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[20] Z. Hasan, D. Harrison, M. Ciesielski: A Fast Partitioning Method for PLA-Based FPGAs, IEEE Design & Test of Computers, September 1992, pp. 34–39.

[21] E.V. Huntington: Sets of Independent Postulates for the Algebra of Logic, Transactions of the American Mathematical Society, vol. 5, 1904, pp. 288–309.

[22] T. Hwang, R.M. Owens, M.J. Irwin: Multi-level logic synthesis using communication complexity, 26th ACM/IEEE Design Automation Conference, 1989, pp. 215–220.

[23] T. Hwang, R.M. Owens, M.J. Irwin: Exploiting Communication Complexity for Multilevel Logic Synthesis, IEEE transactions on computer-aided design, vol. 9, no. 10, October 1990, pp. 1017–1027.

[24] T. Hwang, R.M. Owens, M.J. Irwin: Efficiently Computing Communication Complexity for Multilevel Logic Synthesis, IEEE transactions on computer-aided design, vol. 11, no. 5, October 1992, pp. 545–554.

[25] L. Jóźwiak: The Bit Full-Decomposition of Sequential Machines, EUT Report 89-E-223, Eindhoven University of Technology, The Netherlands, 1989.

[26] L. Jóźwiak, J.C. Kolsteren: An Efficient Method For the Sequential General Decomposition of Sequential Machines, Microprocessing and Microprogramming, North-Holland, vol. 32, 1991, pp. 657–664.

[27] L. Jóźwiak, F. Volf: An Efficient Method for Decomposition of Multiple Output Boolean Functions and Assigned Sequential Machines, EDAC—The European Conference on Design Automation, March 1992, pp. 114–122.

[28] L. Jóźwiak : An Efficient Method for State Assignment of Large Sequential Machines, Journal of Circuits, Systems, and Computers, vol. 2, no. 1, 1992, pp. 1–26.

[29] L. Jóźwiak: General Decomposition and Its Use in Digital Circuit Synthesis, accepted for publication in the special issue of VLSI Design Journal on Decompositions in VLSI Design.

[30] T. Łuba, J. Kalinowski, K. Jasiński: PLATO—a CAD tool for logic synthesis based on decomposition, EDAC—The European Conference on Design Automation, 1991, pp. 61–69.

[31] T. Łuba, J. Kalinowski, K. Jasiński, A. Kraniewski: Combining serial decomposition with topological partitioning for effective multi-level PLA implementations, proceedings IFIP working conference on logic and architectures synthesis, June 1990, in P. Michel, G. Saucier (ed.): Logic and Architecture Synthesis, Elsevier Science Publishers B.V. (North-Holland), 1991, pp. 243–252.

[32] T. Łuba, M. Markowski, B. Zbierzchowski: Logic Decomposition for Programmable Gate Arrays, proceeding EURO ASIC'92, 1992, pp. 19–24.

[33] T. Łuba, H. Selvaraj, A. Kraniewski: A new approach to FPGA-based logic synthesis, Workshop on Design Methodologies for Microelectronics and Signal Processing, October 1993, pp. 20–23.

[34] T. Łuba, H. Selvaraj: A general approach to boolean function decomposition and its application in FPGA-based synthesis, accepted for publication in the special issue of VLSI Design Journal on Decompositions in VLSI Design.

[35] A.A. Malik, D. Harrison, R.K. Brayton: Three-level Decomposition with Application to PLDs, IEEE international conference on computer design, 1991, pp. 628–633.

[36] S. Malik, L. Lavagno, R.K. Brayton, A. Sangiovanni- Vincentelli: Symbolic Minimization of Multilevel Logic and the Input Encoding Problem, IEEE transactions on computer-aided design, vol. 11, no. 7, July 1992, pp. 825–843.

[37] Y. Matsunaga, M. Fujita: Multi-level Optimization Using Binary Decision Diagrams, IEEE international conference on computer-aided design, November 1989, pp. 556–559.

[38] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli: Logic Synthesis for Programmable Gate Arrays, 27th ACM/IEEE, Design Automation Conference, 1990, pp. 620–625.

[39] R. Murgai, N. Shenoy, R.K. Brayton, A. Sangiovanni-Vincentelli: Performance Directed Synthesis for Table Look Up Programmable Gate Arrays, IEEE International Conference on Computer-Aided Design, 1991, pp. 572–575.

[40] S. Muroga, Y. Kambayashi, H.C. Lai, J.N. Culliney: The Transduction Method—Design of Logic Networks Based on Permissible Functions, IEEE transactions on computers, vol. 38, no. 10, October 1989, pp. 1404–1424.

[41] J. Rajski, J. Vasudevamurthy: Testability Preserving Transformations in Multi-level Logic Synthesis, proceedings international test conference, 1990, pp. 265–273.

[42] J. Rajski, J. Vasudevamurthy: The Testability-Preserving Concurrent Decomposition and Factorization of Boolean Expressions, IEEE transactions on computer-aided design, vol. 11, no. 6, June 1992, pp. 778–793.

[43] R.L. Rudell: Multiple-valued logic minimization for PLA synthesis, Ph.D. thesis University of California, Berkeley, memorandum no. UCB/ERL M86/65, 1986.

[44] R.L. Rudell, A. Sangiovanni-Vincentelli: Multiple-Valued Minimization for PLA Optimization, IEEE Transactions on computer-aided design, vol. 6, no. 5, September 1987, pp. 727–750.

[45] G. Saucier, J. Fron, P. Abouzeid: Lexicographical Expressions of Boolean Functions with Application to Multilevel Synthesis, IEEE Transactions on computer aided design of circuits and systems, vol. 12, no. 11, November 1993, pp. 1642–1654.

[46] T. Sasao: Input Variable Assignment and Output Phase Optimization of PLA's, IEEE transactions on computers, vol. c-33, no. 10, October 1984, pp. 879–894.

[47] C.E. Shannon: A Symbolic Analysis of Relay and Switching Circuits, Transactions of the American Institute of Electrical Engineering, vol. 57, 1938, pp. 713–723.

[48] P. Sicard, M. Crastes, K. Sakouti, G. Saucier: Automatic synthesis of boolean functions on Xilinx and Actel Programmable devices, proceedings EURO ASIC'91, May 1991, pp. 142–145.

[49] S. Trimberger: Guest Editor's Introduction: Field-Programmable Gate Arrays, IEEE design & test of computers, special issue on FPGA's, September 1992, pp. 3–5.

[50] J. Vasudevamurthy, J. Rajski: A method for Concurrent Decomposition and Factorization of Boolean Expressions, proceedings international test conference, 1990, pp. 510–513.

[51] W. Wan, M.A. Perkowski: A New Approach to the Decomposition of Incompletely Specified Multi-Output Functions Based on Graph Coloring and Local Transformations and Its Application to FPGA Mapping, EURO-DAC'92—European Design Automation Conference, Hamburg, September, 1992, pp. 230–237.

[52] S. Yang, M.J. Ciesielski: PLA Decomposition with Generalized Decoders, IEEE international conference on computer-aided design, 1989, pp. 312–315.

[53] S. Yang: Logic Synthesis and Optimization Benchmarks User Guide, Version 3.0, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709. This document and the benchmarks can be obtained by anonymous ftp from mcnc.org:/pub/benchmark.

[54] The Programmable Logic Data Book, Xilinx Inc., 1993.

Biographies

FRANK VOLF received his M. Sc. degree in information technology, from the Faculty of Electrical Engineering, Eindhoven University of Technology, the Netherlands, in 1991. He is currently working towards his Ph. D. in the Section of Digital Information Systems at the same university. His Ph. D. work involves research in the field of multiple level logic synthesis. His primary research interests include all aspects of the automatic synthesis and verification of digital circuits, compiler techniques and the design of operating systems. He is a co-author of several scientific papers.

LECH JÓŹWIAK received his M. Sc. and Ph. D. degrees in Electronics from the Warsaw University of Technology, Poland, in 1976 and 1982, respectively. From 1976 to 1979, he worked at the Faculty of Electronics of this University. From 1979 to 1986, he was a chief of the project team in the Research Institute of Computers in Warsaw. From 1986 till now, he is an associate professor in the Department of Digital Information Systems, Eindhoven University of Technology, the Netherlands. His research interests include design theory, circuit and system theory, design theory and artificial intelligence in design and correctness verification. He is an author of numerous research reports and papers.

MARIO P.J. STEVENS received his M. Sc. degree in telecommunication from the Faculty of Electrical Engineering, Eindhoven University of Technology, the Netherlands, in 1971. From 1971 to 1986, he worked as a researcher and lecturer at this Faculty. From 1986 he is a professor at the same Faculty and chairman of the Section of Digital Information Systems. His scientific interests include computer architecture, design methodologies for digital systems, data communication and system software. He is an author of a number of papers and books and advisor in microelectronics and information technology at the Dutch Ministry of Economic Affairs.