LOGIC SYNTHESIS

Synthesis of digital circuits consists of a series of steps involving translation, optimization, and mapping. In general, a design is described at different levels of abstraction using hardware description languages like VHSIC hardware description language (VHDL) or Verilog. Design descriptions at each level are usually optimized based on area, timing, and power dissipation measures to generate a design description at the following lower level. In this article we consider design description and optimization at the logic gate level. For sequential circuits, we assume that a higher level of description, such as the state-transition graph, is available. Our aim is to survey various logic synthesis techniques that target different optimization criteria, like area, timing, and power consumption. It is beyond the scope of this article to delve into the details of logic synthesis algorithms. For details of the algorithms, the readers are referred to the papers and books referenced here.

This article is organized as follows. In the first section we describe two-level logic optimization techniques. Two-level logic usually consists of an AND gate level and an OR gate level. The logic description can be in a product-of-sums or a sum-of-products form (such as in programmable logic arrays). It turns out that logic implemented in multiple levels (rather than two) can be more area efficient. The next section describes multilevel logic optimization algorithms that consider area, timing, and power dissipation. After logic optimization, the design is usually mapped into a target library. Such mapping techniques are called technology mapping and are briefly described in this section. The previous sections consider only combinational circuits. Because sequential circuits can have feedback and use memory elements, like flip-flops or latches,

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

the synthesis techniques differ. Sequential circuits can be represented by state-transition diagrams. In the following section we consider the state assignment and the sequential circuit synthesis problem. The synthesis techniques described in the previous sections are suitable for application-specific integrated circuits (ASIC) or gate array implementations. Recently, a new style of design called the field-programmable gate arrays (FPGAs) has become very popular. FPGAs usually consist of rows of logic modules (which can implement different types of logic gates) with (re)programmable routing architectures. Such logic styles require a different logic synthesis style. The next section considers the basics of logic optimization for FPGA's. Finally conclusions are given in the last section.

TWO-LEVEL LOGIC SYNTHESIS

By definition, two-level logic circuits are logic circuits with two levels of logic gates. A typical technology that uses two-level logic circuits is programmable logic arrays (PLAs). Usually NOR–NOR or NAND–NAND arrays are used. For example, logic function $y = ((x_1 + x_2)' + (x_3 + x_4)')'$ is a NOR–NOR implementation. Here x_i' (or $\overline{x}_i$) denotes the complement of x_i. In addition to PLAs, two-level logic design style is adopted because of its speed advantage. One of the factors that determine circuit delays is the number of stages (levels) through which a signal goes. Therefore, two-level logic circuits are fast compared to multilevel logic circuits (are introduced later) and may be chosen at the cost of area (using more and/or bigger gates to implement the circuit). Another reason that we are interested in two-level logic is that it represents a component of a multilevel logic network. If we can simplify the representation, it simplifies the multilevel logic optimization. For simplicity, AND–OR representation is assumed in this section, that is, sums of products like $z = x_1x_2 + x_3x_4$ are our focus in this discussion. Because by DeMorgan's laws (1) NOR–NOR or NAND–NAND expressions can be rewritten as sums of products, the assumption does not lose generality. For example, $y = [(x_1 + x_2)' + (x_3 + x_4)']'$ mentioned earlier can be rewritten as $y' = x_1'x_2' + x_3'x_4'$ by DeMorgan's laws (1).

Given a Boolean function represented as a sum of products, we want to know how to implement it with minimum area. Most of the literature uses a minimum number of product terms as the cost function to be optimized. This is to simplify the problem and to separate the optimization from the specific technology that implements the function. However, it may not correspond to the minimum area. To understand how different optimization algorithms work, we have to introduce some key concepts.

Basic Concepts

A Boolean function f can be expressed as a sum of products of n literals. These product terms are called minterms. For example, $w = x_1x_2 + x_1x_3$ can be rewritten as $x_1x_2x_3' + x_1x_2x_3 + x_1x_2'x_3$ with three minterms of three literals. An implicant of a Boolean function f is a product term that is contained in f. An implicant p is said to be contained in f if $p = 1$ implies $f = 1$, which is denoted as $p \subseteq f$. Similarly, an implicant p_1 is said to be contained in another implicant p_2, denoted as $p_1 \subseteq p_2$, if $p_1 = 1$ implies $p_2 = 1$. For example, $f = x_1x_2 + x_1x_3$ has implicants x_1x_2 and x_1x_3. Function f contains both x_1x_2 and x_1x_3, and neither x_1x_2 contains x_1x_3 nor does x_1x_3 contain x_1x_2. In addition, $x_1x_2x_3$ is also an implicant of f and is contained in x_1x_2 and in x_1x_3. A prime implicant of f is an implicant of f that is not contained in any other implicant of f, that is, x_1x_2 and x_1x_3 are two prime implicants of f whereas $x_1x_2x_3$ is not.

A cover C_f of a Boolean function f is a set of implicants that contains all the minterms of f, and f contains C_f. A cover is said to be prime if all the implicants of the cover are prime. Sometimes f may have don't care conditions that specify that the result of f is not of concern for certain inputs. In this case, if don't care conditions are denoted by DC_f,

$$f \subseteq C_f \subseteq f \cup DC_f \quad (1)$$

A cover can contain some implicants that are don't care conditions. A minimum cover of a Boolean function f is a cover of f of a minimum number of implicants. In contrast, an irredundant (or minimal) cover is a cover such that no subset of the cover can be a cover of f. If any implicant is taken away from an irredundant cover, it is no longer a cover.

Given the definitions of covers, the two-level logic optimization problem becomes equivalent to finding a minimum cover of a Boolean function. There are two approaches to solving this problem, exact and heuristic. Because solving this problem exactly may not be feasible for large circuits, a heuristic approach is usually taken. Though a heuristic approach may yield a suboptimal solution, it often gives a minimum solution.

Exact and Heuristic Solutions

Among all the minimum covers, there is always a minimum cover that is prime. This was proved by Quine (2) and allows us to limit our search space and to find a minimum cover among all the prime covers. In addition, we can make prime every nonprime implicant of a minimum cover. This is done by replacing the nonprime implicant by a prime implicant that contains it. For example, suppose that $abcde$ is a nonprime implicant and is in a minimum cover C_f. By definition of prime implicants, a prime implicant, say abd, must exist that contains $abcde$. Then abd can replace $abcde$. Usually the area cost for abd is less than that of $abcde$, and therefore prime minimum covers give solutions of smaller area than that of nonprime minimum covers.

Karnaugh map and Quine–McCluskey methods are systematic procedures to simplify two-level logic functions (1). The Karnaugh map method is useful for simplifying functions of two to four variables. It can be extended to handle functions of five and six variables. For example, let $y = a'b' + a'b + ab'c$. Figure 1 shows the Karnaugh map of function y. From the map we conclude that it has two prime implicant and a prime minimum cover is $\{a', b'c\}$. Notice that the dotted squares or rectangles are prime implicants. The Karnaugh map method can be seen as an attempt to find all the prime implicants and a prime minimum cover out of these prime implicants.

The Quine–McCluskey method finds all prime implicants first and builds a prime implicant chart to determine prime minimum covers. For example, let

$$\begin{aligned} z &= ab + ab + a'c' \\ &= abc + abc' + ab'c + a'bc' + a'b'c' \end{aligned}$$

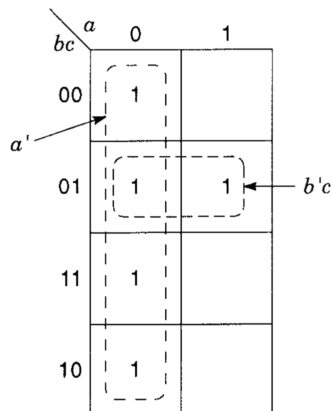

Figure 1. Karnaugh Map of a logic function: It is used to simplify two-level logic functions.

Prime implicants

Minterms	p_1	p_2	p_3	p_4
1	1			
2	1	1		
3			1	
4		1		1
5			1	1

Figure 3. Prime implicant chart of a function: It helps find a minimum cover.

The optimization procedure is carried out in Fig. 2. The table on the left-hand side is a list of minterms and the one on the left is a list of four prime implicants, p_1, p_2, p_3, and p_4. p_1 (0-0 meaning ac) is derived by combining minterm 1 and 2, and p_2 (-10 meaning bc') is the result of combining minterm 2 and 4. The rest are shown as in the figure. Figure 3 shows the prime implicant chart. It is shown that minterm 1 ($a'b'c'$) is covered only by $p1$ and minterm 3 ($ab'c$) is covered only by p_3. Such prime implicants are called essential prime implicants and must be chosen for the minimum prime covers. As a result, minterms 1, 2, 3, and 5 are covered whereas minterm 4 abc' is not. In this case, it is very easy to see that two minimum prime covers exist, $\{p_1, p_3, p_2\}$ and $\{p_1, p_3, p_4\}$. However, when the number of variables and prime implicants increases, computational time and memory used for searching a minimum prime cover may be exponential in the worst case.

Researchers improved the Quine–McCluskey method by building a smaller prime implicant chart and by applying an efficient branch-and-bound algorithm to search for the minimum prime covers (3). Further and other improvements have been done by Refs. 4–7. However, for most practical cases, the run time of using exact solutions may not be tolerable. This motivates heuristic approaches. A heuristic minimizer ESPRESSO (3) gives minimal (not minimum) prime cover. ESPRESSO builds a prime cover and uses an iterative improvement strategy to modify and to delete implicants. Often it gives solutions close to minimum covers.

	abc
1	000
2	010
3	101
4	110
5	111

1,2	0–0	p_1
2,4	–10	p_2
3,5	1–1	p_3
4,5	11–	p_4

Figure 2. Quine-McCluskey method of a function: It is used to simplify two-level logic functions.

MULTILEVEL LOGIC SYNTHESIS

Optimal multilevel logic synthesis is a known difficult problem which has been studied since the 1950s. Unlike two-level logic optimization, the exact optimization methods for multilevel logic are not practical for today's circuit design because of their computational complexity. Therefore, we focus on heuristic optimization methods in our discussion. The first of the modern developments is the logic synthesis system (LSS) at IBM (9), which has a variety of gate arrays and standard cells as target technology. More recent work in multilevel logic optimization includes MIS (10) and BOLD (11). Both MIS and BOLD are aimed at optimization techniques which are independent of particular technologies and were developed to bring multilevel logic optimization to the level of science obtained for two-level logic optimization. Because the results are independent of particular technologies, there is one more step, called technology mapping, that needs to be taken. Technology mapping maps the logic functions to the gates of a particular technology before it can be implemented in a very large scale integration (VLSI) circuit. First we review some basic concepts before introducing the optimization techniques behind MIS and BOLD. Then optimization techniques based on different algebra models (algebraic and Boolean models) are discussed, followed by a brief discussion of other approaches and technology mapping.

Basic Concepts

The *on-set* of a function of f is the set of minterms for which the function evaluates to 1. The *off-set* of the function of f is the set of minterms for which f equals 0. The don't care set or *dc-set* is the set of minterms for which the value of the function is unspecified. A function which does not have a *dc-set* is a *completely specified* function. A function with a nonempty *dc-set* is termed an *incompletely-specified* function.

Sum of Product Form. A *literal* is a Boolean variable (say, x) or its complement (x' or $\bar{x}$). A *cube* can be defined as a product of literals, for example, xyz'. The trivial cubes, written as 0 and 1, represent the Boolean functions 0 and 1, respectively. A sum-of-products (SOP) form (also called *expression*) is a set of cubes. For example, expression $a + b\bar{c}$ consists of two cubes, a and bc'.

An expression is *algebraic* (nonredundant) if no cube in the expression properly contains another. For example, $a + ab$ is redundant because a contains ab. But expression $a + bc$ is algebraic. A Boolean expression is a nonredundant expression.

The *union* of two expressions f and g, denoted as $f \cup g$, is the set that consists of all the cubes of f and g and is transformed into a nonredundant expression. Similarly, the *intersection* of f and g, denoted as $f \cap g$, is the set of all the common cubes of f and g.

Factored Form. The usual representation of a logic function is the sum-of-products form. An alternative representation to this is the factored form. It is the generalization of the sum-of-products form allowing nested parentheses. For example, the expression $ace + ade + bce + bde + \bar{e}$ can be written in factored form as $e(a + b)(c + d) + \bar{e}$. In other words, a factored form is a sum of products of arbitrary depth. Generally the factored form is not unique. For example. the expression $abc + abd + cd$ is itself a factored form but can also be written as $ab(c + d) + cd$ or $abc + (ab + c)d$. Both are factored forms.

In many applications, it is infeasible to describe each single-output function of a multiple-output function as a single expression or a single factored form. Often, a set of intermediate functions is introduced, and each depends on the original inputs and possibly other intermediate functions. Then, each single-output function can be expressed as a function of original inputs and the intermediate functions. For example, the multiple-output function

$$F_1 = [(a+b)c+d]e+f$$
$$F_2 = [(a+b)c+d]g+h$$

can be expressed as the following set of functions involving variables x and y:

$$F_1 = ye+f$$
$$F_2 = yg+h$$
$$y = xc+d$$
$$x = a+b$$

Multi-level logic refers to any multiple-output Boolean function represented by a set of interconnected functions. Therefore, multilevel logic is a particular representation of multiple-output functions.

A multiple-level logic function can be graphically represented as a directed acyclic graph (DAG) (V, E), where V and E are the set of all vertices and edges in the graph, respectively.

The set of input nodes V_{IN} does not have any incoming edges, and the set of output nodes V_{OUT} does not have any outgoing edges. The set of *intermediate* nodes is given by $V_{INT} = V - (V_{IN} \cup V_{OUT})$. Each node $v_i \in V_{INT}$, computes a Boolean function F_i in terms of its fan-in nodes and is also associated with a "local output" variable y_i, where $y_i = F_i$.

The desired design style affects the synthesis and optimization methods. Indeed the search for an interconnection of logic gates that optimize area and/or performance depends on the constraints on the choice of the gates themselves. Multiple-level logic is usually partitioned into two tasks. First, a logic network is optimized while neglecting the implementation constraints on the logic gates and assuming loose models for their area and performance. Second, the constraints on the usable gates (e.g., those represented in the cell library) are taken into account.

Logic Synthesis Using the Algebraic Model

A logic network can be optimized by using the general properties of polynomial algebra to simplify the Boolean model. The simplification includes the assumption that the exponent of every variable in the network is at most one in the polynomial algebra and ignores don't care conditions. Such an assumption simplifies the problem but may yield a less than optimal solution.

Division. Given functions f and p, we can find functions q and r such that $(f = pq + r)$. Every such operation is like the division operation and is therefore called *(algebraic) division* of f by p generating *quotient* q and *remainder* r. The function p is called a divisor of f if r is not null and a factor if r is null.

Optimization can be carried out if good algebraic divisors can be found. The set of algebraic divisors is defined as $D(f) = (g|\, f/g \neq \emptyset)$.

The *primary divisor* of f is defined as $P(f) = [(f/c|c$ is a cube]. That means the primary divisors of an expression f are the expressions f/c where c is a cube.

For example, if

$$f = abc + abde$$

then

$$\frac{f}{a} = bc + bde$$

is a primary divisor.

The *kernels* of f are defined as $K(f) = (k|k \in P(f)$, k is cube-free). An expression is cube-free if it does not have a cube factor. In other words, the kernels of an expression f are the cube-free primary divisors of the expression.

The cube c used to obtain the kernel $k = f/c$ is called the *co-kernel* of f. Continuing with the previous example, f/a is a primary divisor but not cube-free because b is a factor of f/a. Therefore,

$$\frac{f}{a} = b(c+de)$$

However,

$$\frac{f}{(ab)} = c+de$$

is a kernel. ab is called a co-kernel. Note that by definition, co-kernels are always cubes.

Example. Given the expression

$$X = abcdg + abcdh + abce + abcf + abi$$

then

$$\frac{X}{a} = bcdg + bcdh + bce + bcf + bi$$

is a *primary divisor,* but is not a *kernel* because the *expression* is not *cube-free* (cube b divides each term). However,

$$\frac{X}{ab} = cdg + cdh + ce + cf + i$$

and

$$\frac{X}{abcd} = g + h$$

are both *kernels* with associated *co-kernels* ab and $abcd$.

Because no single cube is cube-free, a kernel must contain two or more cubes. Also, because 1 is a cube, if f is cube-free, then f is considered one of its own kernels.

The *level* of a kernel is defined to provide easily identifiable subsets of the set of all kernels. Recall that kernels are expressions, and hence it makes sense to refer to the kernels of a kernel. A kernel is called a *level-0 kernel* of f if it does not have any kernels except itself. No literal appears twice in a level-0 kernel. A kernel is called a *level-n kernel* if it contains a kernel of level $(n - 1)$ but does not contain any kernels of level-n except itself. This gives us a natural partition of the kernels:

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

Example

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

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

The motivation for this definition of the kernels of a logic expression comes from the following theorem which was used in MIS (10):

Theorem 1. f and g have a common multicube divisor if and only if there exists $k_f \in K(f)$, and $k_g \in K(g)$ such that $|k_f \cap k_g| \geq 2$.

That is, two functions have a common multiple-cube divisor if and only if the intersection of a kernel from f and a kernel from g has more than one cube. It is important to remember that an expression is a set of cubes and the intersection of kernels refers to the set intersection of the expressions and not the Boolean intersection of the logic functions implied by the expressions.

The computation of the kernel set of the expression in the logic network is the first step toward the extraction of multiple-cube expressions. Then the candidate common subexpressions to be extracted are chosen among the kernel intersections.

Example. Consider a network with the following expressions:

$$\begin{aligned} f_x &= ace + bce + de + g \\ f_y &= ad + bd + cde + ge \\ f_z &= abc \\ K(f_x) &= [(ac + bc + d), (a+b), (ace + bce + de + g)] \\ K(f_y) &= [(a+b+ce), (cd+g), (ad+bd+cde+ge)] \end{aligned}$$

The kernel set of f_z is empty.

Hence, multiple-cube common subexpressions can be extracted only from f_x and f_y. There is only one kernel intersection, namely, $(a + b) \in K(f_x)$, and $(a + b + ce) \in K(f_y)$. The intersection is $a + b$ and can be extracted to yield

$$\begin{aligned} f_w &= a + b \\ f_x &= wce + de + g \\ f_y &= wd + cde + ge \\ f_z &= abc \end{aligned}$$

Logic Transformation

The goal of multilevel logic optimization is to obtain a representation of the Boolean function that is optimal with respect to area, speed, testability, and power dissipation. To restructure a logic function, the operations described following are used.

Decomposition. The decomposition of a Boolean function is the process of reexpressing a single function as a collection of new functions. For example, the process of translating the expression

$$F = abc + abd + \overline{a}\overline{c}\overline{d} + \overline{b}\overline{c}\overline{d}$$

to the set of expressions

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

is decomposition, shown in Fig. 4. The associated optimization problem is to find a decomposition with minimum total area or power.

Extraction. Extraction is applied to many functions. It is the process of identifying and creating some intermediate functions and variables and reexpressing the original functions in terms of the original and the intermediate variables. Extraction creates nodes with multiple fan-outs. For example, extraction applied to the following three functions:

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

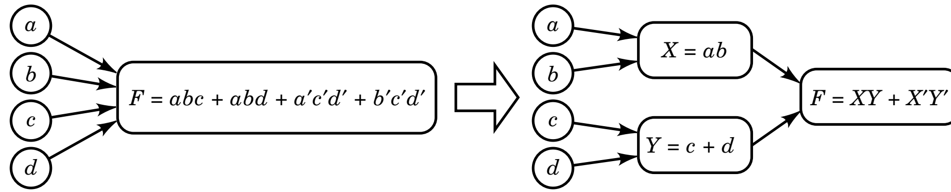

Figure 4. Decomposition decomposes large function into smaller ones.

yields

$$F = XY + e$$
$$G = X\overline{e}$$
$$H = Ye$$
$$X = a + b$$
$$Y = cd$$

Thus the operation identifies common subexpressions among different logic functions that form a network. This is shown in Fig. 5. New nodes corresponding to the common subfunctions are created, and each of the logic functions in the original network is simplified as a result of introducing these new nodes. The optimization operation is to find a set of intermediate functions such that the resulting network has minimum area, low power dissipation, delay, or maximum testability.

Factoring. Factoring is the process of deriving a factored form from a sum-of-products form of a function. For example,

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

can be factored to

$$F = (a + b)(c + d) + e$$

The optimization problem associated with factoring is to find a factored form with the minimum number of literals.

Substitution. Substitution, also resubstitution, of a function G into F is the process of reexpressing F as a function of its original inputs and G. Let us consider the example of Fig. 6. Substituting

$$G = a + b$$

into

$$F = (a + b)(a + c)$$

produces

$$F = G(a + c)$$

This operation creates an arc (the wider line in Fig. 6) in the Boolean network that connects the node of the substituting function, namely, G, to the node of the function being substituted in, namely, F.

Elimination. Elimination, collapsing, or flattening is the inverse operation of substitution. If G is a fan-in of F, collapsing G into F reexpresses F without G. It undoes the operation of substituting G in F. For example, if

$$F = Ga + \overline{G}b$$
$$G = c + d$$

then, collapsing G into F results in

$$F = ac + ad + b\overline{c}\overline{d}$$
$$G = c + d$$

This is illustrated in Fig. 7.

Logic Synthesis Using Don't Cares

Logic networks can also be efficiently synthesized using the concepts of don't cares. Such logic networks obtained by using this optimization is much more testable than merely 100% testable for conventional input and output stuck-at faults. The approach is based on determining the complete don't-care set for each two-level function embedded in a network of such functions. Once this is done, a two-level minimizer can be used to minimize the subfunction. Before we describe the don't care-based logic optimization technique let us consider some definitions.

Let $F(y_1, y_2, \ldots y_n)$ be a Boolean expression. The *cofactor* of F with respect to y_k is given as $(F)_{y_k}$. $(F)_{y_k}$ is obtained by setting y_k to '1' in the expression of F. The cofactor of F with

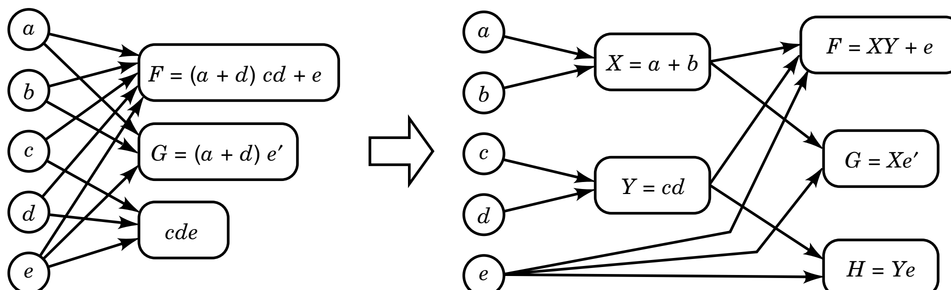

Figure 5. Extraction extracts common expressions from functions.

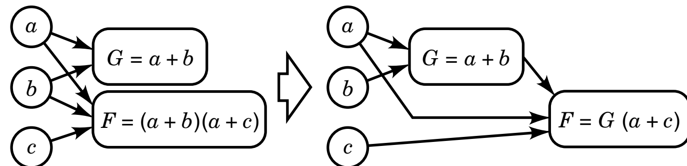

Figure 6. Substitution substitutes one function into another.

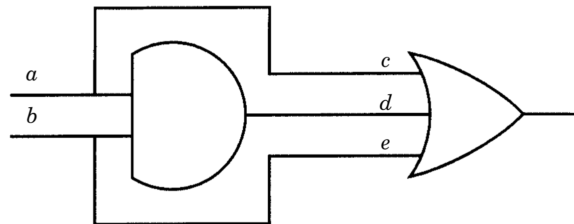

Figure 8. Example showing the intermediate variable don't care set: $d = ab$, $c = a$, and $e = b$ and therefore $d = ce$. Since $d \neq ce$ will never happen, it is don't care and should be utilized for logic optimization.

respect to $\overline{y_k}$, $(F)_{\overline{y_k}}$, is defined similarly. Using Shannon's theorem, F can be represented as

$$F = y_k (F)_{y_k} + \overline{y_k} (F)_{\overline{y_k}} \tag{2}$$

Given a multilevel circuit graph, a set of input vectors exists, specified by the user, which can never occur. These set of inputs are called the *external don't care* set of inputs. For all our discussions, we assume that this set is $\emptyset$.

Note that Boolean functions F_i representing node v_i, when considered in isolation, are completely specified functions. However, when embedded in a Boolean network, F_i is incompletely specified, that is, has a don't-care set. Such redundancies can be generated by using the structure of the Boolean network. In Ref. 8 two sets of don't cares, *intermediate variable don't care set,* DIV (common to all nodes) and *transitive fan-out don't care set,* DT (which is specific to node j), are defined. Then logic optimization is based on such don't cares obtained from the multilevel Boolean network.

The overall *intermediate variable don't care set* is defined by

$$\text{DIV} = \sum_{j=1}^{m} \text{DIV}_j$$

where $\text{DIV}_j = y_j\overline{F_j} + \overline{y}_jF_j$, representing the EXOR function. DIV_j can be best understood by considering the example of Fig. 8. It is clear from the figure that certain set of inputs, such as $d \neq (c.e)$, can never occur at the inputs to the OR gate. The set of such inputs are the *intermediate variable don't care set.*

The set of input vectors for which node i is insensitive to the values of node j (i is in the transitive fan-out of node j and is a primary output) is called the *transitive fan-out don't care set of node j*, DT_j.

$$\text{DT}_j = \bigcap_{i \in FO_j} E_{ij}$$

where

$$E_{ij} = ((F_i)_{y_j} \equiv (F_i)_{\overline{y_j}}) \tag{3}$$

$$= (F_i)_{y_j}(F_i)_{\overline{y_j}} + \overline{(F_i)_{y_j}}\ \overline{(F_i)_{\overline{y_j}}} \tag{4}$$

and FO_j represent the transitive fan-out of j. Let us consider a simple two-input (a and b are inputs whereas c is an output) AND gate. The output c is insensitive to input b when an input of ($a = 0$ and $b = 1$) is applied to the circuit.

Now each node of the Boolean network can be optimized using a two-level optimizer with the don't care sets as described above. However, note that the don't care sets for each node can be large. Hence, researchers have tried to optimize don't care sets using heuristics, such as don't care filters. Such filters tradeoff computational time versus optimization quality.

Other Optimization Algorithms

There are other interesting approaches in addition to the one just discussed. One is to use circuit redundancy to simplify the network (12,13). For example, let a node vi in a logic network F be untestable for stuck-at-0, that is, if F' denotes the new logic network by forcing vi to logic 0, and F and F' are equivalent. If v_i is an input to a NAND gate, the NAND gate can be replaced by a logic value 1. If v_i is an input to a NOR gate, v_i can be eliminated from the inputs of the NOR gate. Transduction (14), global flow (15), and rule-based systems (9) are some of the most significant techniques.

The logic synthesis techniques described previously are geared mainly toward area and performance optimization. We described area optimization techniques in detail. However, performance can be optimized by reducing the fan-out (achieved using logic duplication), by optimizing the number of inverters required (standard CMOS efficiently implement only inverting logic), and by reducing the logic depth during synthesis. More recently, because of the increased number of devices per chip and the proliferation of battery-operated components coupled with the fact that frequency has doubled every two years, power dissipation is becoming a major con-

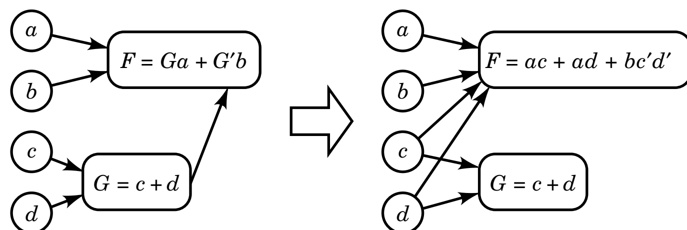

Figure 7. Elimination eliminates one particular function representation from a function. That is, it flattens the original function.

cern. Average power dissipation in a standard CMOS circuit is given approximately by

$$P_{\text{avg}} = V_{\text{dd}}^2 \sum_{i \in \text{gates}} C_i a_i + I_{\text{sckt}}.V_{\text{dd}} + I_{\text{leak}}.V_{\text{dd}}$$

where V_{dd} is the supply voltage, C_i is the capacitance associated with the output of a logic gate, and a_i represents the average number of signal switching at node i. I_{sckt} and I_{leak} represent the short circuit and the leakage currents of the design, respectively. The first component results from the switching current and is by far the dominant (more than 85% in current technology in the active mode of operation) and hence, let us concentrate on the switching component of power. $\Sigma_{i \in \text{gates}}\ C_i a_i$ represents the switched capacitance per unit time. Estimating a_i is difficult because it depends on the primary input signal distribution. It has been shown that the inputs to a logic circuit can be represented as stochastic processes that have certain properties: signal probability (the probability that a signal is logic ONE) and signal activity (the probability of signal switching). Probabilistic and statistical methods (18,19) have been developed to determine the signal activity at the internal nodes of a logic circuit. It should also be noted that in CMOS circuits, C_i is proportional to the fanout of the logic gate. Hence, during logic synthesis, one can try to optimize power dissipation by properly selecting common subexpressions which would reduce the overall switched capacitance based on the given input signal probability and activity (31). However, it should be observed that power conscious technology mapping techniques are also important since the logic network derived after the logic optimization phase is modified during technology mapping (23).

Technology Mapping

In the multilevel logic synthesis described previously, optimization is a technology-independent process. Logic network F generated by the multilevel logic synthesis has to be mapped to real circuits that satisfy timing and area constraints. The real circuits usually consist of a set of predefined gates called library L, and the gates are called library cells. Library cells are technology-dependent. To simplify the mapping from F to library cells of L, two different approaches have been taken—tree-based and Boolean-matching approaches.

Tree-based Matching and Covering. In this approach, logic network F and all library cells are decomposed into a set of base functions B. The set of base functions is usually small. For example, it can be 2-input NAND gate and inverter. The decomposed F corresponds to a graph, called subject graph. Similarly, each decomposed library cell corresponds to a pattern graph. In the subject graph and pattern graphs, each node corresponds to a base function. However, there may be more than one way to decompose F and library cells. Therefore, the final result depends on how we decompose F. In addition to decomposition, the subject graph is partitioned into trees (16). For a node vi in a tree, if a pattern graph is isomorphic to a subtree rooted in vi, the pattern graph matches v_i.

For example, let us consider Fig. 9. Assume that the set of base functions B consists of a 2-input NAND gate and an inverter and that library cells are 2-input NAND and NOR gates and an inverter. Logic network $F = abcd$ is decomposed into three NAND gates and three inverters, which is a subject

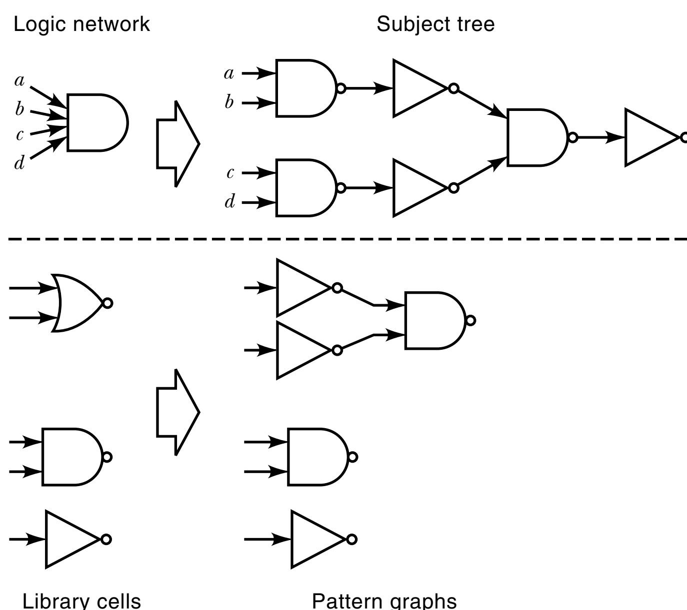

Figure 9. Decomposition of logic network and library cells: Technology mapping first decomposes library cells into pattern graphs and the logic network into a subject tree.

tree. The library cells are also decomposed into base functions. In Fig. 10, the subtrees of the subject tree are matched by some pattern graphs and therefore are mapped into the corresponding library cells.

Assume that the cost of pattern graph does not depend on the parent of v_i. For example, the cost function can be the area of pattern graphs. A bottom-up dynamic programming algorithm can be performed from the leaves to the root to compute the optimum tree covering. However, if delay is the cost function, the cost of a pattern graph depends on the parent of the root of the subtree. An approximate solution to this was proposed in Refs. 17 and 20 by storing a piecewise function at every node.

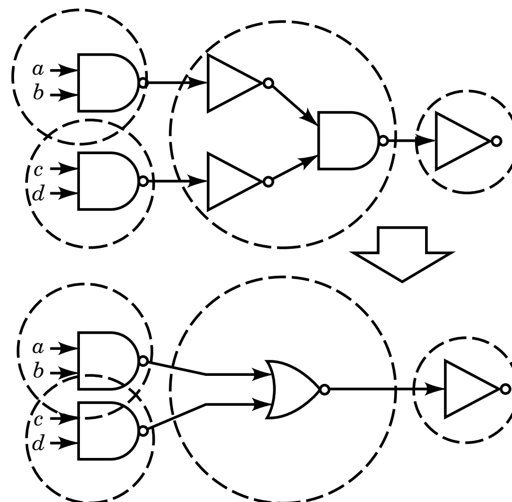

Figure 10. Tree-based technology mapping: Subtrees are matched by pattern graphs and are mapped into corresponding library cells.

Boolean Matching and Covering. In the Boolean-matching approach (21,22) the logic network F is still decomposed into a subject graph. However, the subject graph is partitioned into rooted directed acyclic graphs (DAGs) rather than trees. In addition, the library cell's functions are checked against rooted subgraphs of rooted DAGs by equivalence checking. Equivalence between two Boolean functions can be checked by using ordered binary decision diagrams (OBDDs).

Technology Mapping for Low Power and Deep Submicron Circuits. In addition to area and performance (delays) as cost functions or optimization constraints, power dissipation has become another design constraint. The key to reducing power dissipation is to hide nodes with higher switching activity (23). Another new design concern is the change of delay and area models of library cells. In today's deep submicron circuits, interconnects contribute significantly to delays and die area. Therefore, when optimizing delays and area, we need to take layout into account (24).

SEQUENTIAL CIRCUIT SYNTHESIS

A sequential circuit can be specified by a set of registers (latches) and a logic network, which is a clocked sequential network or synchronous logic network. They can also be described by a more abstract model—finite-state machine (FSM). We first focus on the FSM model.

A finite state machine is defined by a sextuple (I, S, δ, s_0, O, λ), where I, O, S, and s_0 are primary inputs, primary outputs, a set of states, and an initial or reset state ($s_0 \in S$). The next state function δ takes a current state and primary inputs as input and gives a next state. Therefore, $\delta : S \times I \rightarrow S$. The output function λ takes either a state for a Moore model ($\lambda : S \rightarrow 0$) or a state and primary inputs for a Mealy model ($\lambda : S \times I \rightarrow 0$). We concentrate on the Mealy model because it is more general and the techniques that are introduced later can also be applied to Moore machines. Figure 11 shows a Mealy-clocked sequential network. There are two more concise but equivalent ways to describe a FSM rather than the sex-tuple. They are the state-transition table and the state

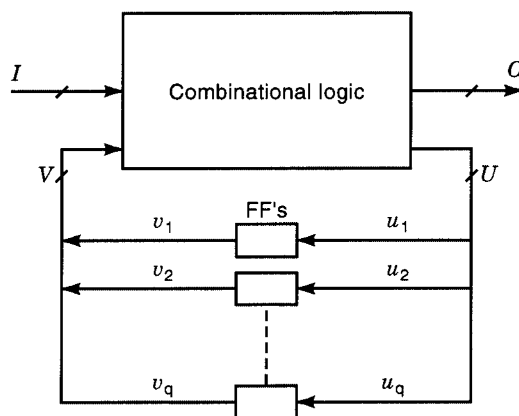

Figure 11. State machine representation: Output and states are functions of input and the previous states.

I	PS	NS	0
I1	S_0	S_1	01
I2	S_0	S_2	02
I3	S_1	S_0	03
I4	S_1	S_2	04
I5	S_2	S_0	05
I6	S_2	S_2	06

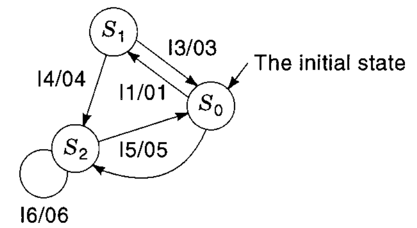

Figure 12. State transition table (left) and graph are two representations of finite state machine.

transition graph (STG) shown in Fig. 12. Each node in the STG represents a state, and the edges are the transition function δ. An FSM is incompletely specified if the next state function and/or the outputs are not specified for some combinations of inputs and present states. Otherwise, it is completely specified. Usually, the less the states are, the smaller the circuit area is. Therefore, it is desirable to reduce the number of states (state minimization), if possible, before some binary numbers are assigned to states (state assignment).

State Minimization

State minimization takes slightly different forms for completely and incompletely specified FSMs. For convenience, we denote the output sequence $OS(IS, s_i)$ assuming that the FSM is initialized in state s_i and an input sequence IS is given. For a completely specified FSM, two states s_i and s_j are equivalent if the output sequences $OS(IS, s_i)$ and $OS(IS, s_j)$ are the same for any input sequence IS. It can be shown that s_i and s_j are equivalent if and only if, for any input I (not input sequence IS), they have the same output and have equivalent next states for the same input (25). This suggests an iterative procedure to find equivalent classes within ns steps and thus obtain the minimum number of states. Number ns is the number of states before minimization. For incompletely specified FSM, a pairwise compatibility is defined (25). However, the computational complexity of finding a minimum is too high, and often times heuristic algorithms are applied instead to obtain near-optimal solutions.

State Assignment

State assignment or state encoding is the process of assigning binary numbers to states of an FSM. Because the number of bits used to encode states is related to circuit complexity a minimum number of state bits is preferred. State assignment for two-level logic circuits can be thought of as symbolic minimization (26,27) whereas present states and next states are input symbols and output symbols on the state-transition table. However, because s_i may appear as present and next states, a constraint must be added to make sure that s_i gets the same encoding for serving as input and output symbols. State assignment for multilevel logic circuits uses multilevel logic network as the combinational logic circuits shown in Fig. 11. Recall from the third section that many transformations optimize the network. A heuristic method was proposed (28–

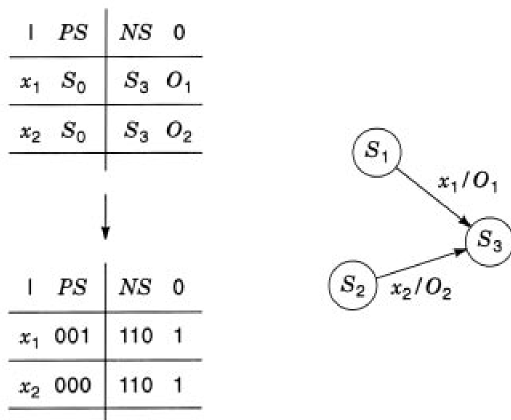

Figure 13. Fanout-oriented encoding tries to maximize the size of common cubes.

30) to consider common cube extraction, that is, state assignment is done so that larger and/or more common cubes can be extracted.

For example, assume that $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$ and 3-bit (minimum number) encoding is used. s_1 and s_2 both have a transition to s_3 when inputs are x_1 and x_2 respectively, as shown in Fig. 13. We encoded s_1 (001) and s_2 (000) so that their Hamming distance is short. The Hamming distance between two states is the number of 1's in s_1 XOR (bitwise) s_2. Let the encoded transition functions be f_1, f_2, f_3 and the state bits be a, b, c. Then f_1 has two cubes $x_1a'b'c$ and $x_2a'b'c$. Therefore, f_1 has a common cube $a'b'$ because

$$x_1a'b'c + x_2a'b'c' = a'b'(x_1c + x_2c') \tag{5}$$

Similarly, f_2 and the output have the same common cube $a'b'$. Keeping the Hamming distance short between states s_i and s_j is called fan-out oriented encoding if they have a transition to the same state. It tries to maximize the size of common cubes.

Figure 14 shows an example of fan-in oriented encoding. The idea is to keep the Hamming distance short between states s_i and s_j with an incoming transition from the same state. Let us consider Fig. 8. We encoded s_1 (111), s_2 (110), and s_3 (001) so that the distance between s_1 and s_2 is short. It tries to maximize the number of common cubes. The common cube $a'b'c$ appears five times.

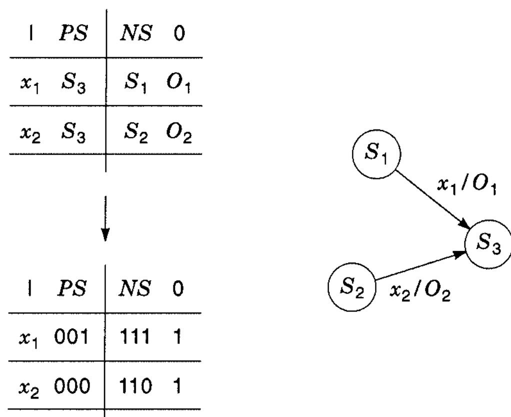

Figure 14. Fanin-oriented encoding tries to maximize the number of common cubes.

FSM and combinational logic synthesis have been conventionally targeted to reducing area and critical path delay (10,26,28). However, power dissipation during the logic synthesis process is being considered only recently. The synthesis process consists of two parts: state assignment, which determines the combinational logic function, and multilevel optimization of the combinational logic, which tries to minimize area while at the same time trying to reduce the sum over all circuit nodes of the product of the circuit activity at a node and the capacitance at the node.

The state assignment scheme in (31) considers the *likelihood of state transitions*—the probability of a state transition (say, from state S_1 to state S_2) when the primary input signal probabilities (probability that an input is equal to logic ONE) are given. The state assignment minimizes the total number of transitions occurring at the V inputs (or the present state inputs) of the state machine shown in Fig. 11. It can be observed that scaled-down supply voltage technologies can still be applied after logic synthesis to reduce power dissipation further. The multilevel logic optimization process is iterative. During each iteration, the best subexpression from among all promising common subexpressions is selected. The objective function is based on both area and power savings. The selected subexpression is factored out of all affected expressions.

LOGIC SYNTHESIS FOR FPGAs

Field-programmable gate arrays (FPGAs) combine the flexibility of mask programmable gate arrays with the convenience of field programmability. This can be achieved by channelled gate array architecture consisting of rows of configurable logic modules interspersed with programmable routing tracks (32,33). In such an architecture, a logic module can be configured to implement different functions by connecting one or more of its inputs to logical 0 (GND) or logical 1 (VCC) or by shorting them together. This connectivity is achieved by programming the appropriate programmable connections (PCs).

Figure 15 shows the row-based FPGA architecture (32,33). Each logic module consists of an interconnection of a set of logic gates and can implement different logic functions. Two possible logic block architectures are multiplexor-based or lookup-table-based. The multiplexor-based architectures use logic blocks that combine a number of multiplexors and some AND or OR gates. Lookup-table-based logic blocks can implement any logic functions with no more than a certain number of inputs (35). A possible logic synthesis and technology mapping technique would exhaustively identify all possible logic functions that a logic module can implement by tying inputs of the logic module to ZERO or ONE and others to signals. The number of unique functions for logic modules can be large (>700 for Actel Act2, which is multiplexor-based). The number is usually much larger in lookup-table-based blocks. A library with these many combinational functions is difficult to manage. The function list can be reduced by qualifying the functions against an existing gate array or FPGA library. The number of functions reduced from 766 to 115 for Act2 by us-

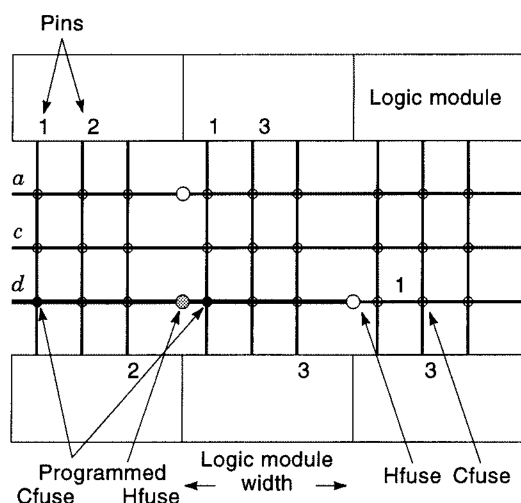

Figure 15. FPGA architecture can configure logic modules to implement different functions.

ing this scheme (34). The reduced number of functions, used after logic synthesis for technology mapping, produced excellent results on benchmark circuits. Hence, such logic synthesis and technology mapping requires minimal changes in the logic synthesis and technology mapping algorithms described earlier for gate arrays and ASIC's.

Let us consider the lookup-table-based FPGA architecture. A direct mapping technique can also be used for mapping logic functions into FPGAs. We assume that logic synthesis or an optimization step has already been performed. The logic network is decomposed into a forest of trees. The network does not have to be represented as a set of 2-input NAND gates and inverters because there is no explicit library. Each tree is optimally mapped into the lookup table by using dynamic programming (36). Note that the structure of the logic function is not important for mapping, but what is important is the number of inputs because a lookup table can implement any logic function up to a certain number of inputs. Hence, a modified technology mapping can efficiently map a logic network directly into a lookup-table-based architecture.

SUMMARY

In recent years automatic logic synthesis and technology mapping have been used very successfully to synthesize random logic and control circuitry for optimizing area, timing, and power dissipation. In this article we have surveyed combinational and sequential logic synthesis and technology mapping techniques. We also presented a modified technology mapping technique for FPGAs. Unfortunately the details of the algorithms were omitted, but readers should find the references useful for further studies.

BIBLIOGRAPHY

1. F. Hill and G. Peterson, *Introduction to Switching Theory and Logical Design,* 3rd ed., New York: Wiley, 1981.
2. W. Quine, The problem of simplifying truth functions, *Amer. Math. Mon.,* **59**: 521–531, 1952.
3. R. Rudell and A. Sangiovanni-Vincentelli, Multiplied-value minimization for PLA optimization, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.,* **CAD-6**: 727–750, 1987.
4. O. Coudert and J. Madre, Implicit and incremental computation of primes and essential primes of Boolean functions, *Proc. Design Autom. Conf.,* 1992, pp. 36–39.
5. O. Coudert, J. Madre, and H. Fraisse, A new viewpoint on two-level logic minimization, *Proc. Design Autom. Conf.,* 1993, pp. 625–630.
6. M. Dagenais, V. Agarwal, and N. Rumin, McBOOLE: A new procedure for exact logic minimization, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-5**: 229–232, 1986.
7. P. McGeer et al., ESPRESSO-SIGNATURES: A new exact minimizer for logic functions, *Proc. Design Autom. Conf.,* 1993, pp. 618–621.
8. K. Bartlett et al., Multilevel logic minimization using implicit don't cares, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst. IC's,* **CAD-7** (6): 723–740, 1988.
9. J. Darringer et al., LSS: A system for production logic synthesis, *IBM J. Res. Dev.,* **28** (5): 537–545, 1984.
10. R. Brayton et al., Multiple-level logic optimization system, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-6**: 1062–1081, 1987.
11. K. A. Bartlett et al., Multilevel logic minimization using implicit don't cares, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-7**: 723–740, 1988.
12. D. Bryan, F. Brglez, and R. Lisanke, Redundancy identification and removal, *Int. Workshop Logic Synthesis,* 1991.
13. S. C. Chang et al., Layout driven logic synthesis for FPGAs, *Proc. Design Autom. Conf.,* 1994, pp. 308–313.
14. S. Muroga et al., The transduction method-design of logic networks based on permissible functions, *IEEE Trans. Comput.,* **C-38**: 1404–1424, 1989.
15. L. Berman and L. Trevillyan, Global flow optimization in automatic logic design, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-10**: 557–564, 1991.
16. K. Keutzer, DAGON: Technology binding and local optimization by DAG matching, *Proc. Design Autom. Conf.,* 1987, pp. 341–347.
17. R. Rudell, Logic Synthesis for VLSI Design, Ph.D. Thesis, Univ. California, Berkeley, 1989.
18. F. Najm, A survey of power estimation techniques in VLSI circuits, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.,* **VLSI-2** (4): 446–455, 1994.
19. T.-L. Chou and K. Roy, Accurate Estimation of Power Dissipation in CMOS Sequential Circuits, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.,* **VLSI-4** (3): 369–380, 1996.
20. H. Touati, *Performance oriented technology mapping,* Ph.D. Thesis, Univ. California, Berkeley, 1990.
21. F. Mailhot and G. De Micheli, Technology mapping with Boolean matching, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-12**: 559–620, 1993.
22. C. R. Morrison, R. M. Jacoby, and G. D. Hachtel, Techmap: Technology mapping with delay and area optimization, in G. Saucier and P. M. McLellan (eds.), *Logic and Architecture Synthesis for Silicon Compilers,* Amsterdam, The Netherlands: North-Holland, 1989, pp. 53–64.
23. M. Pedram, Power minimization in IC design: Principles and applications, *ACM Trans. Des. Autom. Electron. Syst.,* **1** (1): 3–56, 1996.
24. M. Pedram, N. Bhat, and E. S. Kuh, Combining technology mapping and layout, The VLSI Design: *An Int. J. Custom-Chip Des., Simulation Test.,* **5** (2): 111–124, 1997.
25. F. Hill and G. Peterson, *Switching Theory and Logical Design,* New York: Wiley, 1981.

26. G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli, Optimal state assignment for finite state machines, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-4**: 269–284, 1985.
27. G. De Micheli, Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-5**: 597–616, 1986.
28. S. Devadas et al., MUSTANG: State assignment of finite-state machines targeting multi-level logic implementations, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-7**: 1290–1300, 1988.
29. X. Du et al., MUSE: A MUltilevel Symbolic Encoding algorithm for state assignment, *IEEE Trans. Comput.-Aided Des. Integr. Circuit Syst.,* **CAD-10**: 28–38, 1991.
30. B. Lin and A. R. Newton, Synthesis of multiple level logic from symbolic high-level description language, *Proc. IFIP Int. Conf. VLSI,* 1989, pp. 187–196.
31. K. Roy and S. Prasad, Circuit Activity Based Logic Synthesis for Low Power Reliable Operations, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.,* **VLSI-1** (4): 503–513, 1993.
32. A. E. Gammal et al., An architecture for electrically configurable gate array, *IEEE J. Solid State Circuits,* **24** (2): 394–398,1989.
33. J. Birkner et al., A very high speed field programmable gate array using metal to metal antifuse programming elements, *IEEE Custom Integr. Circuits Conf.,* May 1991, pp. 1.7.1–1.7.6.
34. C.-H. Shaw et al., An FPGA architecture evaluation framework, *ACM Workshop FPGAs,* 1992, pp. 15–20.
35. H.-C. Hsieh et al., Third generation architecture boosts speed and density of field programmable gate arrays, *IEEE Custom Integr. Circuits Conf.,* 1990, pp. 31.2.1–31.2.7.
36. R. Francis, J. Rose, and K. Chung, A technology mapping program for lookup table based field programmable gate arrays, *27th Design Autom. Conf.,* 1990, pp. 613–619.

TAN-LI CHOU
Intel Corporation
KAUSHIK ROY
Purdue University

LOGIC, TEMPORAL. See TEMPORAL LOGIC.