IEICE TRANS. INF. & SYST., VOL. E76-D, NO. 9 SEPTEMBER 1993

INVITED PAPER *Special Issue on Synthesis and Verification of Hardware Design*

Optimization of Sequential Synchronous Digital Circuits Using Structural Models

Giovanni De MICHELI†, *Nonmember*

SUMMARY We present algorithms for the optimization of *sequential synchronous digital circuits* using structural models, i.e. interconnections of combinational logic gates and synchronous registers. This approach contrasts traditional methods using state diagrams or transition tables and leveraging state minimization and encoding techniques. In particular, we model circuits by *synchronous logic networks*, that are weighted multigraphs representing interconnections of gates implementing scalar combinational functions. With this modeling style, *area* and *path delays* are explicit and their variation is easy to compute when circuit transformations are applied. Sequential logic optimization may target *cycle-time* or *area minimization*, possibly under area or cycle-time constraints. Optimization is performed by a sequence of transformations, directed to the desired goal. This paper describes the fundamental mechanisms for transformations applicable to sequential circuits. We review first *retiming* and *peripheral retiming* techniques. The former method optimizes the position of the registers, while the latter repositions the registers to enlarge maximally the combinational region where combinational restructuring algorithms can be applied. We consider then *synchronous algebraic* and *Boolean transformations*, that blend combinational transformations with local retiming. Both classes of transformations require the representation of circuits by means of logic expressions with labeled variables, the labels representing discrete time-points. Algebraic transformations entail manipulation of time-labeled expressions with algebraic techniques. Boolean transformations exploit the properties of Boolean algebra and benefit from the knowledge of *don't care* conditions in the search for the best implementation of local functions. Expressing *don't care* conditions for sequential circuits is harder than for combinational circuits, because of the interaction of variables with different time labels. In addition, the feasibility of replacing a local function with another one may not always be verified by checking for the inclusion of the induced perturbation in local explicit *don't care* set. Indeed, the behavior of sequential circuits, that can be described appropriately by the relation between input and output traces, may require relational models to express *don't care* conditions. We describe a general formalism for sequential optimization by Boolean transformations, where the *don't care* conditions are expressed implicitly by *synchronous recurrence equations*. We present then an optimization method for this model, that can exploit degrees of freedom in optimization not possible for other methods, and hence providing solutions of possible superior quality. We conclude by summarizing the major features and limitations of optimization methods using structural models.

key words: *computer hardware and design, synchronous circuits, CAD, logic synthesis*

1. Introduction

Computer-aided design (CAD) tools are routinely used for microelectronic design. In particular, *logic synthesis* and *optimization* techniques have been applied with success to product-level design of digital circuits in the last decade. The dominant use of *semicustom* design styles, based on libraries of cells (e.g. gate-arrays and standard-cells), has motivated the developments of digital design techniques for multiple-level logic circuits. The goals of circuit synthesis and optimization is to minimize the circuit delay (or area) under area (or delay) constraints, as well as to maximize circuit testability. The optimization problems are highly complex to solve, because the related decision problems are intractable. Thus heuristic methods are commonly used.

Combinational logic design problems are well understood, even though better solution methods are continuously searched for [3]. On the other hand, sequential circuit optimization is still in a state of evolution. Even though good methods are known, they do not always perform satisfactorily on some types of circuits. A better understanding of design methods for sequential circuits is thus very important for the development of efficient CAD synthesis tools and for their widespread acceptance by circuit designers.

We consider in this paper *synchronous* circuits, because most digital designs have synchronous operation. Synchronous logic circuits are interconnections of combinational logic gates and registers with synchronous clocking. Feedback connections are restricted to be through synchronous registers, to guarantee race-free design. Synchronous circuits can be modeled by the interconnection of their components, thus using a *structural* view. Alternatively, the state transitions can be described in terms of tables or diagrams. We refer to this modeling style as *behavioral view* of synchronous circuits.

Most research on sequential circuit optimization has used state-based representations, i.e. behavioral views. In this case, synthesis and optimization relate to solving classical problems, such as state minimization, state assignment and decomposition [1], [16]. There

Manuscript received January 12, 1993.

† The author is with the Center for Integrated Systems, Stanford University, Stanford, CA 94305, USA.

are two major drawbacks of this approach. First, the *state* information is *explicit*, but the *area* and *path delay* information is *implicit* in the state-based representation. Thus, figures of merit for optimizing state-based representations, such as state-count or encoding-length, are not directly correlated to the actual objectives (e.g. area and/or performance). In particular, predictions of performance improvement are difficult to achieve when using state-based models. Second, it is not possible to take advantage of the original circuit structure by iteratively improving a netlist-based circuit specification.

Recently, methods have been suggested for sequential circuit optimization that use structural models and stepwise refinement of initial circuits. The optimization paradigm parallels combinational multiple-level logic design, where circuit transformations are iteratively applied to circuits, until the objectives are met or no feasible transformation is found. With structural models, the *area* and *path delay* information is *explicit*, while the *state* information is now *implicit*.

Synthesis and optimization methods for sequential circuits using structural models can complement those methods that exploit behavioral, state-based *finite-state machine* models. Advantages and disadvantages exist in both approaches. Thus it is important for CAD synthesis systems to incorporate both.

In this paper, we survey optimization methods for sequential multiple-level digital circuits, represented by structural models. We assume that the original circuit, that we would like to optimize by stepwise refinement, is either a schematic designed by a human, or a netlist synthesized by high-level synthesis tools or by classical sequential synthesis programs.

We survey first the state of the art in this field, by reviewing previous work, such as *retiming* [13] and *peripheral retiming* [14]. We consider then circuit transformations for synchronous logic networks, with algebraic and Boolean flavor. We present explicit and implicit models for representing the *don't care* conditions in synchronous circuits and for exploiting their degrees of freedom in sequential optimization methods.

2. Synchronous Sequential Circuit Structural Model

We consider structural models of digital synchronous circuits. Such circuits can be specified by interconnections of combinational logic gates and clocked registers. The interconnection is arbitrary but no direct combinational feedback is allowed, to preserve the synchronous property. For the sake of simplicity, we assume that all registers are driven by one clock (i.e. single-phase circuits) and that latching is always positive (or always negative) edgetriggered. Circuit designs using master-slave registers fall in this class. We assume, again for the sake of simplicity, that clock skew, register setup, hold and propagation times are negligible.

We model synchronous circuits by *synchronous logic networks*. A synchronous logic network is described in terms of *labeled Boolean variables* and *Boolean equations*. Each Boolean variable corresponds to either a primary input/output of the circuit or to the output of a combinational logic gate. Boolean variables are labeled, the label being an integer that represents a discrete point on the time axis. The notions of literal, sum and/or product of literals, as well as function and relation over time-labeled variables, are straightforward extensions of the usual concepts in switching theory. In this paper, labels are represented as parenthesized superscripts. We denote the Boolean expressions associated with variables by capital calligraphic letters.

Example 2.1: A synchronous circuit is shown in Fig. 1(a). The circuit decodes an incoming data stream coded with bi-phase marks, as produced by a compact-disk player. The corresponding synchronous logic network is described by the following set of equations:

$$a^{(n)} = i^{(n)} \oplus i^{(n-1)}$$

$$b^{(n)} = i^{(n-1)} \oplus i^{(n-2)}$$

$$c^{(n)} = a^{(n)} b^{(n)}$$

$$d^{(n)} = c^{(n)} + d^{(n-1)}$$

$$e^{(n)} = d^{(n)} e^{(n-1)} + d'^{(n)} b'^{(n)}$$

$$v^{(n)} = c^{(n)}$$

$$s^{(n)} = e^{(n-1)}$$

The right-hand side of the first expression is denoted as $\mathcal{A}$, the second by $\mathcal{B}$, et cetera. □

The network is modeled by the *synchronous*

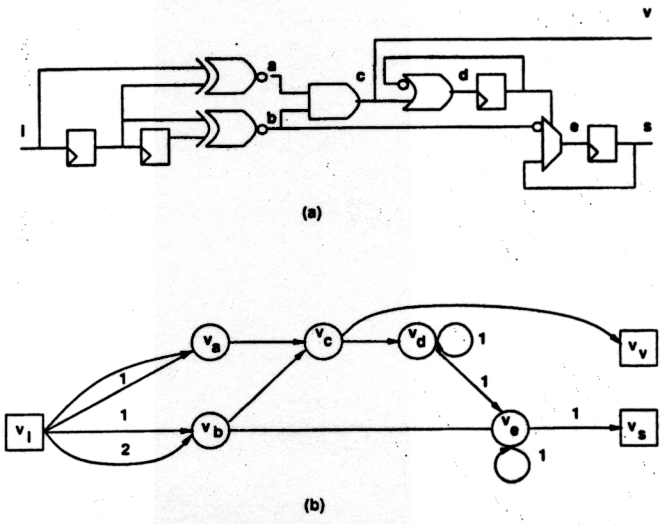

Fig. 1 (a) Synchronous circuit. (b) Synchronous logic network.

network graph, that is a directed weighted multi-graph $G(V, E, W)$, whose vertex set $V = V^I \cup V^G \cup V^O$ is partitioned into input, internal and output vertices that are in one-to-one correspondence with the variables corresponding to the set of primary inputs, logic gates and primary outputs respectively. The subscript of a vertex is the corresponding variable. For example, we denote the vertex related to variable i as vertex v_i.

When considering the set of Boolean equations, the dependency relation of the support variables induces the edge set E of the graph, and the *synchronous delay offsets*, i.e. difference in time labels, the corresponding edge weight set W. With this model, a direct connection between two logic gates corresponds to an edge with zero weight, while a connection through a register to a unit weight. A connection through a k-stage shift register has weight k. Zero weights are sometimes omitted in the graphical representations. The graph is a multi-graph, because the inputs to a logic gate may be signals with different synchronous delay offsets, and thus be modeled by a vertex that is head of multiple edges with different weights. For completeness, there is also a (weighted) edge to each output vertex in V^O from the internal vertex in V^G corresponding to the logic gate generating that output signal.

Example 2.2: The synchronous network graph corresponding to the network of Example 1(a) is shown in Fig. 1(b). There are two edges between v_i and v_a with zero and unit weight. Note that zero-weights are not shown. There are two unit-weighted cycles. □

For each pair of vertices joined by a path in $G(V, E, W)$, the *path weight* is the sum of the weights along the path. We assume that each cycle (i.e. closed path) has strictly positive weight, to model the restriction of breaking combinational logic cycles by at least one register.

In general, a synchronous logic network may have cyclic dependencies, i.e. its corresponding graph be cyclic. A network is called *definite*, or *unidirectional*, when the graph $G(V, E, W)$ is acyclic. A network is called *pipeline* when it is definite and all path weights from any input to any output vertex are equal. Note that the combinational logic network (without synchronous registers) introduced by Brayton [5] is just a special case of the synchronous logic network that is definite and whose labels are all zeroes.

We associate a *propagation delay* with each gate and corresponding vertex. For each pair of vertices joined by a path in $G(V, E, W)$, the *path delay* is the sum of the propagation delays along the path.

3. Algorithms for Sequential Optimization

There are different approaches to optimizing synchronous networks using structural models. The simplest is to ignore the registers and to concentrate on the combinational component, using techniques of combinational logic synthesis [3]. This is equivalent to deleting the edges with strictly positive weights, and to optimizing the corresponding combinational logic network. Needless to say, the removal of the registers from the network segments the circuit and weakens the optimality.

A radically different approach is *retiming*. By retiming a network, we move the position of the registers only. Hence we do not change the graph topology, but we modify the weight set W. Leiserson and Saxe [13] presented polynomially-bound algorithms for finding the optimum retiming, that minimizes the circuit cycle-time or area. Unfortunately, retiming may not lead to the best implementation, because only register movement is considered.

The most general approach to synchronous logic optimization is to perform network transformations that blend retiming with combinational transformations. Such transformations can have the algebraic or Boolean flavor. In the latter case, the concept of *don't care* conditions must be extended to synchronous networks.

We review first previous work by presenting retiming and a recent extension, called peripheral retiming. Then we survey recent results on synchronous logic transformations using algebraic and Boolean techniques. We consider last the specification of *don't care* conditions for synchronous networks, and optimization methods with both explicit and implicit representations of *don't care* conditions.

We concentrate in this paper on the *fundamental mechanisms* for circuit transformations. The transformations themselves can be driven by an overall performance-oriented (or area-oriented) optimization algorithm, that selects the transformation type and the targets. We refer the reader to references [2], [5], [10], [12], [15] for details and examples of overall optimization strategies.

3.1 Retiming

The original *retiming* algorithm was presented first by Leiserson and Saxe. A later paper [13] encompasses all their major contributions to the solution of the problem. The circuit cycle-time (or area) can be minimized by moving the register position, subject to area (or cycle-time) constraints. We review here cycle-time minimization only. While *retiming* a circuit, the combinational component is not modified. Therefore optimizing the cycle-time by retiming is orthogonal to performance-optimization by combinational logic speed-up [17].

Leiserson showed that the minimum clock period corresponds to some path delay between a pair of vertices of the synchronous logic network. Therefore the search for an optimum cycle-time can be reduced to

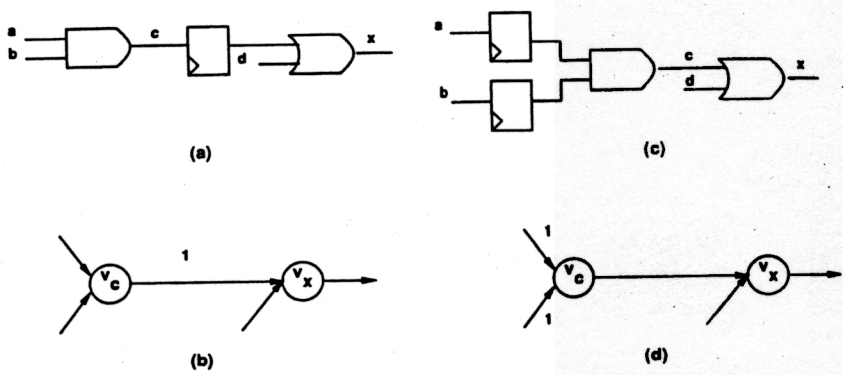

Fig. 2 (a) Circuit fragment. (b) Network fragment. (c) Circuit fragment after retiming. (d) Network fragment after retiming. (Zero weights are omitted.)

verifying whether a retimed circuit can operate at a given clock rate.

To formalize the retiming problem, we associate a retiming vector to the network, whose entries are in one-to-one correspondence with the vertices. Each element represents the amount of register units moved from the outputs of the corresponding gate to its inputs. (Negative entries represent the opposite register movement). The retiming of the corresponding variable is equivalent to adding the retiming entry to its label. The retiming of an algebraic expression the retiming of all its variables. We denote the retiming of a variable or an expression by an integer k by the operator $R^k(\cdot)$. The weight on each edge is increased by the retiming of its head and decreased by the retiming of its tail.

Example 3.1: Consider the circuit fragment shown in Fig. 2(a), whose network is shown in Fig. 2(b). A retiming of vertex v_c by 1 leads to the circuit fragment of Fig. 2(c), whose network is shown in Fig. 2(d). Note that the fragment can be expressed by the equations: $x^{(n)}=c^{(n-1)}d^{(n)}$; $c^{(n)}=a^{(n)}b^{(n)}$ before retiming and $x^{(n)}=c^{(n)}d^{(n)}$; $c^{(n)}=a^{(n-1)}b^{(n-1)}$ after retiming. Thus retiming vertex v_c by 1 corresponds to adding 1 to the label of variable c. (Note that $c^{(n+1)}=a^{(n)}b^{(n)}$ is equivalent to $c^{(n)}=a^{(n-1)}b^{(n-1)}$.) □

A necessary condition for a valid retiming is that the weights of all the edges remain non-negative. A second necessary condition is that the weight on any path whose propagation delay exceeds the cycle-time must be at least one. This condition is equivalent to stating that in any valid implementation the delay of any path with zero registers (weight) is bounded from above by the cycle-time.

Both conditions can be represented by linear inequalities in terms of the entries of the retiming vector. Therefore, the retiming decision problem can be cast into solving a set of inequalities, or equivalently checking for positive cycles in a representative graphs. Retiming can be solved exactly by the Bellman-Ford Algorithms in $O(|V|^3)$ time [13], or by more efficient relaxation schemes that can exploit the sparsity of the network. It can also be cast as a mixed integer-linear program and solved by the simplex algorithm.

Despite the fact that retiming guarantees a global minimum cycle-time, it has not been used much by logic synthesis systems. The main reasons are the following. Retiming was conceived for communication networks and not for logic networks. The assumption of constant gate delay is invalidated by the fact that gate fanouts change as registers move. At times, performance-optimal implementations require adding several registers and may have unacceptable area. Note that when wiring is taken into account, excessive area correlates to degraded performance. Moreover, retiming neglects the possibility of restructuring the combinational component of the network, which is the source of the propagation delay.

Extensions to retiming have been proposed, such as retiming for multiple-phase circuits [4].

3.2 Peripheral Retiming

Peripheral retiming is a novel technique introduced by Malik et al. [14] to leverage combinational logic synthesis as much as possible in sequential circuit design. Peripheral retiming can be thought of as defining a boundary (periphery) around a circuit and extracting all registers from the region inside the boundary. This allows the designer to apply combinational logic techniques to the region (e.g. the combinational speed-up algorithm [17]) and to return later the registers to that region so that the optimized circuit is equivalent to the original one.

In order to extract the registers from a region, the corresponding network must be definite. Cyclic networks can be cut by removing feedback edges. Peripheral retiming can be applied in correspondence with different cuts [14], using a *sliding window* paradigm. An important degree of freedom in peripheral retiming is that edges crossing the periphery can have temporarily negative weights. This corresponds to borrow some

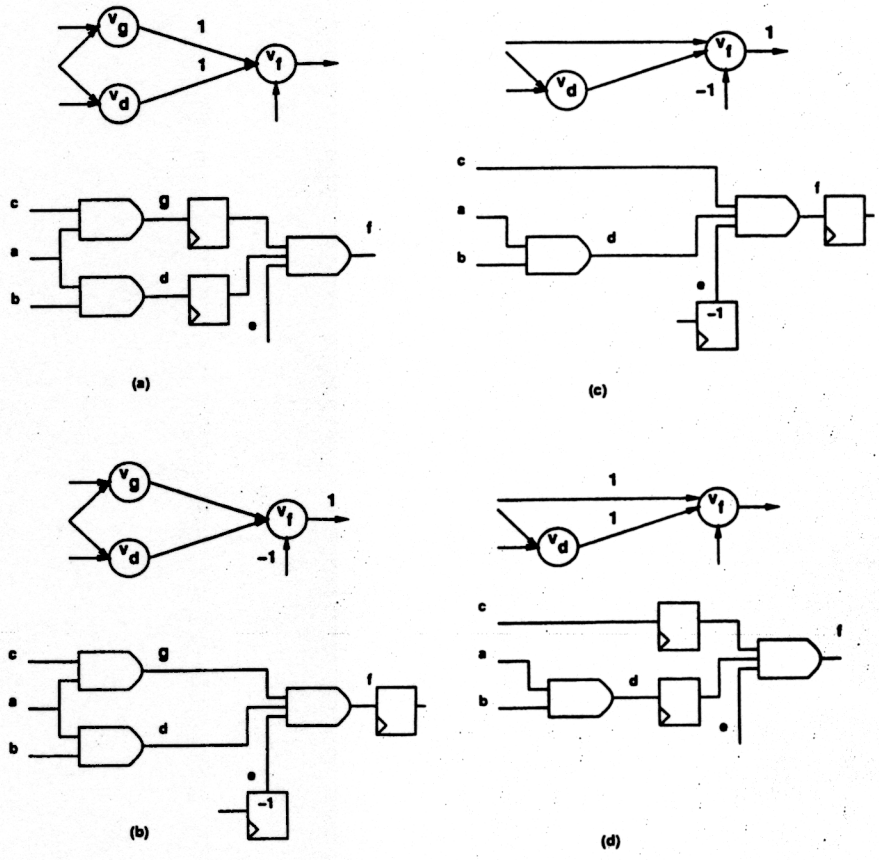

Fig. 3 (a) Synchronous network and circuit. (b) Modified network and circuit by peripheral retiming. (c) Optimized network and circuit after combinational optimization. (d) Synchronous network and circuit after final retiming.

time from the environment (outside the periphery), to extract the registers.

Example 3.2: Consider the synchronous circuit of Fig. 3(a) [14]. One *AND* gate is redundant, but combinational techniques cannot detect it, when operating on the network fragments obtained by removing the registers. Figure 3(b) shows the circuit after peripheral retiming, where the registers have been pushed to the periphery of the circuit. Note that one register has negative weight, corresponding to borrowing time from the environment.

The result of applying combinational optimization to the peripherally retimed circuit is shown in Fig. 3(c), where the redundancy is detected and eliminated. The circuit after retiming is then shown in Fig. 3(d). Note that no negative delay is left in the circuit. □

Two issues are important to apply this method. First, the determination of the circuit class for which peripheral retiming is possible. Second, the characterization of the combinational transformations for which there exist a valid retiming, so that negative weights (if any) can be removed.

For peripheral retiming, the circuits must be modeled by definite synchronous logic networks with the following property [14]. When considering any I/O pair ($v_i \in V^I$, $v_j \in V^O$), no two paths $v_i, \cdots, v_j$ must differ in path weights and for any I/O path $v_i, \cdots, v_j$ the weight must equal the sum of two integers $\alpha_i + \beta_j$ associated with v_i and v_j. This is a fairly strong restriction. For example, the circuits of Fig. 1 and 8 do not satisfy this condition, while that of Fig. 3(a) does. Obviously pipelined networks satisfy the assumptions for peripheral retiming.

Let us consider now the applicable logic transformations. Malik *et alii* showed that a valid retiming of a peripherally retimed network requires non-negative input/output path weights. Obviously, it is always possible to retime a peripherally retimed circuit whose topology has not changed, by restoring the original position of the registers. Nevertheless, some combinational logic transformations may introduce input/output paths with negative weights. When this happens, the transformations must be rejected.

The importance of peripheral retiming stems from the fact that the optimization of the register position can be performed in conjunction with the optimization of the combinational logic. The paradigm for peripheral retiming allows us to separate the two tasks, and

therefore leveraging powerful existing synthesis programs.

Recent related work by Dey et al. [6] has shown a way of partitioning synchronous circuits into subnetworks, called *consistent corollae*, defined on the basis of signal reconvergence. They showed that such corollae satisfy the assumption of peripheral retiming. Dey proposed a general method for circuit optimization, based on corolla partitioning, peripheral retiming and combinational logic resynthesis.

3.3 Algebraic Transformations for Synchronous Circuits

Unfortunately, many synchronous logic networks do not satisfy the assumptions for performing peripheral retiming. Notable examples are those where two paths with different path weight reconverge, as for example those shown in Figs. 1 and 8. In this case, circuit optimization can be performed by combining retiming and combinational logic transformations. We call *synchronous algebraic transformations* the extensions of the algebraic transformations for combinational logic [5] that incorporate local retiming.

Examples of synchronous algebraic transformations are synchronous elimination, substitution, extraction and decomposition [11]. They are extension of the corresponding combinational transformations [5]. In this section, we call *fanin* (*fanout*) set of a vertex the subset of vertices that are tail (head) of an edge whose head (tail) is that vertex, and we denote the set by $FI(\cdot)$ $(FO(\cdot))$.

The *elimination* of a variable with label $(n+k)$ is the replacement of the variable by its corresponding expression retimed by k. Given two internal vertices v_i and $v_j \in FI(v_i)$, the elimination of v_j into v_i is the elimination of variable j in all its occurrences in the expression $\mathcal{J}$ for v_i. The elimination of vertex v_j is its elimination into all the vertices in $FO(v_j)$. Note that the elimination of a variable with label zero is equivalent to the elimination used in combinational logic synthesis [5]. The elimination of a variable with non-zero label corresponds to merging two logic gates that are separated by a register, by shifting the register to the inputs of the gate corresponding to the variable being eliminated. An example is shown in Fig. 4, where variable $c^{(n-1)}$ has been eliminated. This corresponds to merging of the *AND* and *OR* gates into a complex gate, and to shifting of the registers to its inputs.

Let us consider now *substitution* (also called resubstitution) [5] for synchronous logic networks. Let $\mathcal{J}$, $\mathcal{J}$, $\mathcal{Q}$ and $\mathcal{R}$ be algebraic expressions of Boolean functions. Then $\mathcal{J}$ is a *synchronous divisor* of $\mathcal{J}$ if $\exists k \leq 0$ such that $\mathcal{J} = R^k(\mathcal{J})\mathcal{Q} + \mathcal{R}$ and $R^k(\mathcal{J})\mathcal{Q} \neq \phi$. Given two internal vertices v_i and v_j such that the expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{J}$, the substitution of v_j into v_i is the factoring of $\mathcal{J}$ as $R^k(j) \cdot \mathcal{Q} + \mathcal{R}$. An algorithm for synchronous division was presented in Ref. [11]. The algorithm takes as inputs two synchronous expressions $\mathcal{J}$, $\mathcal{J}$ and it attempts to divide $\mathcal{J}$ by $R^k(\mathcal{J})$ for decreasing values of k starting from 0. Standard division algorithms are used [5] with arguments that are syntactically modified copies of $\mathcal{J}$, $\mathcal{J}$ where labeled variables are aliased. The algorithms terminates successfully when a non-trivial quotient $\mathcal{Q}$

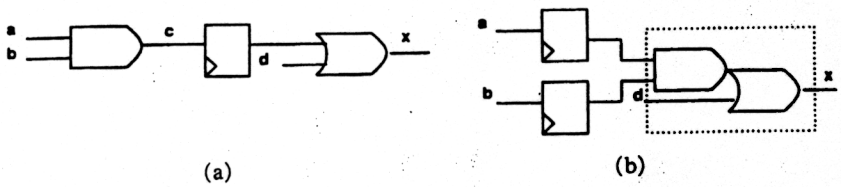

Fig. 4 (a) Fragment of synchronous network: $x^{(n)} = d^{(n)} + c^{(n-1)}$; $c^{(n)} = a^{(n)}b^{(n)}$. (b) Example of synchronous elimination: $x^{(n)} = d^{(n)} + a^{(n-1)}b^{(n-1)}$.

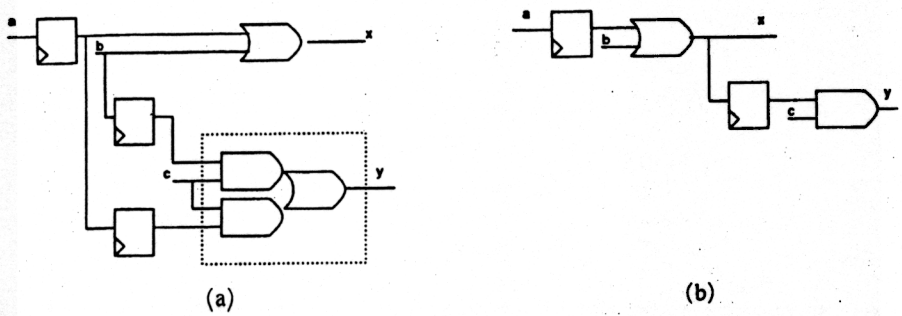

Fig. 5 (a) Fragment of synchronous network: $x^{(n)} = a^{(n-1)} + b^{(n)}$; $y^{(n)} = a^{(n-2)}c^{(n)} + b^{(n-1)}c^{(n)}$. (b) Example of synchronous substitution: $x^{(n)} = a^{(n-1)} + b^{(n)}$; $y^{(n)} = x^{(n-1)}c^{(n)}$.

is found, i.e. $R^k(\mathcal{J})\,\mathcal{Q} \neq \phi$. Otherwise it terminates when k is such that at least a variable in $R^k(\mathcal{J})$ has label smaller than the corresponding variable in $\mathcal{J}$, thus insuring the lack of any non-void quotient for any current or smaller value of k.

Note again that the divisors defined in Ref. [5] are a subset of the synchronous divisors and therefore substitution with zero retiming (i.e. $k=0$) is equivalent to substitution in combinational logic. The substitution of a variable with non-zero retiming corresponds to adding one (or more) register between two gates to simplify the latter. An example is given in Fig. 5. The complex gate corresponding to variable $y^{(n)}$ is simplified by using variable $x^{(n)}$ retimed by -1, i.e. $R^{-1}(x^{(n)}) = x^{(n-1)}$.

The *extraction* of a common sub-expression of expressions $\mathcal{J}$ and $\mathcal{J}$ corresponding to two vertices v_i and v_j is the addition to the network of a vertex v_l (with the related edges) corresponding to a common synchronous divisor of $\mathcal{J}$ and $\mathcal{J}$ and to the factoring of $\mathcal{J}$ and $\mathcal{J}$ in terms of the new variable l. Similarly, the *decomposition* of an expression $\mathcal{J}$ its replacement by the expression: $R^k(j)\,\mathcal{Q} + \mathcal{R}$, where j is a new variable, its corresponding expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{J}$, k is an integer and $R^k(j)\,\mathcal{Q} \neq \phi$. The decomposition of a vertex v_i implies the addition to the network of vertex v_j. Decomposition can be applied recursively to v_i and v_j.

Synchronous algebraic transformations can be combined with combinational logic transformation and global retiming. In particular, it was shown that synchronous elimination can be applied to gates that are head of critical paths and synchronous substitution to gates that are tails of critical paths. In both cases, such transformations are often the only ones that can locally improve the cycle time. Unfortunately, it was also shown that the frequency in which such transformations can be applied successfully in real circuits is low.

3.4 Boolean Transformations for Synchronous Circuits

Boolean transformations for logic synthesis exploit the full power of the Boolean representation and the use of *don't care* sets. Hence, overall optimization results are potentially better than those achieved by using algebraic methods only. Boolean simplification is an example of a Boolean transformation. It consists of replacing a local function (modeling a logic gate) of a synchronous logic network by a simpler one, where simpler may relate to smaller area and/or delay. Let us consider a generic vertex of the network, say v_x. The replacement of a local function f_x with another one g_x can be seen as a local *perturbation* $\delta = f_x \oplus g_x$. The replacement (perturbation) is feasible as long as the input/output behavior of the network does not change. An area/delay optimal replacement can be chosen among the feasible ones.

A key problem for sequential optimization is checking the feasibility of a perturbation. In the combinational case, it is possible to associate with each vertex of network a local *don't care* set, represented as a Boolean function over network variables. The inclusion of the perturbation in the local *don't care* set is then a necessary and sufficient condition for the feasibility of the replacement. In other words, local upper bounds on the perturbation can be derived from the overall network.

In the sequential case, it is also possible to compute the perturbation induced by a local replacement. Similarly, it is possible to compute local *don't care* conditions, that can be expressed by Boolean functions and represented, for example, by *sum of products* of labeled variables. Differently from the combinational case, the inclusion of the perturbation in the local *don't care* set is a sufficient but not necessary condition for the validity of the replacement [8], [9].

The reason for the difference can be explained from a theoretical point of view as follows. Whereas the input/output behavior of combinational networks can be represented as Boolean functions, the input/output behavior of sequential networks is naturally represented by a relation between input and output traces. The relational model encapsulates wider degrees of freedom in finding equivalent representations. Unfortunately, these degrees of freedom cannot always be represented in terms of local Boolean functions.

From a practical standpoint, there are two major approaches for implementing Boolean transformations that exploit *don't care* conditions. The former is to compute *explicit* local *don't care* sets, whose bounding the perturbation guarantees that the transformation is feasible. This approach entails an extension of the methods for computing combinational *don't care* sets. Unfortunately, it may not capture all feasible transformations, and therefore is suboptimal. The latter method is to represent the *don't care* sets in an *implicit* way, by using a relational representation. The added complexity of this approach is rewarded by the possibility of representing all the degrees of freedom for sequential optimization at any vertex of the synchronous logic network.

3.4.1 Explicit *Don't Care* in Conditions Synchronous Networks

As in the case of combinational circuits, *don't care* conditions are related to the impossible patterns that are input to a (sub-) network, called *controllability don't cares* and to those for which the outputs are not sampled, called *observability don't cares*. Differently from the combinational case, observability *don't care*

sets represent the observability of a variable at present and future times. In general, *don't care* conditions in synchronous circuits may contain *time-invariant* and *time-dependent* components. Only the use of the former is straightforward for logic simplification. The latter may relate to the circuit initialization or to periodic patterns produced by the circuit [7].

Example 3.3: Consider the circuit of Fig. 6. Let us consider the input controllability *don't care* conditions for network N_2. Assume that the network is initialized by the sequence $(b^{(-4)}, b^{(-3)}, b^{(-2)}) = (1, 0, 1)$. The limited controllability of the inputs of N_2 is reflected by the set of its impossible input sequences, denoted by CDC_{in}. For example, $u^{(n)}v'^{(n+1)}$ is an impossible input sequence for N_2. Indeed for $u^{(n)}$ to be equal to 1 it must be $a^{(n)}=b^{(n)}=1$; but $b^{(n)}=1$ implies $v^{(n+1)}=1$. Hence, for N_2.

$$u^{(n)}v'^{(n+1)} \subseteq CDC_{in};\ \forall n \geq -4.$$

This is an example of a time-invariant *don't care* condition.

As a consequence of the initializing sequence, output v cannot assume the value 0 at time $-3, -1$. Hence:

$$v'^{(-3)} + v'^{(-1)} \subseteq CDC_{in}.$$

This is instead an example of a transient *don't care* condition, due to the initialization of the network.

The interconnection of the two networks limits the observability of the primary outputs of N_1. We compute now the output observability *don't care* conditions for variable $v^{(n)}$ of N_1.

In particular, the output of N_2 can be expressed in terms of u and v as:

$$x^{(n)} = y^{(n-1)} + y'^{(n)} + u'^{(n-1)}$$
$$= u'^{(n-1)} + v^{(n-1)} + u^{(n)} \oplus v^{(n)}.$$

The value of $v^{(n)}$ can be observed at the output of N_2 only at time n or at time $n+1$. In particular, $v^{(n)}$ is observable at time n if $y^{(n-1)}=0$ and $u^{(n-1)}=1$. The observability *don't care* of v at time n can thus be described by the function:

$$ODC_{v^{(n)}}^{(n)} = u'^{(n-1)} + y^{(n-1)} = u^{(n-1)} + v'^{(n-1)}$$

while the observability at time $n+1$ is described by:

$$ODC_{v^{(n)}}^{(n+1)} = (y^{(n+1)}u^{(n)})'$$
$$= y'^{(n+1)} + u'^{(n)}$$
$$= u^{(n+1)} \oplus v^{(n+1)} + u'^{(n)}.$$

Sufficient conditions for never observing $v^{(n)}$ at the primary output of N_2 are described by:

$$ODC_{v^{(n)}} = ODC_{v^{(n)}}^{(n)}\ ODC_{v^{(n)}}^{(n+1)}$$

in particular containing the cube $u'^{(n-1)}u'^{(n)}$. Since $u^{(n)}=a^{(n)}b^{(n)}$, then $(a'^{(n-1)}b'^{(n-1)})\,(a'^{(n)}+b'^{(n)})$ belongs to the observability *don't care* set of N_1 associated with output $v^{(n)}$. Thus $a'^{(n-1)}a'^{(n)}$ is an input sequence for N_1, that represent a situation when output $v^{(n)}$ is not observed by the environment. □

The computation of the controllability and observability *don't care* sets associated with the variables of a sequential network can be achieved by network traversal methods, for definite networks, and by iterative methods for cyclic networks. We refer the interested reader to Ref. [7] for details.

It is important to remark that the simplest case for *don't care* computation is the one of pipeline networks. In these networks, all reconverging paths have equal weights and hence the *don't care* express the network constraints in terms of variables with the same time label. In other words, the presence of registers in pipeline networks does not add to the complexity of the *don't care* set computation, which is equal to that of combinational networks. It is interesting to note that peripheral retiming can always be applied to pipelined networks, so that all registers can be moved to the circuit periphery for computing the *don't care* conditions.

Once the local don't care sets have been computed, Boolean simplification can be applied by invoking, for example, a two-level logic minimizer on the local function and the associated *don't care* set. This is a straightforward extension of Boolean simplification of combinational networks to functions with time labeled variables. In practice, the time labeled variables are aliased by other variables when the minimizer is invoked.

Example 3.4: Consider the network N_2 of Fig. 6. Let us consider the optimization of the function for variable $y^{(n)}=u^{(n)} \oplus v^{(n)}$ The local *don't care* conditions

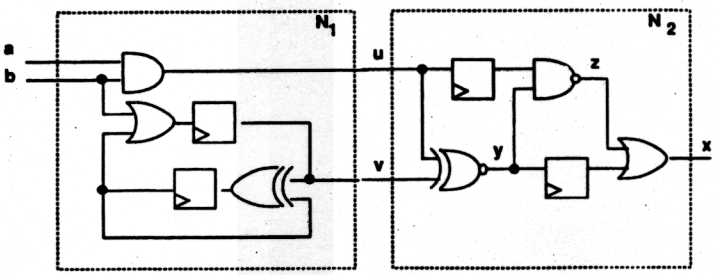

Fig. 6 Interconnected networks.

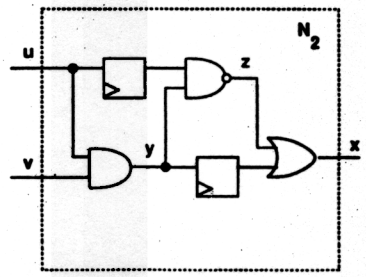

Fig. 7 Optimized network N_2.

for this variable include $u'^{(n-1)}u'^{(n)}+u^{(n-1)}v'^{(n)}$, as derived in the Example 3. (The first term is contained in $ODC_{v^{(n)}}$ and the second in the time-invariant component of CDC_{in} for N_2.) Let us rename the variables, for the sake of clarity. Let $u=u^{(n)}$; $v=v^{(n)}$; $w=u^{(n-1)}$. Then, we want to minimize $y=uv+u'v'$ with *don't care* set $w'u'+wv'$. Since $u'v'$ is included in the *don't care* set, we can write $y=uv$. Returning to the labeled notation, the local function can be replaced by $y^{(n)}=u^{(n)}v^{(n)}$, with a savings of two literals, by Boolean simplification, as shown in Fig. 7. □

3.4.2 Implicit *Don't Care* Conditions in Synchronous Networks

In the case of synchronous circuit optimization, there may exist feasible replacements of local functions whose corresponding perturbations are not included in the local explicit *don't care* set. It may then be necessary to resort to a relational model if we want to consider all possible degrees of freedom for optimization.

Example 3.5: Consider the network of Fig. 8(a). Assume that the external *don't care* set is empty, for the sake of simplicity. It can be easily verified that the inverter can be replaced by a direct connection, leading to the simpler and equivalent circuit of Fig. 8(b).

We try to interpret this simplification with the methods reviewed before. First, note that the network has two input/output paths of unequal weight, ruling out the use of (peripheral) retiming techniques. Second, note that the inverter can be removed, even though the perturbation is $\delta=x'^{(n)}\oplus x^{(n)}=1$. If the perturbation had to be included in the local *don't care* set, this could lead us to the erroneous conclusion that the local *don't care* set $DC_{y^{(n)}}=1$, and hence that y could be replaced by a permanent TRUE or FALSE value. □

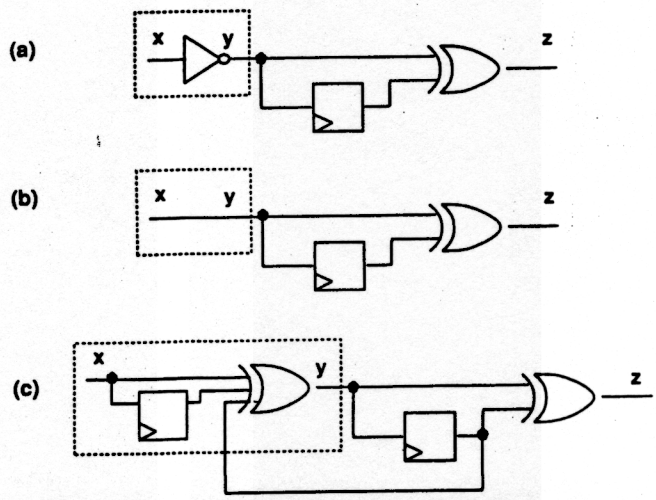

Fig. 8 (a) Example of a non-retimable but optimizable circuit. (b) Optimized circuit implementation. (c) Other circuit implementation.

As we stated before, whereas the inclusion of the perturbation in a local *don't care* set is a sufficient condition for equivalence, it is by no means necessary in the case of synchronous networks. Therefore we must search for general conditions for checking the feasibility of replacements. The most general terminal specification of a synchronous circuit is its trace set. Hence transformations are feasible when they yield indistinguishable behavior in terms of input/output traces, for all time-points (possibly excluding the external *don't care* of the network).

Example 3.6: Consider again the network of Fig. 8 (a) and assume that the external *don't care* set is empty. The input/output behavior of the network is:

$$z^{(n)}=x'^{(n)}\oplus x'^{(n-1)}\forall\, n\geq 0.$$

Consider subnetwork N_1 shown inside a box in the Figure. Any implementation of N_1 is valid as long as:

$$y^{(n)}\oplus y^{(n-1)}=x'^{(n)}\oplus x'^{(n-1)}\forall\, n\geq 0.$$

The above equation represents the constraints on the replacement for subnetwork N_1. Possible solutions are the following:

- $y^{(n)}=x'^{(n)}\forall\, n\geq 0$. This corresponds to the original network, shown in Fig. 8(a).
- $y^{(n)}=x^{(n)}\forall\, n\geq 0$. This corresponds to removing the inverter, as shown in Fig. 8(b). (It can be derived by noticing that the parity function is invariant to complementation of an even number of inputs).
- $y^{(n)}=x^{(n)}\oplus x^{(n-1)}\oplus y^{(n-1)}\forall\, n\geq 0$. This solution can be derived by adding the term $\oplus y^{(n-1)}$ to both sides of the equation. The corresponding circuit is shown in Fig. 8(c).

Note that the last implementation of the network introduces a feedback connection. □

Equating the terminal behavior of the original network and of the one embedding the local replacement gives rise to an implicit *synchronous recurrence equation*, that relates the network variables at different time-points. The synchronous recurrence equation specifies the network and implicitly all the degrees of freedom for its optimization. Hence *don't care* conditions are represented *implicitly*. Circuit optimization based on synchronous recurrence equations is widely applicable to sequential synchronous circuits. In general, optimization algorithms must insure a feasible solution that satisfies some optimality criterion. Feasibility implies finding a function and initial conditions (for the replacement) that satisfy the synchronous recurrence equation at all time-points of interest [8]. Optimality may be related to area or delay estimates.

Damiani *et alii* [8], [9] proposed a method for computing a minimum *sum* of *product* replacement, under the simplifying assumption that no further cycles

are introduced in the network. We present here the highlights of the method, by elaborating on an example. A tabulation of the possible values of the variables leads to the specification of a *relation table*, relating the possible input and output traces that satisfy the equation.

Example 3.7: Consider again the network of Fig. 8 (a). The corresponding synchronous recurrence equation is:

$$y^{(n)} \oplus y^{(n-1)} = x'^{(n)} \oplus x'^{(n-1)} \forall\, n \geq 0.$$

that can be tabulated as follows:

$x^{(n)}\ x^{(n-1)}$	$y^{(n)}\ y^{(n-1)}$
0 0	{00, 11}
0 1	{01, 10}
1 0	{01, 10}
1 1	{00, 11}

The second column of the table shows the possible output traces in conjunction with the input traces. □

Circuit optimization can be done by using a relation minimizer that finds the optimum function for the subnetwork to be replaced that is compatible with the corresponding Boolean relation [3]. Compatibility must be insured for all time-points. Alternatively, the solution can be determined by representing the desired replacement by a truth table in terms of unknown coefficients. Constraints on the feasible values of the coefficients can be inferred from the relation table. Obviously, truth tables satisfying the constraints correspond to feasible network replacements. Among these, an optimal solution may be chosen.

Example 3.8: Assume that network N_1 of Fig. 8 is replaced by a function specified by the following truth table:

$x^{(n)}\ x^{(n-1)}$	f
0 0	f_0
0 1	f_1
1 0	f_2
1 1	f_3

We can now re-express the constraints of the relation table in terms of the coefficients. Consider output traces {00, 11} implying $y^{(n)} = y^{(n-1)}$, or equivalently $f^{(n)}(x^{(n)}, x^{(n-1)}) = f^{(n-1)}(x^{(n-1)}, x^{(n-2)})$. The corresponding input traces are {000, 001, 110, 111}. For input trace 000, $f^{(n)}(0,0) = F^{(n-1)}(0,0)$ implies $f_0 = f_0$ or equivalently $(f_0' + f_0)(f_0 + f_0') = 1$. For input trace 001, $f^{(n)}(0,0) = f^{(n-1)}(0,1)$ implies $f_0 = f_1$ or equivalently $(f_0' + f_1)(f_0 + f_1') = 1$. Similar considerations apply to the remaining input traces {110, 111} and to those related to output traces {01, 10}. The resulting constraints on the coefficients, excluding the tautological ones and duplications, are:

$$(f_0' + f_1)(f_0 + f_1') = 1$$
$$(f_3' + f_2)(f_3 + f_2') = 1$$
$$(f_1' + f_2')(f_1 + f_2) = 1$$
$$(f_1' + f_3')(f_1 + f_3) = 1$$
$$(f_2' + f_0')(f_2 + f_0) = 1$$

Solutions are: $f_0 = 1$; $f_1 = 1$; $f_2 = 0$; $f_3 = 0$ and $f_0 = 0$; $f_1 = 0$; $f_2 = 1$; $f_3 = 1$. The first solution corresponds to selecting $y^{(n)} = x'^{(n)}$, while the second to $y^{(n)} = x^{(n)}$. □

With this formalism, the possible solutions are the sets of coefficients that make all the clauses true. Among the feasible solutions, an optimal one can be chosen to satisfy any particular property, e.g. delay or number of literals. The search for a feasible or optimum solution requires solving a *binate covering* problem, which is a minimum-cost satisfiability problem. The binate nature stems from the fact that coefficients can appear in the clauses with both polarities. Exact and heuristic methods can be used for the optimal synthesis of the function [8], [9].

We summarize here the most important points of this approach. First, the direct synthesis of a function that replaces a subnetwork is used instead of the classical Boolean optimization step. Second, the degrees of freedom (represented by *don't care* conditions in classical Boolean optimization) are represented implicitly as constraints implied by a synchronous recurrence equation. Third, the synthesis methods involves the minimization of a Boolean relation or the solution of a binate covering problem in terms of unknown coefficients of the truth table of a Boolean function.

This synthesis technique allows us to define a circuit transformation that is applicable across registers even in presence of reconverging paths with different weights. Therefore, it is the most general transformation that can be applied among those described here. It subsumes Boolean simplification and division. Unfortunately, the solution to this problem is hard to achieve, because it involves binate covering and because the problem size grows exponentially with the number of inputs of the subnetwork being replaced. Nevertheless, the method has been shown to be widely applicable to circuit optimization [8], [9], by limiting the size of the support of the local function to be replaced.

4. Conclusions

Logic-level optimization of synchronous digital circuits can be performed by using structural network models and by applying circuit transformations. This

approach has advantages over those using state-based models, because area and path delays are explicit in the representation and the effect of circuit transformation can be evaluated precisely. The disadvantage of structural modeling style is its lower level of abstraction and hence the potential larger amount of information required to describe a circuit, when compared to state transition diagrams.

Several optimization algorithms have been proposed for sequential circuits using structural models. Retiming can be used to select the optimum register positions. Peripheral retiming extends the retiming concept to extract the registers from a region, where combinational optimization can be applied. Algebraic and Boolean transformations mix combinational logic restructuring with local register movement. The overall circuit optimization can be performed by executing a sequence of transformations of different types, as for combinational multiple-level logic optimization. The nature of the system driving the logic transformations can be very general [2], [5], [10], [12], [15].

Results on using structural optimization techniques have been very encouraging [6]-[9], [11], [14]. Nevertheless there are still open problems that will stimulate further research in the field. The most important issue that needs to be addressed is understanding the relations between transformations on structural models and optimization of state-based ones. Indeed logic transformations that affect registers change the state count and their encoding, as well as state minimization and encoding change the underlying structure of the circuit. Understanding and predicting the modification of both the structural and behavioral views of sequential circuits is important from both a theoretical and practical standpoint. Other open problems are related to perfecting the logic transformations, by making them more efficient and applicable to subcircuits of larger size. For example, Boolean transformations using the synchronous recurrence equation model require solving an intractable problem of exponential size in the number of the inputs of the subcircuit (or function) to be replaced. This motivates the development of smart heuristics.

Acknowledgments

This research was sponsored by NSF/ARPA, under grant No. MIP 9115432 and by DEC jointly with NSF, under a PYI Award program.

References

[1] Ashar, P., Devadas, S. and Newton, A. R., *Sequential Logic Synthesis*, Kluwer, 1991.

[2] Bartlett, K., Cohen, W., De Geus, A. and Hachtel, G., "Synthesis and Optimization of Multilevel Logic under Timing Constraints," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. CAD-5, no. 4, pp. 582-596, Oct. 1986.

[3] Brayton, R., Hachtel, G. and Sangiovanni, A., "Multilevel Logic Synthesis," *IEEE Proceedings*, vol. 78, no. 2, pp. 264-300, Feb. 1990.

[4] Bartlett, K., Borriello, G. and Raju, S., "Timing Optimization of Multiphase Sequential Circuits," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. 10, no. 1, pp. 51-62, Jan. 1991.

[5] Brayton, R. K., Rudell, R., Sangiovanni-Vincentelli, A. and Wang, A. R., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. 6, no. 6, pp. 1062-1081, Nov. 1987.

[6] Dey, S., Brglez, F. and Kedem, G., "Partitioning Sequential Circuits for Logic Optimization," *Proceedings of 3rd International Workshop on Logic Synthesis*, Research Triangle Park, 1991.

[7] Damiani, M. and De Micheli, G., "*Don't care* Specifications in Combinational and Synchronous Logic Circuits," *IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst.*, Mar. 1993, and *CSL Technical Report*, CSL-TR 92-531, 1992.

[8] Damiani, M. and De Micheli, G., "Synthesis and Optimization of Synchronous Logic Circuits from Recurrence Equations," *Proceedings of EDAC*, pp. 226-231, 1992.

[9] Damiani, M. and De Micheli, G., "Recurrence Equations and the Optimization of Synchronous Logic Circuits," *DAC, Proceedings of the Design Automation Conference*, Annaheim, pp. 556-561, Jun. 1992.

[10] Darringer, J., Brand, D., Gerbi, J., Joyner, W. and Trevillyan, L., "LSS: A System for Production Logic Synthesis," *IBM Journal of Research and Development*, vol. 28, no. 5, pp. 537-545, Sep. 1984.

[11] De Micheli, G., "Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization," *IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst.*, vol. 10, no. 1, pp. 63-73, Jan. 1991.

[12] De Micheli, G., "Performance-oriented synthesis in the Yorktown Silicon Compiler," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. CAD-6, no. 5, pp. 751-765, Sep. 1987.

[13] Leiserson, C. and Saxe, J., "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5-35, 1991.

[14] Malik, S., Sentovich, E. M., Brayton, R. K. and Sangiovanni-Vincentelli, A., "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. 10, no. 1, pp. 74-84, Jan. 1991.

[15] Muroga, S., Kambayashi, Y., Lai, H. and Culliney, J., "The Transduction method-Design of Logic networks based on permissible functions," *IEEE Trans. Comput.*, vol. 38, no. 10, pp. 1404-1424, Oct. 1989.

[16] Saucier, G., de Paulet, M. C. and Sicard, P., "ASYL: A Rule-Based System for Controller Synthesis," *IEEE Trans. Comput. -Aided Des. Integrated Circuits & Syst.*, vol. CAD-6, no. 6, pp. 1088-1097, Nov. 1987.

[17] Singh, K., Wang, A., Brayton, R. and Sangiovanni, A., "Timing Optimization of Combinational Logic," *Proceedings of ICCAD*, pp. 282-285, 1988.

Giovanni De Micheli is Associate Professor of Electrical Engineering and, by courtesy, of Computer Science at Stanford University. From 1984 to 1986 he worked at the IBM T.J. Watson Research Center, Yorktown Heights, New York, where he was project leader of the Design Automation Workstation group. Previously he held positions at the Department of Electronics of the Politecnico di Milano, Italy and at Harris Semiconductor, Melbourne, Florida. He received a Dr.Eng. degree, *Summa cum Laude*, in Nuclear Engineering from the Politecnico di Milano, Italy, in 1979, a M.S. and a Ph.D. degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 1980 and 1983 respectively. Dr. De Micheli was granted a *Presidential Young Investigator* award in 1988. He received the 1987 *Best Paper Award* for the best paper published on the IEEE Transactions on CAD/ICAS and two *Best Paper Awards* at the Design Automation Conference, in 1983 and in 1993. His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, optimization and verification of VLSI circuits. He is co-editor of the book: *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1987 and co-author of the book: *High-level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer, 1992. He was also co-director of the Advanced Study Institute on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, under the sponsorship of NATO in 1986 and in 1987. Dr. De Micheli is a Senior Member of IEEE. He is associate editor of the IEEE *Transactions on VLSI Systems* and of *Integration: the VLSI Journal*. He was technical and general chairman of the *International Conference on Computer Design-ICCD* in 1988 and 1989 respectively. He has served as member of the technical committee of the *ICCD*, *ICCAD* and *DAC* Conferences.