Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization

Giovanni De Micheli, *Senior Member, IEEE*

Abstract—**This paper presents a new approach to logic synthesis of digital synchronous circuits. We present a model for synchronous circuits that supports logic transformations aimed at optimizing the circuit performance. Previous synthesis approaches attacked this problem by separating the combinational logic from the registers and by applying circuit transformations to the combinational component only. We show in this paper instead how to optimize concurrently the circuit equations and the register position by a set of algorithms based on logic transformations. Experimental results on benchmark circuits are reported.**

I. Introduction

THE IMPORTANCE of logic synthesis is pivotal in the computer-aided design of integrated circuits. Logic synthesis systems have been the object of extensive investigation, and commercial implementations have shown to be practical for product-level design of digital circuits.

Most digital designs are **synchronous** logic circuits that are interconnections of logic gates and registers with synchronous clocking. Feedback connections are restricted to be through synchronous registers, to guarantee race-free design. Semi-custom circuit implementations, such as standard cells and sea-of-gates, have motivated the use of multiple level (or multiple stage) logic synthesis techniques. In particular, such implementations have shown to be more flexible and faster than two-level implementations, such as programmable logic arrays. As a result, several techniques for multiple level logic synthesis techniques have been investigated and clever algorithms for combinational logic synthesis have been reported in the literature [1]–[5].

However, techniques for synthesizing synchronous logic circuits have been lagging behind, due to the additional complexity of handling registers and feedback connections. Some logic synthesis systems deal with such circuits by partitioning them into an interconnection of a combinational logic component and registers [1]. The combinational portion of the circuit is optimized by combinational logic algorithms. Then registers are added back to the circuit. Needless to say, such optimization techniques are limited in their scope by this partitioning strategy. An attempt to overcome this problem was recently proposed [6] in which registers are temporarily removed to extract the largest portion of a synchronous circuit that can be dealt with by combinational logic techniques.

Some other logic synthesis cope with synchronous designs by exploiting heuristic solutions to classical problems, such as state minimization and state assignment [7]. In this case, the optimization is done on a logic **behavioral** model in terms of state diagrams or equivalent representations. The drawbacks of this approach are two-fold. First, it is hard to evaluate correctly the circuit timing characteristics and to develop timing optimization algorithms. Secondly, it is not possible to improve a netlist-based circuit specification in a step-wise way in order to take advantage of the original circuit structure.

In this paper we attack the synchronous logic synthesis problem by considering algorithms that operate on the **structural** specification of a synchronous circuit. We consider circuit models that do not separate registers from the combinational component. For this reason, we introduce the concept of synchronous Boolean network and we study transformations on this network that preserve I/O equivalence and that can be used to improve the circuit cycle time and/or area in a step-wise way. Some of these transformations are extensions of those used in combinational logic synthesis and operate *within* and *across* the register boundaries, by exploiting the possibility of moving the register positions.

It is important to remember that a technique to position the registers in a network, called **retiming**, was introduced by Leiserson and Saxe [8] in a different context, where logic synthesis transformations were not considered. Indeed the retiming-based algorithms presented in [8] find the register assignment corresponding to the fastest implementation (or minimal area implementation) of a network. Unfortunately, there has been no major use of retiming techniques in logic synthesis, because of the emphasis on combinational logic techniques. In addition, the optimality of Leiserson's algorithms has limited value in logic synthesis, because it assumes a static network topology, and therefore, disregards transformations that alter combinational logic gates and their interconnections.

This paper presents a model for synchronous logic synthesis that combines retiming with combinational logic synthesis techniques. We describe logic transformations that can be used to optimize the circuit area (under cycle-time constraints) or the cycle time (under area constraints). However, due to the novelty and complexity of the problem, we cannot at present report on a comprehensive approach to solving (even heuristically) these synchronous logic-synthesis problems. For this reason, we concentrate here on the use of logic transformations for the cycle-time minimization problem and we present some results on benchmark circuits to show the advantages and limitations of the approach.

II. Basic Concepts and Definitions

We consider structural models of digital circuits. Such circuits can be specified by an interconnection of combinational logic gates and clocked registers. We assume first that all the

Manuscript received January 1, 1990. This work was supported by the National Science Foundation under Contracts MIP-8710748 and MIP-8719546, and by DEC, AT&T, and Cray Research, jointly with the National Science Foundation under a PYI Award.

The author is with the Center for Integrated Systems, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4055.

IEEE Log Number 9039381.

0278-0070/91/0100-0063$01.00 ©

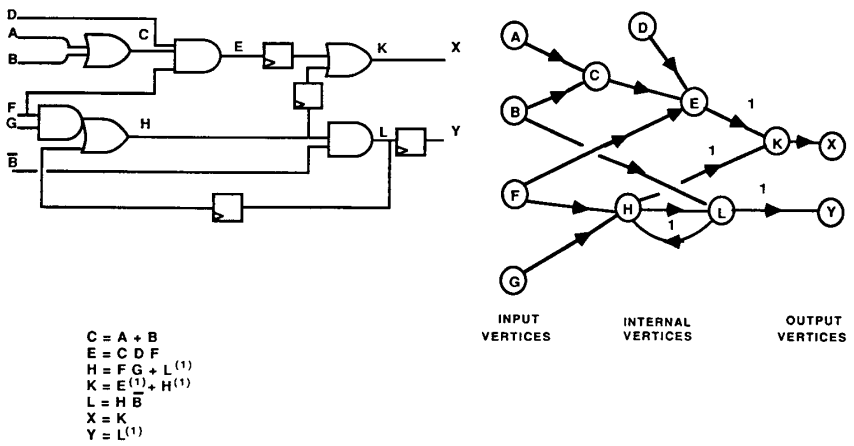


Fig. 1. Synchronous circuit and its representations.

registers are driven by one clock (i.e., single-phase circuits) and that the latching is always positive (or always negative) edge triggered. (Master–slave registers consisting of a cascade interconnection of latches gated by the clock and its complement fall in this class.) We assume that the clock has a period T (cycle time), and that the clock skew, the register setup, hold, and propagation times are negligible.[1]

We model synchronous circuits by **synchronous Boolean networks**. A synchronous Boolean network is described in terms of **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. A positive integer label on a variable (superscript) denotes the synchronous register delay, if any, of the corresponding signal with respect to the primary input or combinational logic gate that generates it. Zero-valued labels are omitted for the sake of simplicity. Each Boolean equation has an unlabeled variable (i.e., with zero-valued label) as a left term and a Boolean expression as a right term. The latter specifies the value of the left-term variable in terms of other (labeled) variables, i.e., it is a multiple-input single-output combinational logic function. We denote by $\mathcal{J}$ the Boolean expression associated to variable i.

The network is modeled by the **synchronous network graph**, that is, a directed weighted multigraph $G(V, E, W)$, whose vertex set $V = V^I \cup V^G \cup V^O = \{v\}$ 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. We denote v_i the vertex corresponding to variable i. The edge set E and the edge weight set W are defined as follows. There is an edge between v_i and v_j with weight k when variable i appears in the expression $\mathcal{J}$ for vertex v_j with label k. Zero-valued weights are not indicated by convention. There is a (weighted) edge to each output vertex in V^O from the internal vertex in V^G corresponding to the gate generating that output signal. 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. An example of a synchronous digital circuit and its representation is shown in Fig. 1.

In general, a synchronous Boolean network may have cyclic dependencies, i.e., its corresponding graph may be cyclic. A network is called **unidirectional** when the graph $G(V, E, W)$ is acyclic. Note that the combinational Boolean network (without synchronous registers) introduced by Brayton [1] is just a special case of the synchronous Boolean network that is acyclic and whose labels are all zeroes.

The (direct) **fan-in** set of a vertex v_i is the subset of vertices that are tail of an edge (with zero weight) whose head is v_i and is denoted by $FI(v_i)$ ($DFI(v_i)$). Similarly the (direct) **fan-out** set of a vertex v_i is the subset of vertices that are head of an edge (with zero weight) whose tail is v_i and it is denoted by $FO(v_i)$ ($DFO(v_i)$). Each internal vertex $v_i \in V^G$ (i.e., corresponding to a gate) has as attributes an area estimate l_i in terms of literal count [1] and a positive propagation gate delay d_i. Each input and each output vertex has zero delay.

The propagation delay model captures the difference in speed of gates implementing various Boolean expressions. Therefore, it is a function of the structure of the Boolean expression. For example, in the case of CMOS technology, such a structure is characterized by the maximum number of N-type and P-type devices in series. The delay function is assumed to be a monotonically increasing function of l_i. It is important to remark that an accurate gate propagation delay model should include loading factors and device sizes. We assume that the choice of device sizes for a gate is done in a successive stage of logic design, the technology mapping, so that it compensates for loading factors. Therefore, this propagation delay model includes an average loading factor.

Each vertex v_i has a **data ready time** t_i, that is, the time at which the signal generated by the corresponding gate is ready with respect to the clock edge [9]. We assume the primary inputs to be synchronized to the clock positive edge, and therefore, their data ready time is zero. For any other vertex v_i, the ready time is the sum of its propagation delay d_i to the largest data ready time of its inputs that are not registers, i.e.:

$$t_i = d_i + \max_{v_j \in DFI(v_i)} (t_j).$$

[1] In the case that the register setup t_s and propagation t_p times are not zero, it suffices to consider a reduced effective cycle time $T - t_s - t_p$, which represents an upper bound to the propagation delay through the combinational logic.

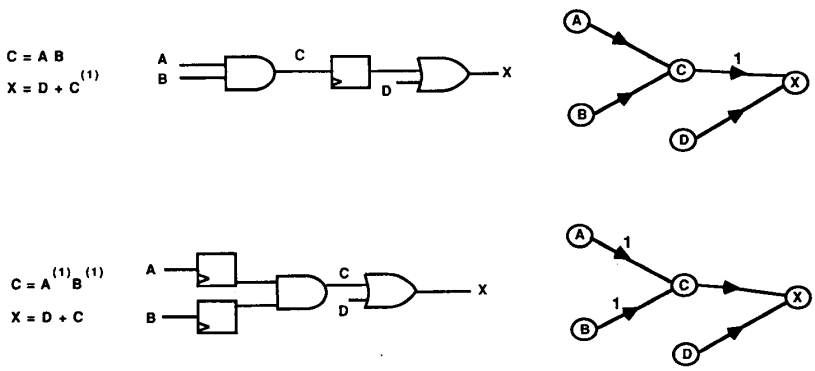


Fig. 2. Retiming vertex v_i by +1.

Since the subgraph representing the direct fan-in relation is acyclic, the data ready time can be computed by topological sort.

Given a cycle time T, a synchronous network is a **timing feasible** implementation if all the data ready times are bounded from above by the cycle time, i.e.:

$$T \geq \max_{v_i \in V} (t_i).$$

Each vertex v_i has a **slack** s_i representing the additional delay that the vertex can tolerate while preserving timing feasibility of the network for a given T [9]. In a (timing feasible) network a vertex is **critical** if its slack is negative (null).

The area taken by a network implementation depends on the total number of literals and registers required. For each variable i, let m_i be the maximum of the labels that the variable takes in the network representation. Then m_i represents the number of synchronous registers that are connected in cascade at the output of the corresponding gate. An area estimate can be computed as:

$$A = \alpha \sum_{v_i \in V^G} l_i + \beta \sum_{i \in V} m_i.$$

where α and β are coefficients taking into account the relative area cost of a literal and a register. Given an area bound $A_{\max}$, a network is an **area feasible** implementation if $A_{\max} \geq A$, and it is a **feasible** implementation if it is both area feasible and timing feasible.

III. Logic Transformations in Synchronous Logic Synthesis

The problem of minimizing the cycle time (area) of a synchronous Boolean network implementation, possibly under area (cycle time) constraints, is difficult and no efficient exact solution method is known. Most techniques for multiple level logic optimization are based on network transformations, that preserve the I/O equivalence of the network and achieve area/time optimal solutions with respect to some local criterion. Transformations are classified as **local** and **global**. Transformations are said to be local when they modify the representation of a Boolean expression at a time (e.g., factoring or Boolean simplification of an expression at one vertex of the network). Such transformations have been presented in [1], [2] for combinational logic synthesis and can be used (without significant extensions) in synchronous logic synthesis, because they do not depend on the network model. Global transformations target more than one expression at a time and they attempt to improve the network by restructuring the global interconnections (e.g., elimination, resubstitution, and extraction). We consider here global transformations extended to synchronous logic synthesis in relation with network retiming.

Retiming [8] is a technique that determines a register assignment in a network (i.e., a set of weights in $G(V, E, W)$) so that it is a timing feasible implementation for a given cycle time T, if such an assignment exists. In our context, the retiming of variable i by an integer r corresponds to adding r to its label, and the retimed variable is denoted by $i^{(\cdot + r)}$, where the dot in the superscript represents the label of variable i before retiming (e.g., for variable i witn label 2, fully denoted by $i^{(2)}$, a retiming by $r = 3$ yields $i^{(2+3)} = i^{(5)}$). Similarly, the retiming of an expression $\mathcal{I}$ by an integer r corresponds to adding r to the labels of all its operands and it is represented by $\mathcal{I}^{(\cdot + r)}$. The positive (negative) retiming of an internal vertex v_i by r_i is the shift of r_i register delays from its outputs (inputs) to its inputs (outputs). It corresponds to retiming by r_i the expression $\mathcal{I}$ of v_i and to retiming by $-r_i$ the variable i in the expressions of the vertices of $FO(v_i)$. An example is shown in Fig. 2.

The retiming of the I/O vertices corresponds to transferring synchronous delays from the circuit to the surrounding environment and vice versa. The retiming of an input vertex v_i by r_i corresponds to removing r_i synchronous delays from the corresponding input signal. Therefore, it is just the retiming by $-r_i$ of the variable i in all the expressions of the vertices of $FO(v_i)$. The retiming of an output vertex v_i by r_i corresponds to adding r_i synchronous delays on the output signal. Therefore, the retiming of an output vertex v_i by r_i is just the retiming by r_i of the expression $\mathcal{I}$ of v_i.

In the sequel, we refer to retiming as to the retiming of one or more vertices. It was shown in [12] that retiming the internal vertices preserves the I/O behavior of the network, provided that the resulting labels of the variables are non-negative. Therefore, the retiming of a vertex is valid only for some restricted values of r. Retiming all I/O vertices by the same quantity r preserves also the I/O behavior, again provided that the resulting labels are non-negative. Note that when the network graph is connected, if any I/O vertex is retimed, then all the I/O vertices must be retimed by exactly the same quantity r to preserve equivalence. A network retiming is **feasible** for a cycle time T, if the retimed network is a timing feasible implementation with non-negative labels and I/O equivalent to the original network.

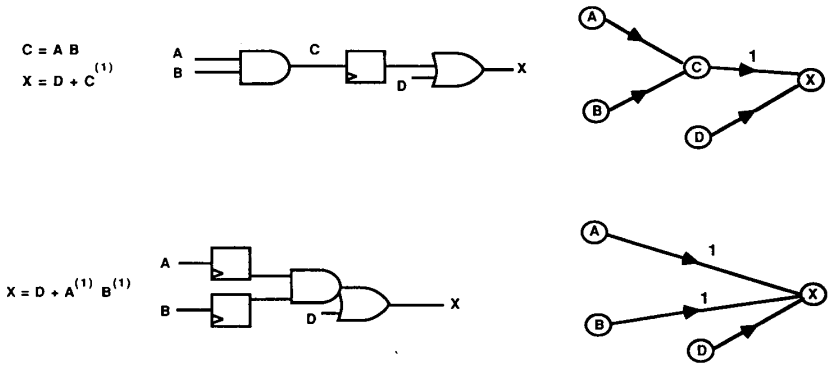


Fig. 3. Elimination of vertex v_c.

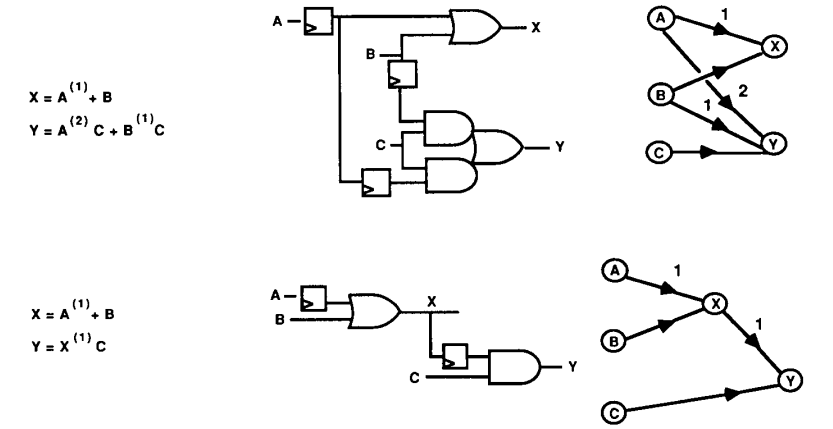


Fig. 4. Resubstitution of v_c into v_x.

Leiserson and Saxe proposed an algorithm in [8] that searches for the minimum T for which there exist a feasible retiming. The corresponding network is said to be **timing optimal with respect to retiming**. We consider here retiming in connection with logic transformations that alter the structure of the network graph.

The **elimination** of a variable with label 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 $\mathcal{G}$ for v_i (Fig. 3). 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 [1], [2]. The elimination of a variable with a nonzero 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.

Let us consider the area cost (or value) of an elimination, say v_j into v_i. An elimination changes the total number of literals in a network by δ_l. This number can be computed as $\delta_l = n_{ji}(l_j - 1) - l_j$, where n_{ji} is the multiplicity of variable j in expression $\mathcal{G}$ [1], [2]. When elimination is performed across a register boundary, then it is important to compare the saving in terms of literals with the possible increase of registers δ_m.

Example: Consider the circuit of Fig. 3. The variation in the number of literals is: $\delta_l = n_{cx}(l_c - 1) - l_c = 1(2 - 1) - 2 = -1$, i.e., one literal is saved. Assume that variable c is not used in any other expression and that $m_a = m_b = 0$, i.e., no register is present at the output of v_a and v_b. After the elimination one register is needed to delay a and b and no register is needed at the output of v_c that is deleted from the network. Then $\delta_m = 1$ and $\delta_A = -\alpha + \beta$. ■

Let us consider now the **resubstitution** [1], [2] for synchronous Boolean networks. Let $\mathcal{G}$, $\mathcal{J}$, $\mathcal{Q}$, and $\mathcal{R}$ be Boolean expressions. Then $\mathcal{J}$ is a **synchronous divisor** of $\mathcal{G}$ if $\exists r \geq 0$ such that $\mathcal{G} = \mathcal{J}^{(\cdot,+r)}\mathcal{Q} + \mathcal{R}$ and $\mathcal{J}^{(\cdot,+r)}\mathcal{Q} \neq \varnothing$. Note that the product $\mathcal{J}^{(\cdot,+r)}\mathcal{Q}$ may have the algebraic or Boolean flavor, as defined in [1]. Given two internal vertices v_i and v_j such that the expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{G}$, the resubstitution of v_j into v_i is the factoring of $\mathcal{G}$ as $j^{(\cdot,+r)}\mathcal{Q} + \mathcal{R}$. An algorithm for synchronous division was presented first in [13] and it will be described in detail later. Note again that the divisors defined in [1] are a subset of the synchronous divisors, and therefore, resubstitution with null retiming (i.e., $r = 0$) is equivalent to resubstitution in combinational logic. The resubstitution of a variable with nonzero retiming corresponds to adding one (or more) register between two gates to simplify the latter (Fig. 4).

When resubstituting v_j into v_i, the variation in literals can be

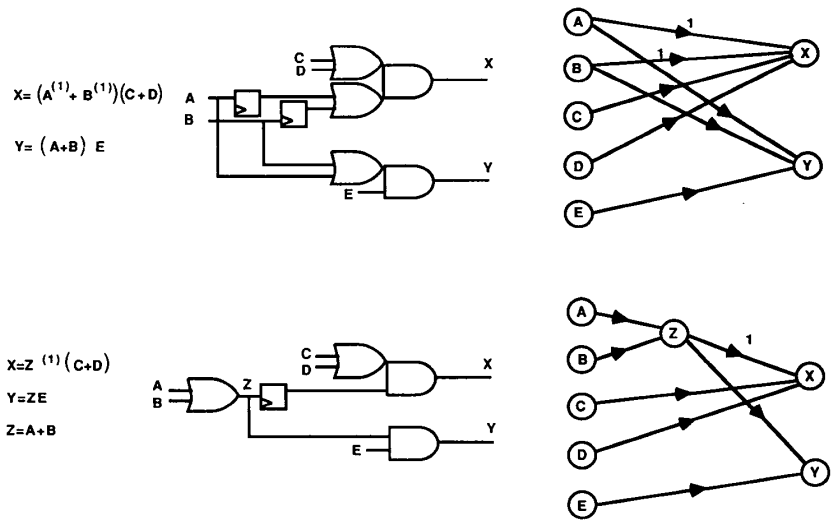


Fig. 5. Extraction of v_z.

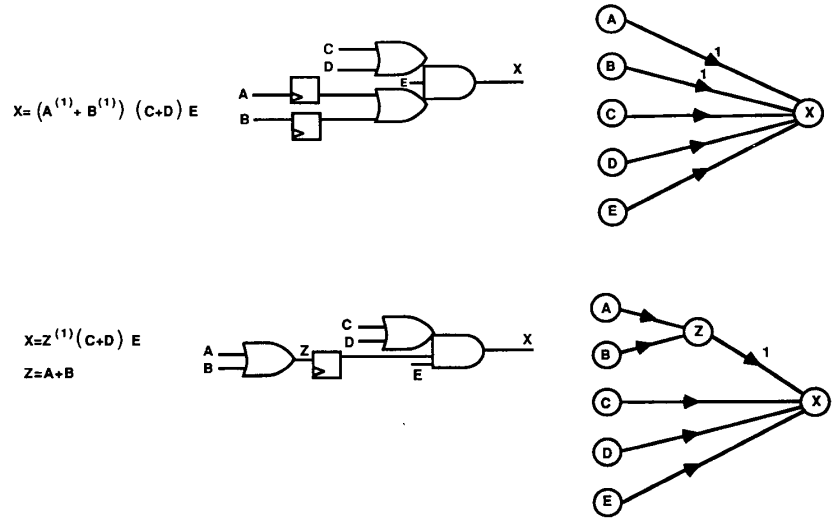


Fig. 6. Decomposition of v_x.

computed as $\delta_l = -n_{jl}(l_j - 1)$, where n_{jl} is the multiplicity of variable j in expression $\mathcal{G}$ [1], [2]. The number of registers in the network is affected only by resubstitutions across register boundaries (i.e., when $r > 0$).

Example: Consider the circuit of Fig. 4. The variation in the number of literals is: $\delta_l = -n_{xy}(l_x - 1) = -1(2 - 1) = -1$, i.e., one literal is saved. (Note that the original expression of y could be factored as $c(a^{(2)} + b^{(1)})$). Assume that $m_x = 0$ and that no additional delayed values of a and b are needed to gates other than those shown in Fig. 4. Then $\delta_m = -1$ and $\delta_A = -\alpha - \beta$. ■

The **extraction** of a common subexpression of expressions $\mathcal{G}$ 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 $\mathcal{G}$ and $\mathcal{J}$ and to the factoring of $\mathcal{G}$ and $\mathcal{J}$ in terms of the new variable l (Fig. 5). The local change in area due to extractions: $\delta_l = -n(l_l - 1) + l_l$, where usually $n = 2$ because vertex v_l is extracted from $n = 2$ other vertices. The number of registers in the network is affected only by extraction across register boundaries.

Example: Consider the circuit of Fig. 5. The variation in the number of literals is: $\delta_l = -2(l_z - 1) + l_z = -2(2 - 1) + 2 = 0$, i.e., the number of literals is constant. The variation in register is: $\delta_m = -1$, and therefore, $\delta_A = -\beta$. ■

There are different ways of decomposing a Boolean expression. In this paper we define **decomposition** of an expression $\mathcal{G}$ its replacement by the expression: $j^{(\cdot+r)}\mathcal{Q} + \mathcal{R}$, where j is a new variable, its corresponding expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{G}$ and $j^{(\cdot+r)}\mathcal{Q} \neq \varnothing$. The decomposition of a vertex v_i implies the addition to the network of vertex v_j (Fig. 6). Decomposition can be applied recursively to v_i and v_j.

Note that decomposition increases the number of literals δ_l.

For this reason, decomposition is used in combinational logic synthesis only to break the large expression that has no efficient implementation or to satisfy timing goals. However, decomposition in synchronous logic synthesis can lead to a reduction of the number of registers, and therefore, be beneficial for area reduction as well.

Example: Consider the circuit of Fig. 6. The variation in the number of literals is $\delta_l = +1$. The variation in register is: $\delta_m = -1$, and therefore, $\delta_A = \alpha - \beta$. ■

IV. Algorithms for Synchronous Logic Synthesis

Since optimizing synchronous Boolean networks is a difficult problem, heuristic optimization is achieved, as in combinational logic synthesis, by applying an operator to the network (i.e., iterating transformations of a given kind) until local optimality with respect to this operator is found. Then a different operator is applied.

The transformations presented in Section III can be used to optimize the circuit area (without/with cycle time constraints) or the circuit cycle time (without/with area constraints). Area optimization with a given transformation (e.g., elimination, resubstitution, etc.) can be achieved by using a greedy strategy in selecting the vertices that are the target of this transformation. The selected vertices are those such that the variation in area $\delta_A = \alpha\delta_l + \beta\delta_m$ is negative and minimal. A detailed description of these transformations is reported in [14].

We concentrate here on logic transformations that reduce the cycle time. For this reason, we present first an algorithm for constructing a feasible retiming for a given cycle time T, if one exists. The algorithm is based on the synchronous network model and supports the design of large synchronous networks. Then we present an algorithm for synchronous division. Eventually we present techniques for timing optimization using logic transformations across register boundaries.

4.1. An Algorithm for Finding a Feasible Retiming

We describe in this section an algorithm that can be used to construct a feasible retiming of a synchronous network for a given cycle time T. It is based on an algorithm described first in [10] and later in [11], but it is not so well known as the one presented in [8]. The original algorithm was not geared towards modeling Boolean networks: in particular multiple synchronous I/O's were not supported. In this paper we are concerned with networks with multiple I/O's, under the assumptions that all inputs are synchronous to the system clock. Such a model better conforms to synchronous digital circuits that need to be interconnected among each other.

The algorithm is iterative in nature. At each step, the vertices whose data ready time is larger than the required cycle time T are flagged and put in a temporary set M. The vertices of set M are retimed by $r = +1$. These steps are repeated until the network is timing feasible for T or procedure *exit* returns TRUE.

```
retime {
    for (k = 1;;k++) {
        Compute t_i for each vertex v_i ∈ V;
        M = {v_m | t_m > T};
        if (M = ∅)
            return(TRUE);
        else {
            if (exit)return (FALSE);
            Retime by 1 all vertices in M;
            if (M ∩ V^O) ≠ ∅) set-outputs;
        }
    }
}

set-outputs {
    Retime by 1 all primary output vertices not in M;
    S = {v ∈ V^G | ∃ a zero weight path from an input vertex
        to v};
    Retime by 1 all the input vertices and those in S;
}

exit {
    return (k ≥ |V|);
}.
```

This algorithm differs from the original one [10] by having a conditional call to the subroutine *set-outputs*. Let us consider first the analysis of the original algorithm and let us consider networks without multiple I/O's. In this case procedure *set-outputs* is never called. The following theorem applies to such networks.

Theorem 1: Given a cycle time T, algorithm *retime* returns TRUE if and only if a feasible retiming exists [10]. ■

Let us consider now synchronous Boolean networks with multiple I/O's. It can be easily shown that when the algorithm returns TRUE, a feasible retiming for the given cycle time T is constructed by the algorithm. Indeed, in this case all the data ready times are bounded by the cycle time T. When an output vertex is in set M, then all I/O vertices are retimed by $r = +1$, to preserve equivalence. Retiming an output vertex corresponds to delaying the corresponding signal by one cycle. Therefore, all other output vertices are retimed (to keep the output signal in phase with each other) and a synchronous delay is recovered by retiming the inputs vertices and an appropriate set S of internal vertices. This set S, possibly empty, includes those internal vertices connected to some input vertex by a zero-weight path. Retiming by $r = +1$ the input vertices and those in the set S guarantees that no negative label is introduced while retiming the I/O's. In addition, since $t_m > T$ implies $t_i > T\ \forall v_i \in DFO(v_m)$, then the retiming of a vertex implies the retiming of all the vertices on zero-weighted paths originating from it as well. Therefore, no negative weights (labels) can be introduced by retiming internal vertices. Therefore, I/O equivalence is preserved at each iteration of the algorithm.

Furthermore it can be shown that no feasible retiming exists when the algorithm returns FALSE.

Theorem 2: For any synchronous Boolean network described by $G(V, E, W)$ and a given cycle time T, algorithm *retime* returns TRUE if and only if a feasible retiming exists. ■

Proof: To prove the theorem, it is sufficient to note that running algorithm *retime* on any multiple I/O network $G(V, E, W)$ is equivalent to running the same algorithm on a modified network without I/O's. Consider a modified network obtained by merging the input and output vertices into a dummy vertex v_h, with $d_h = T$, and by adding one to the weights of all edges

incident to v_h. For any feasible retiming of both networks, the data ready time is the same for each pair of corresponding internal vertices. Indeed a retiming of the modified network cannot remove the synchronous register delays from the dummy vertex v_h to any vertex depending on a primary input, and therefore, the data ready time of these vertices is preserved. In addition, since any retiming of the modified network does not change the cycle weights in the corresponding graph [8], then all the I/O paths weights are preserved in the original network. Therefore, a feasible retiming of the modified network co-implies a feasible retiming of the original network. Consider now algorithm *retime*. The retiming of a primary output vertex in the original network corresponds to retiming v_h in the modified network, and therefore, to retiming all other primary output vertices. In turn, the retiming of v_h causes the retiming of all the vertices in the set S. Therefore, running algorithm *retime* on any multiple I/O network is equivalent to running the same algorithm on the corresponding modified network and the claim follows from Theorem 1. ■

The theorem shows that the existence of a feasible retiming can be computed in $O(|V||E|)$ time for general synchronous Boolean networks, because each of the $|V|$ iterations involves the computation of the data ready times, which can be done by topological sort ($O(E)$). In some cases, the algorithm can terminate earlier.

Theorem 3: If at any iteration of the algorithm, $\exists v_m \in M \cap S$ and v_m is a primary output, then no feasible retiming exists. ■

Proof: In this case, there is a zero weighted path from some input vertex to v_m and $t_m > T$. Since the path weight must be preserved, then t_m cannot be reduced. ■

This theorem provides an early exit condition which is incorporated into procedure *exit* of algorithm *retime*.

```
exit {
   if (k ≥ |V|) return (TRUE);
   S = {v ∈ V | ∃ a zero weight path from an input vertex to
      v};
   if ((M ∩ S ∩ V^O) ≠ ∅) return (TRUE);
   return (FALSE);
}.
```

Algorithm *retime* has two major advantages over the original retiming algorithm [8]. First, the description of a synchronous Boolean network structure in terms of a (sparse) graph suffices to implement the algorithm. This contrasts the requirements for the algorithm in [8], that needs two full square matrices of dimension $|V|$. Second, *retime* is an incremental algorithm, and so it can be applied in connection with network transformations that make small modifications to the network to check feasibility.

The algorithm requires the update of the data ready times at each iteration. Note that not all the data ready times need to be recomputed. Therefore, a selective-trace algorithm can be used to determine the vertices whose data ready times need to be updated. Let X represent the subset of vertices that are retimed at a given step of the algorithm. The algorithm starts by selecting the subset Y of the vertices in X whose direct fan-in set is not in X. These vertices represent gates connected to some inputs or to some register-outputs and whose data ready time is equal to their propagation delay. Then, the algorithm iteratively updates the data ready times of the vertices in the direct fan-out cone of these vertices. The data ready time of the remaining vertices need not to be updated.

```
update(X) {
   while (X ≠ ∅) {
      Y = {v ∈ X | DFI(v) ∩ X = ∅};
      compute data ready time for vertices in Y;
      X = X ∪ DFO(Y);
      X = X − Y;
   }
}.
```

4.2. Synchronous Division

Synchronous division is required to perform resubstitution across register boundaries, as described in Section III. Indeed, the resubstitution of a vertex v_j into vertex v_i, requires representing the expression at vertex v_i as $\mathcal{G} = j^{(\cdot,+r)}\mathcal{Q} + \mathcal{R}$. Therefore, $\mathcal{J}$ must be a synchronous divisor of $\mathcal{G}$. We consider here only algebraic division [1]. The condition that an expression is a synchronous divisor of another one is checked by routine *synchronous-divisors*, that iterates algebraic divisions. Algebraic division of the two expressions is performed by procedure *alg-div*, which is described in [1], [2].

```
synchronous-divisors(𝒢, 𝒥) {
   𝒬ℛ = ∅;
   𝒢𝒢 = expand(𝒢);
   for (r = 0;;r++) {
      𝒥𝒥 = expand(𝒥^(·,+r));
      If(exit(𝒢, 𝒥^(·,+r)))return
      𝒬ℛ = 𝒬ℛ ∪ alg-div(𝒢𝒢, 𝒥𝒥);
   }
}.
```

Algorithm *synchronous-divisors* operates as follows. Procedure *expand* replaces every variable with a nonzero label by a new variable. Therefore, expressions $\mathcal{GG}$ and $\mathcal{JJ}$ are polynomials that can be divided by algorithm *alg-div* [1], [2]. Procedure *exit* returns TRUE if any variable in $\mathcal{J}^{(\cdot,+r)}$ has a label larger than the maximum of the labels that the corresponding variable takes in $\mathcal{G}$. In this case, no nontrivial divisor can be found, because expression $\mathcal{JJ}$ contains a literal not in $\mathcal{GG}$, and therefore, $\mathcal{JJ}$ cannot divide $\mathcal{GG}$ [1], [2]. Clearly this condition is true for any value of r larger than the current index of the loop of the algorithm. Note that when both expressions $\mathcal{G}$ and $\mathcal{J}$ have no labels, then $\mathcal{GG} = \mathcal{G}$ and $\mathcal{JJ} = \mathcal{J}$, and the algorithm performs just the algebraic division as in [1], [2] and returns after one iteration.

The algorithm stores the quotient and the remainder of the division in $\mathcal{QR}$, which is initialized empty. Note that an expression $\mathcal{J}^{(\cdot,+r)}$ may divide an expression $\mathcal{G}$ for more than one value of r. Therefore, the algorithm stores all nontrivial quotients $\mathcal{Q}$ and remainders $\mathcal{R}$ in $\mathcal{QR}$. When multiple choices are possible, the resubstitution algorithm selects the most convenient divisor based on the timing (and/or area) estimates.

4.3. Algorithms for Cycle-Time Minimization

The problem of minimizing the cycle time T is approached as in the case of combinational logic [2], [9]. The strategy is to generate a sequence of networks that are timing feasible for de-

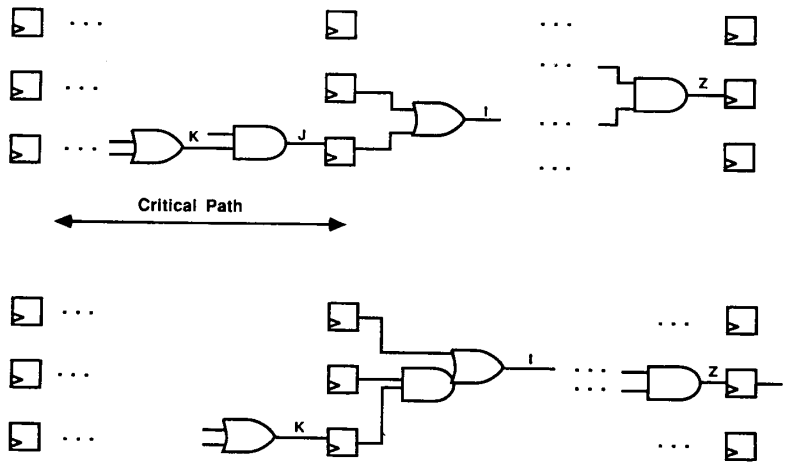


Fig. 7. Elimination at the head of a critical path.

creasing values of T. Transformations are applied to the critical vertices of each network in the sequence. A network is timing optimal with respect to a transformation when no further reduction of the cycle time can be achieved by applying the transformation.

A network can be made optimal with respect to retiming by running algorithm *retime* for decreasing values of T. In particular, Leiserson and Saxe suggested to compute the path propagation delays between all vertex pairs, and to binary search among these values for the minimum value for which *retime* returns TRUE [8]. While the computation of all-pair delays may be computationally expensive, a convenient heuristic to solve the problem is to decrease T by fixed increments, so that its value can be a practical choice for the cycle time.

A straightforward strategy for timing optimization is to alternate the search for a network that is timing optimal with respect to retiming (i.e., that optimizes the register position) with a set of transformation on the combinational portion of the circuits (obtained by temporarily removing the registers). Such an approach has the appeal of leveraging efficient algorithms for combinational logic techniques, such as those used in MIS [2]. Unfortunately this approach falls short in a few cases.

First note that a network may be optimal with respect to the available transformations, but it may be improved by the combined application of two (or more) transformations. In particular, a combinational logic transformation may lead to a nontiming feasible network for which there exists a feasible retiming. The drawbacks of this approach are the expanded search space and the need of storing temporarily the network after the transformation. A more efficient approach is to use transformations across register boundaries, that can be thought of as a combination of retiming of a vertex and a combinational transformation in a single step.

We would like to comment now on the advantages of the transformations across register boundaries that are useful for timing optimization. Let us assume that the network is optimal with respect to retiming (by using the *retime* algorithm for decreasing values of T) and with respect to the other transformations within register boundaries (as described in [9] and in [2]). We assume that T is the minimum cycle for which the network is timing feasible and we address the problem of reducing it by attempting transformations across register boundaries. Also in this case, logic transformations (as described in Section III) are applied to the critical vertices of the synchronous Boolean network, that is made timing feasible for decreasing values of T. The section of candidate vertices for the transformations is guided by the following considerations.

Let us consider first elimination. In particular, we consider as candidates for elimination the critical vertices whose corresponding gate is connected to a register, i.e., at the head of a critical path (Fig. 7). Let us assume, for the sake of simplicity that there is only one such candidate, say v_j and that it is critical (i.e., its slack $s_j = 0$ or equivalently its data ready time $t_j = T$). The elimination of such a vertex shortens the critical path and it is beneficial if no other longer critical path is introduced in the circuit. Therefore, to verify the feasibility of the elimination of a candidate vertex v_j, we must consider the increase of data ready time of each vertex $v_i \in FO(v_j)$. The data ready time at these vertices may increase, because the corresponding propagation delay may increase, being a monotonic function of the number of literals that are increased by elimination. If such increases are all strictly bound by the corresponding slacks, then the elimination is accepted because there is a cycle time $T' < T$ for which the network is timing feasible after the elimination.

Let us now consider resubstitution across register boundaries of two vertices, say v_j into v_i. In this case, the data ready time t_i may decrease and t_j remains constant. Indeed, the number of literals l_i at vertex v_i is decreased by the resubstitution, and its corresponding propagation delay may decrease. Thus candidates for resubstitution can be searched among critical vertices that are the tail of a critical path (i.e., v_i is a tail of a critical path and $v_j \in FI(v_i)$). Candidates are selected to minimize locally the cycle time T. Since an upper bound on the decrease of T is the variation in propagation delay d_i, this variation can be used to rank candidates. Consider, for example, the circuit of Fig. 8. The critical path has as a tail vertex v_i. The resubstitution of vertex v_j into v_i decreases the propagation delay d_i, and therefore, reduces the data ready time of the vertex at the head of the critical path. If the maximum value of the data ready time is attained at that vertex only, then the cycle time T can be reduced.

Similar considerations apply to decomposition across register boundaries. Suppose for example that we decompose an expression $\mathfrak{I}$ as $j^{(\cdot+\prime)}\mathfrak{Q} + \mathfrak{R}$. Then the delay through v_i may de-

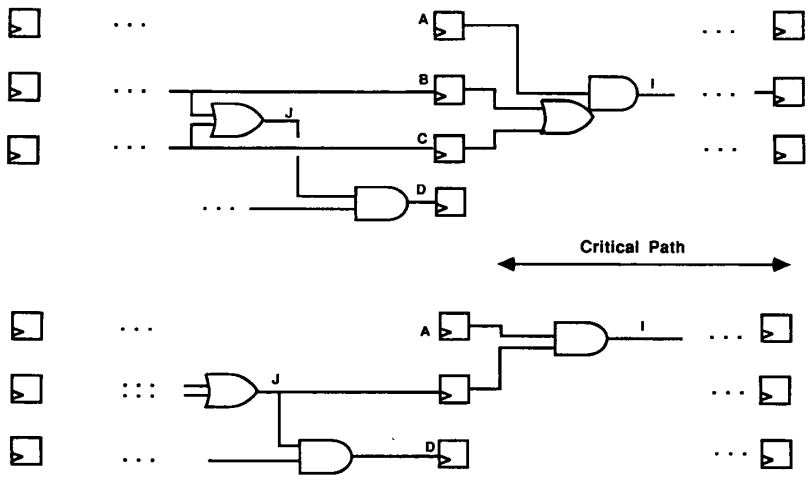


Fig. 8. Resubstitution at the tail of a critical path.

crease. Therefore, the candidates for decomposition are still the tail of critical paths, and the variation in d_i can still be used to rank candidates. However, in this case, contrary to resubstitution, it is important to verify that no other critical paths are introduced with v_j as a head. This check can be done by verifying that d_j is bounded by the slack of each vertex $v_k \in DFI(v_j)$.

V. Experiments on Benchmark Circuits and Results

Even though retiming techniques have been known for a decade, we are unaware of reports of its application to logic synthesis. Experimental results with an implementation of the timing optimization algorithms based on retiming have shown that the quality of the results depends on: i) the logic depth of a circuit; ii) the delay model; and iii) the circuit type. For this reason it is hard to assess the value of these techniques as a whole, and our analysis is more articulate.

The logic depth of a circuit can be measured by the average number of logic stages (with bounded number of inputs) between two register boundaries. It is obvious (and confirmed by the experiments) that the likelihood of changing the logic by retiming in a shallow circuit is low. On the other hand, deep circuits (i.e., with many logic stages between registers) are more amenable to be retimed because there exist a larger set of equivalent networks that can be obtained by retiming. The depth of the network can be increased by performing decomposition and extraction prior to retiming. Unfortunately these operations change the timing performance of the network, making it difficult to assess the intrinsic gain due to retiming.

The delay model also affects the results. Since changing the propagation delay through a gate is equivalent to changing a datum of the problem, different final results are obtained for different models. Therefore, the quality of the results has to be calibrated with the delay models. In our experiments, we used the following formula for the propagation delay: $d = 0.8 + 0.1N + 0.1P$, where N and P are the maximum number of N and P transistors in series, respectively. The area model, i.e., the choice of the coefficients α and β that represent the relative area cost of literals and registers, affects the acceptance of the transformations when minimal area implementations are sought for. In the case of unconstrained timing optimization, that is, in the case dealt with here, the area coefficients affect the total area estimate. In our experiments we have chosen $\alpha = 1$ and $\beta = 8$.

The type of circuit being optimized is also important. Some sequential circuits used as benchmarks are finite state machines (FSM's). Such circuits are characterized by having cycles in the network graph of weight equal to one. They are also characterized by shallow logic expressions, because they are in general derived from two-level representations. While the depth of the circuit can still be increased by logic transformations, FSM's are still hard to retime because their state equations are relatively simple and often their critical path is an I/O path with weight equal to zero. Since the weight on that path cannot be changed by retiming, the critical I/O path is a lower bound on the cycle time.

Pipelined circuits, when deep enough, can be in general improved by retiming. An interesting class of circuits are those that are synthesized automatically from behavioral descriptions and that have merged data path and control. These circuits often show longer critical paths in the data path portion than in the control part, and therefore, benefit from an uniform register redistribution.

The algorithms have been tested on benchmark circuits. In particular, the examples Ex1–7 are derived from the MCNC FSM examples Ex1-7. Since these benchmarks are provided in two-level forms, the starting points for our experiments have been derived by means of an arbitrary, but fixed, sequence of logic synthesis steps. Area and timing variations are computed from these starting points. The examples Ex8–12 were synthesized directly from high-level descriptions with no other manipulation at the logic level. In particular Ex8 and Ex9 are the phase decoder and the receiver of the *daio* chip [15], Ex10–Ex12 are benchmarks for high-level synthesis (*gcd*, *length* and *proadd*). Ex13 and Ex14 are two ALU's, derived from the MCNC multiple level benchmark circuits *Alu*2 and *Alu*4, respectively, by adding output registers.

Table I reports the overall results of reducing the cycle time. The runtimes are in the order of a few seconds on a DEC-station 3100 for the largest circuit. The following transformations have been applied to the benchmark circuits: *decomposition*, *retiming*, *elimination*, and *resubstitution*. Other transformations on the combinational portion of the circuits (as performed by [2]),

TABLE I
COMPARATIVE TIMING AND AREA VARIATION

		Original Circuit		Optimized Circuit		Absolute Variation		Relative Variation	
Example	Type	Time	Area	Time	Area	Time	Area	Time	Area
Ex1	FSM	16.5	690	16.5	588	0.0	−102	0.0%	−14.0%
Ex2	FSM	11.1	421	10.7	439	−0.4	+18	−3.6%	+2.3%
Ex3	FSM	7.9	218	7.7	282	−0.2	+64	−3.9%	+29.0%
Ex4	FSM	12.1	271	11.3	253	−0.8	−18	−6.6%	−6.6%
Ex5	FSM	6.9	202	6.9	199	0.0	−3	0.0%	−1.4%
Ex6	FSM	9.9	264	8.9	264	−1.0	0	−1.0%	0.0%
Ex7	FSM	8.6	231	8.5	225	−0.1	−6	−1.1%	−2.5%
Ex8	HL	20.9	2491	19.7	2643	−1.2	+152	−5.7%	+6.1%
Ex9	HL	24.6	2266	21.4	2610	−3.2	+344	−13.0%	+15.1%
Ex10	HL	35.0	1036	33.5	1418	−1.5	+382	−4.2%	+36.8%
Ex11	HL	8.5	213	6.5	188	−2.0	−25	−23.5%	−11.7%
Ex12	HL	11.4	211	6.7	222	−4.7	+11	−41.2%	+5.2%
Ex13	PIPE	17.8	662	9.7	1078	−8.1	+416	−45.5%	+62.8%
Ex14	PIPE	22.1	1165	12.2	1949	−9.9	+784	−44.8%	+67.2%

have not been applied for the sake of evaluating the limitations of the new algorithms.

In the case of the FSM examples, the algorithms achieve an average decrease in cycle time of 3.4% with a negligible average area variation. The critical paths of circuits Ex1 and Ex5 are fully combinational I/O paths, making the synchronous techniques useless to speedup the circuit. For circuits synthesized from high-level representations, the algorithms achieve an average speedup up of 13.9% and an average increase in area of 13.8%. A larger variation is achieved for the last circuits, where retiming creates two pipeline stages, as expected. The average speedup is 45.1% and an average increase in area is 65.6%. Note that the increased area cost is mainly due to an increase in the number of registers.

VI. Concluding Remarks and Future Directions

This paper has presented a new approach to the optimal logic synthesis of digital synchronous circuits, based on the concurrent optimization of the circuit equations and the register positions. This method, which combines retiming techniques with network restructuring operations, can achieve, in principle, results that are at least as good as those obtained by other logic synthesis approaches that separate the combinational logic from the registers. Indeed, standard combinational logic synthesis techniques are compatible with the new algorithms and can be integrated in the same framework. The experimentation of the algorithms on benchmark circuits has given some preliminary encouraging results. Significant reduction of the cycle time has been achieved in the case of networks synthesized from high-level descriptions and in the case of pipelined circuits, while marginal improvements have been measured in the case of FSM circuits. The overall average speedup is about 16% at the expenses of an area increase of 19.5%.

This research has shown the feasibility of approaching synchronous sequential logic design from a new perspective, based on a stepwise refinement of a structural logic representation, in terms of an interconnection of components. We think that the problem of selecting the optimal number, type, and interconnection of registers in sequential logic design can be modeled by optimizing synchronous Boolean networks. Therefore, new approaches to classical problems, such as the state assignment problem, could be developed with this setting. However, several problems are not yet solved and deserve further research. First, there should be a study of the appropriate set of logic transformations for synchronous sequential logic, with particular reference to the possibility of reaching all the possible circuit configurations with equivalent I/O behavior. Second, there should be a search of efficient retiming techniques supporting an extended propagation delay model with explicit fan-out dependency as well as satisfying both upper and lower bounds on propagation delays. Such an extension would support logic synthesis techniques for circuit designs with gated latches and with non-negligible clock skew. Third, there should be a study of technology mapping techniques that take advantage of the information contained in the synchronous Boolean network and of the application of retiming techniques to mapped networks.

Acknowledgment

The author would like to thank Thierry Klein, Roger Yip, and Luis Stevens who worked on this project at Stanford University. They developed some of the ideas of the retiming, synchronous elimination, and synchronous resubstitution algorithms and implemented them. The author would also like to acknowledge the stimulating discussions with Andrew Fox, Michiel Ligthart, and Frederic Mailhot at Stanford and those with Sharad Malik and Ellen Sentovich at U. C. Berkeley. He would also like to thank the anonymous referees, whose comments were useful in improving the manuscript.

References

[1] R. Brayton, "Algorithm for multilevel synthesis and optimization," in G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Ed., *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*. Martinus Nijhoff, 1987.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1062-1081, Nov. 1987.

[3] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan, "LSS: A system for production logic synthesis," *IBM J. Res. Develop.*, vol. 28, no. 5, pp. 537-545, Sept. 1984.

[4] K. Bartlett, W. Cohen, A. De Geus, and G. Hachtel, "Synthesis and optimization of multilevel logic under timing constraints,"

IEEE Trans. Computer-Aided Design, CAD-5, pp. 582–596, Oct. 1986.

[5] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method—Design of logic networks based on permissible functions," *IEEE Trans. Comput.*, vol. 38, pp. 1404–1424, Oct. 1989.

[6] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," in *Proc. Hawaii Int. Conf. on System Sciences*, Kona, HI, vol. 1, Jan. 1990, pp. 397–406.

[7] G. Saucier, M. Crastes de Paulet, and P. Sicard, "ASYL: A rule-based system for controller synthesis," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp.1088–1097, Nov. 1987.

[8] C. Leiserson, F. Rose, and J. Saxe "Optimizing synchronous circuitry by retiming," in R. Bryant, Ed., *Third Caltech Conference on VLSI*. Computer Science 1983, pp. 87–116.

[9] G. De Micheli, "Performance-oriented synthesis in the Yorktown silicon compiler," *IEEE Computer-Aided Design*, vol. CAD-6, pp. 751–765, Sept. 1987.

[10] J. Saxe "Decomposable searching problems and circuit optimization by retiming: Two studies in general transformations of computational structures," Ph.D. dissertation, Dept. Comput. Sci., Carnegie Mellon Univ., 1985.

[11] C. Leiserson and J. Saxe, "Retiming synchronous circuitry," Internal Rep. MIT/LCS/TM-372, 1988.

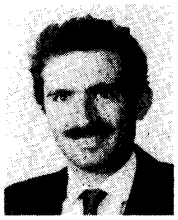
[12] ——, "Optimizing synchronous systems," *J. VLSI Computer Syst.*, vol 1, no. 1, pp. 41–67, Spring 1983.

[13] G. De Micheli and T. Klein, "Algorithms for synchronous logic synthesis," in *Proc. Int. Symp. on Circuits and Systems*, Portland, OR, pp. 756–761, May 1989.

[14] G. De Micheli and R. Yip, "Logic transformations for synchronous logic synthesis," in *Proc. Hawaii Int. Conf. on System Sciences*, Kona, HI, vol. 1, Jan. 1990, pp. 407–416.

[15] M. Ligthart, A. Bechtolsheim, G. De Micheli, and A. El Gamal "Design of a digital audio input output chip," in *Proc. Custom Integrated Circuit Conf.*, San Diego, May 1989, pp. 15.1.1–15.1.6.

*



Giovanni De Micheli (S'79–M'83–SM'89) received the Dr. Eng. degree in nuclear engineering from the Politecnico di Milano, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He has held positions at the Department of Electronics, Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, FL. From 1984 to 1986, he was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he was Project Leader of the Design Automation Workstation Group. Presently, he is an Associate Professor of Electrical Engineering and Computer Science at Stanford University. 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 the co-editor of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation* (Martinus Nijhoff, 1987).

Dr. De Micheli is a member of the editorial board of *IEEE Design and Test* magazine. He was the Technical and General Chairman of the International Conference on Computer Design in 1988 and 1989, respectively.