Synthesis of High-Performance Digital Circuits:
Logic Transformations for Cycle-Time Reduction of Synchronous Circuits

Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

We review in this paper different techniques to enhance the circuit performance of synchronous digital circuits. In particular we concentrate on the mechanisms that can be used in a global framework for performance optimization. We survey logic transformations that operate on structural models of synchronous networks, described in terms of interconnections of logic gates and registers. These approaches are alternative to traditional sequential logic optimization algorithms based on state transition diagrams.

1 Introduction

Logic synthesis and optimization techniques have been used successfully for product-level design of digital circuits in the last decade. Nevertheless only some digital design problems have been solved satisfactorily by the use of Computer-Aided Design (CAD) tools. In particular, *high-performance* digital design is still often hand-crafted, due to the enormous importance of tuning the circuit implementation to achieve top performance with a given circuit technology.

Logic synthesis techniques have been striving to optimize three figures of merit of digital designs: *area*, *testability* and *performance*. The three issues are extremely complex because of the intractability of the underlying decision problems. For this reason, rule-based systems and heuristic algorithms have been used in this perspective.

Combinational circuit optimization has reached a certain maturity today. Most problems are understood, even though better solution methods are continuously searched for. Sequential circuit optimization is still in its infancy due to the additional complexity of handling registers and feedback connections. Recently, much attention has been devoted to *asynchronous* circuits, because interface asynchronous circuits are often a serious bottlenecks in digital design.

Here we consider *synchronous* circuits, because most digital designs have synchronous operation. This correlates to a relatively easier design methodology. Synchronous logic circuits are interconnections of 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. We call such a model a *structural* view of a synchronous circuits. Alternatively, the state transitions may be described in terms of tables or diagrams. We refer to this models as *behavioral views*.

Historically, sequential logic synthesis has been studied on behavioral models. Synthesis and optimization relates then to solving classical problems, such as state minimization and state assignment [1] [16]. In this paper, we report on *performance-oriented* optimization based on an iterative refinement of a circuit. We consider structural views, because they can be more easily related to timing models for delay evaluation than the corresponding behavioral models. We assume that the original structural model, that we would like to optimize, is either a human-designed schematic, or a netlist synthesized by high-level synthesis tools or by classical sequential synthesis programs.

We present here logic transformations that aim at reducing the *cycle-time* of synchronous circuits, by refining structural descriptions. 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 and/or removing registers. We concentrate on those transformations that are specific to synchronous circuits and that can be embedded in CAD systems for performance-oriented synthesis.

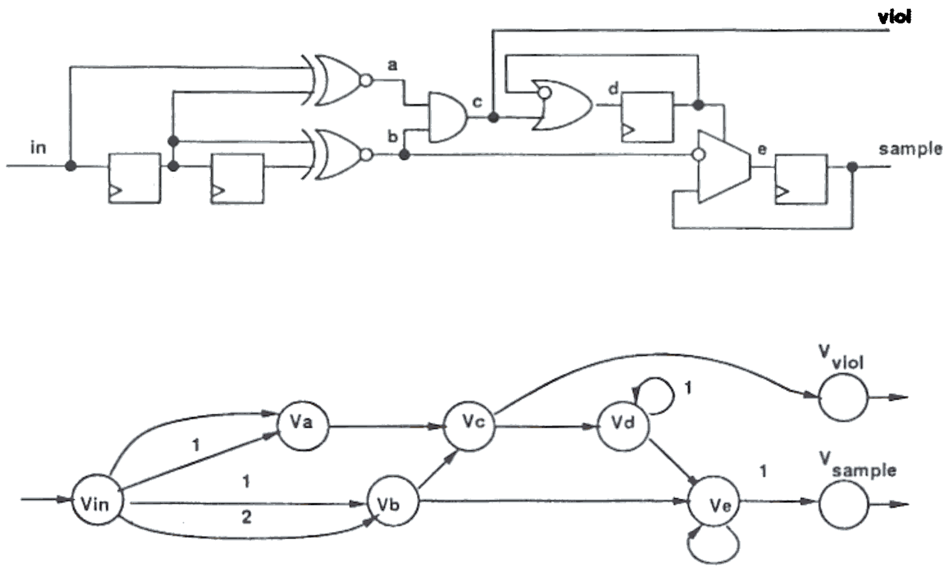

Figure Synchronous Boolean Network and its representation.

2 Circuit model.

We consider structural models of digital circuits. Such circuits can be specified by an interconnection of combinational logic gates and clocked registers. For the sake of simplicity, we assume first that all the 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 fall in this class. We assume, again for the sake of simplicity, that the clock skew, the register setup, hold and propagation times are negligible.

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 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. Labels are represented here in two equivalent ways: parenthesized superscripts and subscripts to which a current index n is added. For example, variable x with label $k = 2$ is denoted as $x^{(2)}$ or x_{n+2}, while the unlabeled variable is denoted by x or x_n. Each Boolean equation has an unlabeled variable (i.e. with zero-valued label) as left term and a Boolean expression as 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 $\mathcal{I}$ the Boolean expression associated to variable i.

The network is modeled by the *synchronous network graph*, that is a directed weighted graph $G(V, E, W)$, 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 representations is shown in Figure 1. Note that other representations of synchronous networks have been proposed [6], where variables and vertices have been associated also to nets of the circuits. This extended notation is not used here, because not necessary for the level of detail of the topics dealt with in this paper.

In general, a synchronous Boolean 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. The *latency* of a pipeline network is the product of the cycle-time times the I/O path weight. Note that the combinational Boolean network (without synchronous registers) introduced by Brayton [4] is just a special case of the synchronous Boolean network that is acyclic and whose labels are all zeroes.

The *fanin* (*fanout*) set of a vertex v_i is the subset of vertices that are tail (head) of an edge whose head (tail) is v_i and it is denoted by $FI(v_i)$ ($FO(v_i)$). We associate a *propagation delay* to each vertex and to each path. Note that a correct computation should include *false paths* detection [13]; otherwise only upper-bounds on delay can be computed and these can be misleading at times. Note also that the model can be extended to the use of transparent latches and multiple clock phases. Recently Sakallah [15] proposed methods for verifying the correctness of such networks as well as computing the minimum cycle time.

3 Algorithms for cycle-time minimization.

Some logic synthesis systems deal with cycle-time minimization by considering sequential circuits as an interconnection of a combinational logic component and registers. 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.

It is the purpose of this paper to survey those techniques that exploit the particular nature of sequential circuits. Therefore we do not report here on performance-oriented synthesis techniques for combinational circuits, that can still play an important role in optimizing sequential ones. We refer the reader to [17] and [13] for details.

We concentrate in this paper on the *fundamental mechanisms* for circuit transformations. The transformations themselves can be driven by an overall performance optimization algorithm, that selects the transformation type and the targets. We refer the reader to [2, 4, 8, 10, 14] for details and examples of overall optimization strategies.

3.1 Retiming.

The original *retiming* algorithm was presented first by Leiserson and Saxe [11]. The circuit cycle-time can be minimized by moving the register position. While *retiming* a circuit, the combinational component is not modified. Therefore the approach is orthogonal to performance-optimization by combinational logic speed-up.

Leiserson showed that the minimum clock period corresponds to some path delay between a pair of vertices of the synchronous Boolean network. Therefore the search for an optimum cycle-time can be reduced to 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 weight on each edge is increased by the retiming of its head and decreased by the retiming of its tail.

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 [11], 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 Boolean networks. The assumption of constant gate delay is invalidated by the fact that gate fanouts change as registers move. The total number of registers (that relates to circuit area and power) cannot be bounded during cycle-time optimization. Therefore, a performance-optimal implementation may have an 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 [3].

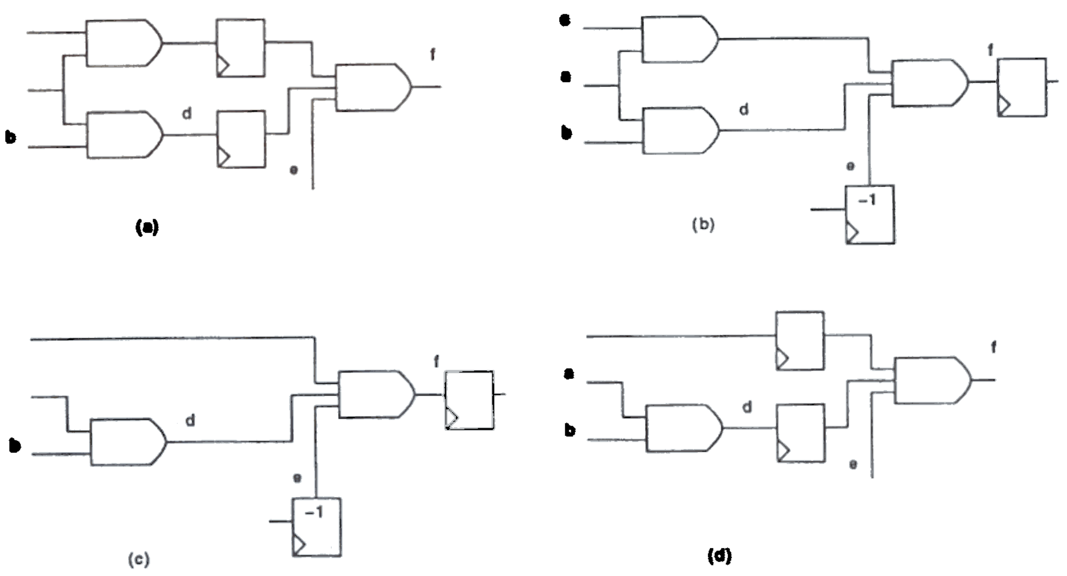

Figure 2: Example of peripheral retiming [12]. (a) Original circuit. (b) Peripherally retimed circuit with time borrowing. (c) Resynthesis. (d) Return of borrowed time.

3.2 Peripheral retiming.

Peripheral retiming is a novel technique introduced by Malik [12] to leverage combinational logic synthesis as much as possible in sequential circuit design. Peripheral retiming can be thought as of 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 speedup algorithm [17]) and to return later the registers to the 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, for example by using the *sliding window* paradigm [12]. An important degree of freedom of peripheral retiming is that edges crossing the periphery can have temporarily negative weights. This corresponds to borrow some time from the environment (outside the periphery), to extract the registers.

Malik showed also the necessary and sufficient conditions on a circuit for the existence of a peripheral retiming. When considering any I/O pair ($v_i \in V^I, v_j \in V^O$), no two paths $v_i, \ldots, v_j$ must differ in path weights and for any I/O path $v_i, \ldots, v_j$ the weight must equal the sum of two integers $\alpha_i + \beta_j$ associated with v_i and v_j. Obviously pipelined networks satisfy the assumptions for peripheral retiming.

An example of peripheral retiming is given in Figure 2, where the conditions for peripheral retiming are satisfied. Thus, all registers can be extracted from the region. To accomplish this, time is borrowed from input e, by introducing a temporary negative synchronous delay. The extraction of the registers allows a tool to resynthesize the corresponding combinational circuit. A redundant AND gate is removed. Then, retiming is applied to return the registers into the circuit. Note that the peripheral retiming conditions guarantee that negative synchronous delays can always be removed, so that the final implementation is a feasible circuit.

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.

Brglez [5] proposed a way of partitioning synchronous circuits into sub-networks, called *consistent corollae*, defined on the basis of signal reconvergence. He showed that such corollae satisfy the assumption of peripheral retiming. Then he 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 Boolean 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 shown in Figures 1 and 5. In this case, circuit optimization can be performed by combining locally retiming

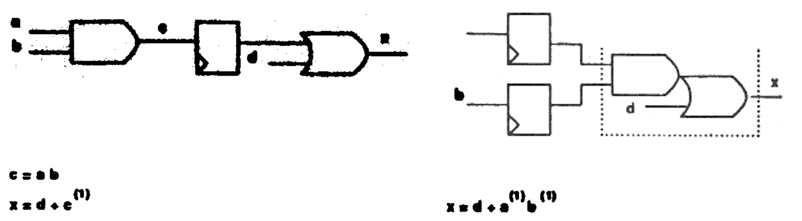

Figure 3: Example of synchronous elimination.

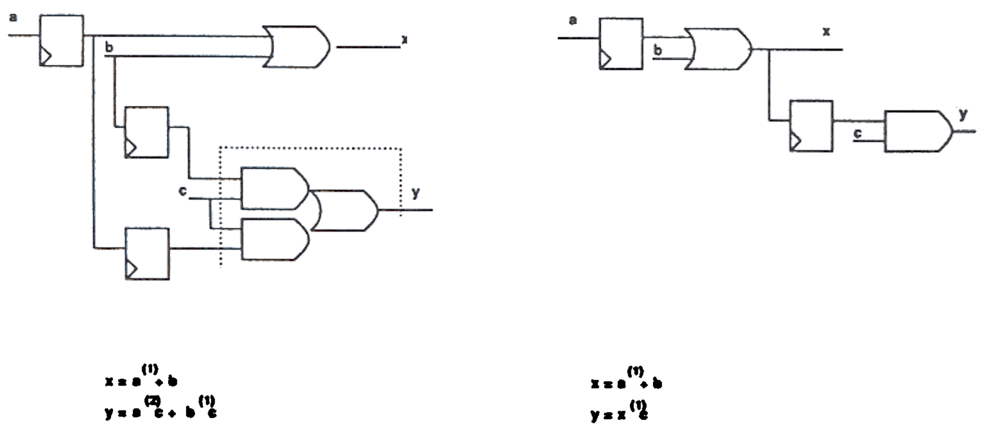

Figure 4: Example of synchronous resubstitution.

and combinational logic transformations. We call *synchronous algebraic transformations* the extensions of the algebraic transformations for combinational logic [4] that incorporate local retiming. We call here 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 using the operator $R^k(\cdot)$.

Examples of synchronous algebraic transformations are synchronous elimination, resubstitution, extraction and decomposition [9]. They are extension of the corresponding combinational transformations [4].

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 $\mathcal{I}$ 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 [4]. 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 Figure 3, where variable c has been eliminated. This corresponds to the merging of the AND and OR gates into a complex gate, and to the shift of the registers to its inputs.

Let us consider now *resubstitution* [4] for synchronous Boolean networks. Let $\mathcal{I}, \mathcal{J}, \mathcal{Q}$ and $\mathcal{R}$ be Boolean expressions. Then $\mathcal{J}$ is a *synchronous divisor* of $\mathcal{I}$ if $\exists k \geq 0$ such that $\mathcal{I} = R^k(\mathcal{J})\mathcal{Q} + \mathcal{R}$ and $R^k(\mathcal{J})\mathcal{Q} \neq \emptyset$. Given two internal vertices v_i and v_j such that the expression $\mathcal{J}$ is a synchronous divisor of $\mathcal{I}$, the resubstitution of v_j into v_i is the factoring of $\mathcal{I}$ as $R^k(j)\mathcal{Q} + \mathcal{R}$. An algorithm for synchronous division was presented in [9]. Note again that the divisors defined in [4] are a subset of the synchronous divisors and therefore resubstitution with null retiming (i.e. $k = 0$) is equivalent to resubstitution in combinational logic. The resubstitution 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 Figure 4. The complex gate corresponding to variable y is simplified by using variable x delayed by one, i.e. $R^1(x)$.

The *extraction* of a common sub-expression of expressions $\mathcal{I}$ 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 $\mathcal{I}$ and $\mathcal{J}$ and to the factoring of $\mathcal{I}$ and $\mathcal{J}$ in terms of the new variable l. Similarly, the *decomposition* of an expression $\mathcal{I}$ 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 $\mathcal{I}$, k is an integer and $R^k(j)\mathcal{Q} \neq \emptyset$. 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 resubstitution to gates that are tails of critical paths. In both cases, often such transformations are 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. 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 spell 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 [6].

The most simple case for *don't care* computation is the one of pipeline networks, where the effect of registers can be ignored. In other words, such networks can always undergo peripheral retiming, and all registers be moved to the circuit inputs or outputs as far as the *don't care* computation is concerned. Therefore, the *don't care* evaluation is as complex as in the combinational case. The same applies to the *don't care* set computation within a corolla. In the general case, the evaluation of the *don't care* set is more involved, due to the reconvergence of signals, possibly with different synchronous delays.

There is an additional complication in using the *don't care* conditions as degrees of freedom to perform Boolean transformations. For combinational circuits, *don't care* conditions can be represented as sets (and in particular as sum of products of cubes). Each element of the set represents an independent condition, and the *don't care* ensemble represents all the degrees of freedom for optimization. In synchronous circuit, the individual *don't care* conditions are correlated and therefore the overall degrees of freedom for optimization cannot be simply described by a set.

This remarkable property does not necessarily stem from feedback in the network. It is present also in definite networks and it is due to the existence of reconverging paths with different synchronous delays. Consider for example the circuit of Figure 5. It can easily be verified that the inverter driving the variable y can be replaced by a simple interconnection, i.e. that the function $g(x) = x'$ can be replaced by $f(x) = x$. Since there are two I/O paths with different weight, no peripheral retiming operation is possible on the circuit. It is also interesting to observe that the inverter can be replaced even though there are no independent *don't care* conditions associated to it. To check this, it suffices to observe that any *don't care* condition on y would result in the possibility of replacing the inverter with a constant 1 or 0, which is clearly incorrect.

Damiani [7] proposed a formulation that allows us to capture the degrees of freedom for the optimization of a subnetwork embedded in a synchronous system. In this approach, it is necessary to fully capture the terminal specifications imposed on that subnetwork. In the synchronous case, the most general terminal specifications are represented by the set of its possible *execution traces*, where a trace is defined as a pair of input/output sequences. Here, due to the limited space, we will show the approach on the simple example of Figure 5.

In the circuit in Figure 5, we seek to replace the input inverter by a simpler function, generating the intermediate signal y. The replacement is possible as long as the input/output behavior of the whole network is unaffected. The desired input/output behavior for the network is $z_n = x'_n \oplus x'_{n-1}$. The primary output z can be expressed in terms of the internal signal y (to be re-synthesized) as $z_n = y_n \oplus y_{n-1}$. The signal y must therefore satisfy the constraint: $y_n \oplus y_{n-1} = x'_n \oplus x'_{n-1}, \quad \forall n \geq 0$. The above equation represents the constraint on the execution traces by the circuit replacing the inverter. It can be rewritten as $(x'_n \oplus x'_{n-1})\overline{\oplus}(y_n \oplus y_{n-1}) = 1$ and it is called a *synchronous recurrence equation..*

It is worth remarking that for any given input sequence $x(\cdot)$, there exist more than one output sequence $y(\cdot)$ that satisfy the equation. Two possible solutions are

$$\begin{array}{ll} y_{-1} = x_{-1}; & y_{-1} = 0 \\ y_n = x_n \ \forall n \geq 0; & y_n = x_n \oplus x_{n-1} \oplus y_{n-1} \ \forall n \geq 0 \end{array}$$

The first solution corresponds to replacing the inverter by a wire and it is shown in Figure 5 (b). The second, shown in Figure 5 (c), relates to a more complex circuit with feedback and it is obtained by adding y_{n-1} to both terms of the equation. The assignments of y_{-1} correspond to the assignment of the *initial conditions* for the subcircuit.

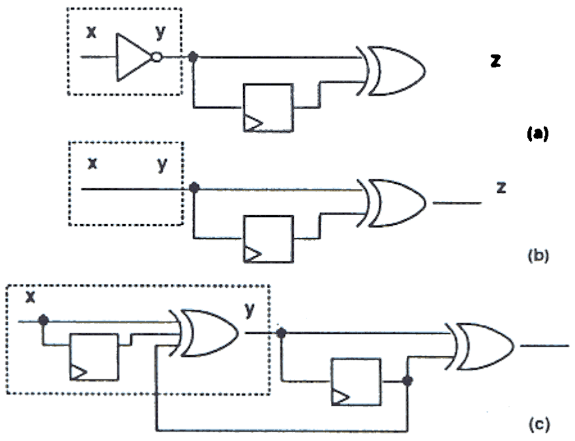

Figure 5: (a) Example of a non-retimable but optimizable circuit. (b) Simplified circuit implementation. (c) Other circuit implementation.

In the general case, the problem can be formulated by representing the degrees of freedom as a constraint equation, that relates the variable associated to the subnetwork to be optimized to its input variables. Note that variables are labeled to denote the time dependency. Such a constraint equation is a synchronous recurrence equation and it describes implicitly the *don't care* conditions.

A solution to a synchronous recurrence equation is a logic function f, that can replace the subnetwork. Therefore the Boolean optimization problem can be seen as the synthesis of an appropriate function, that satisfies the boundary constraints set by the synchronous recurrence equation.

While this function can be general in nature, a reasonable simplification is to assume that is definite, i.e. that f is a combinational function, whose support are the (possibly delayed) variables associated to the network inputs. As an example, for the circuit of Figure 5 (a), we would be seeking solutions of the type shown in Figure 5 (b), and not in Figure 5 (c).

Since the function f is the unknown of the problem, it can be represented by its *truth table*, where the entries of the truth table are *coefficients* to be determined. For the previous example, we seek a function $f(x_n, x_{n-1})$ of minimum cost that can replace the inverter. The function is entirely described by its truth table, represented in Table 1. The coefficients f_0, f_1, f_2, f_3 represent the unknowns of the problem. Feasible solutions (in terms of definite networks) are represented by $f_0 = 1, f_1 = 1, f_2 = 0, f_3 = 0$ (corresponding to the original inverter) and by $f_0 = 0, f_1 = 0, f_2 = 1, f_3 = 1$ (corresponding to the simple interconnection).

x_n	x_{n-1}	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

Table 1: Tabular representation of an unknown function $f(x_n, x_{n-1})$.

By expressing the synchronous recurrence equation in terms of the coefficients of the truth table, it is possible to determine a set of clauses that fully describes the problem. For the previous problem, the synchronous recurrence equation is $(x'_n \oplus x'_{n-1})\overline{\oplus}(y_n \oplus y_{n-1}) = 1$. Corresponding any assignment of (x_n, x_{n-1}, x_{n-2}), we can derive the values of y_n, y_{n-1} that satisfy the recurrence relation. These are tabulated in the second column of Table 2. For example, for the assignment $x_n = 0, x_{n-1} = 1$ (second row of Table 2), the relation reduces to $y_n \oplus y_{n-1} = 1$, that is true for $(y_n = 0, y_{n-1} = 1)$ or $(y_n = 1, y_{n-1} = 0)$.

We can now re-express the constraints on y_n, y_{n-1} in terms of the coefficients. For the relation table of Table 2, corresponding to the assignment $(x_n, x_{n-1}, x_{n-2}) = (0, 0, 1)$, the possible assignments for (y_n, y_{n-1}) are either $(0, 0)$ or $(1, 1)$, i.e. it must be $(y_n + y'_{n-1})(y'_n + y_{n-1}) = 1$. Since we assume $y_n = f(x_n, x_{n-1})$ and $y_{n-1} = f(x_{n-1}, x_{n-2})$, we have $y_n = f(0,0) = f_0$ and $y_{n-1} = f(0, 1) = f_1$. Therefore, the possible assignments for y_{n-1}, y_n are also described by $(f_0 + f'_1)(f'_0 + f_1) = 1$. The same process is repeated for all rows of the relation table. The resulting constraints on the truth table coefficients are described in column 3 of

x_n	x_{n-1}	x_{n-2}	$y_n\ y_{n-1}$	$y_n\ y_{n-1}$
0	0	–	00, 11	$(f_0' + f_0)(f_0 + f_0') = 1$
0	1	–	01, 10	$(f_1' + f_3')(f_1 + f_3) = 1$
1	0	–	01, 10	$(f_2' + f_1')(f_2 + f_1) = 1$
1	1	–	00, 11	$(f_3' + f_3)(f_3 + f_3') = 1$

Table 2: Relation table for the inverter optimization problem. The second column shows the possible assignments to y_n, y_{n-1} corresponding to each input sequence; the third one expresses those assignments in terms of the coefficients f_j.

Table 2.

With this formalism, the possible solutions are the sets of coefficients that make true all the clauses in the table. 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.* The binate nature stems from the fact that coefficients can appear in the clauses with both phases. Exact and heuristic methods can be used for the optimal synthesis of the function f [7].

We summarize here the most important points of this approach. First, the 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 here as constraints implied by a synchronous recurrence equation. Third, the synthesis methods involves the solution of a binate covering problem; the implications among the values of the coefficients relate to the fact that the degrees of freedom are correlated.

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 functional synthesis problem is difficult, because it involves binate covering and because its size is exponential with the number of inputs of the subnetwork being replaced. Therefore optimization steps based on synthesis from recurrence relations should be used in conjunction with other circuit transformations, with complementary properties and that can also insure a fine granularity of the overall network.

3.5 Wave Pipelining.

The cycle time of combinational networks can be reduced by pipelining techniques. The introduction of intermediate registers shortens the combinational logic paths, allowing for the reduction of the cycle time. Since we consider here logic circuits where data are strobed in and out every clock cycle, then *throughput* is just the inverse of the cycle-time. Unfortunately, the added registers increase the overall area and power consumption. In addition, the overall latency of a pipelined circuit tends to be higher than the propagation delay through the corresponding combinational logic circuit, because of the propagation delays through the registers and the need to set the cycle time to be larger than the largest propagation delay in all the combinational pipe-stages.

We consider now a design style and CAD algorithms for designing high-performance pipelined circuits with *wave pipelining* [19]. Wave pipelining can be applied to combinational circuits to reduce the cycle-time, just as regular pipelining. However, when compared to regular pipelining, lower latency, power and area consumption can be achieved.

In essence, wave pipelining consists of a pipeline implementation without intermediate registers. Throughput (and therefore cycle-time) are comparable to a regular pipeline implementation. Instead, latency, area and power consumption are smaller, because of the lack of the intermediate registers. In addition, wave pipelining simplifies the clocking distribution.

The difficulty in designing a wave pipelined circuit stems from the following fact. The cycle-time is now smaller than the propagation delay through the combinational logic. Therefore, at any given time, more than one wave of data propagates between the boundary registers. Data are stored temporarily by the capacitances of the circuit. On the other hand, in a regular pipeline implementation, only one wave of data propagates between two registers at any given time. This is shown in Figure 6.

A necessary condition for correct operation of a wave pipelined circuit is that the waves do not mix. Therefore the path propagation delays from the inputs to each vertex of the Boolean network must be equal or approximatively equal. In a wave pipelined circuit, the cycle-time is bounded from below by the maximum unbalance in path delays (plus a component related to clock skew, rise/fall time and setup/hold times.) Con-

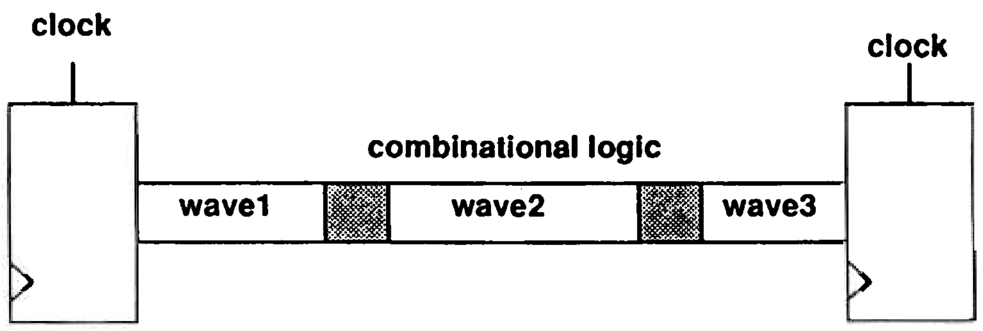

Figure 6: Regular pipelining versus wave pipelining.

versely, in a regular implementation the cycle-time is bounded from below by the maximum path delay (plus again clock skew and setup time.)

Exact path delay balancing is not only difficult to achieve in practice but also not very relevant. Indeed, in such case, the circuit cycle-time would still be determined by the clock skew and by the timing parameters of the boundary registers. It is important though to achieve a path balancing comparable to these quantities. This was shown to be feasible by experimenting with practical circuits.

To achieve a near balancing of path delays, two techniques can be used: inserting delay elements and fine tuning circuit parameters, such as currents. The former problem is discrete in nature and serves the purpose to reduce large unbalances. A polynomial-time algorithm to solve this problem was reported in [19]. The latter is a continuous optimization problem, that can be approached by linear or non-linear programming techniques.

In practice, it has been shown that circuits can be balanced so that two or three waves can fit, i.e. that a two/three-fold decrease in cycle-time can be achieved with respect to the combinational circuit. A few wave-pipelined implementation have been achieved. Historically, the first application was in the floating point unit of he IBM 360/91 computer. Recent implementations included a population counter realized in bipolar CML technology [18], whose cycle time was decreased by a factor 2.5 by using wave pipelining.

4 Summary

Logic-level optimization of synchronous digital circuits, and in particular cycle-time minimization, can be performed by using a structural network model and by applying circuit transformations. The transformations include those that can be applied just to the combinational component of the circuit, and that can be extended to moving and/or removing registers. 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 former operations are based on an extension of algebraic transformations to labeled expressions. The latter require the computation of *don't care* conditions. Since the extractions of *don't care* conditions in synchronous networks can be difficult, due to some correlation induced by the reconvergence of paths with different weights, optimization can be achieved by functional synthesis under the constraints imposed by a synchronous recurrence equation.

Wave pipelining allows us to optimize the cycle-time and the latency of a circuit, by providing a pipeline mode of operation without intermediate registers. The design of wave pipelined circuits requires careful balancing of the path delays and has it been used successfully to design high-performance chips.

5 Acknowledgments

This research was sponsored by NSF/ARPA, under grant No. MIP 8719546 and by DEC jointly with NSF, under a PYI Award program,

References

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

[2] K.Bartlett, W.Cohen, A.De Geus and G.Hachtel, "Synthesis and Optimization of Multilevel Logic under Timing Constraints" *IEEE Transactions on CAD/ICAS*, Vol. CAD-5 No. 4, pp.582-596, October 1986.

[3] K.Bartlett, G.Borriello and S.Raju, "Timing Optimization of Multiphase Sequential Circuits", *IEEE Transactions on CAD*, vol. 10, pp. 51-62, 1991.

[4] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD*, November 1987, pp. 1062-1081

[5] S. Dey, F. Brglez, and G. Kedem, " Partitioning Sequential Circuits for Logic Optimization", *Proceedings of 3^{rd} Int'l Workshop on Logic Synthesis* , Research Triangle Park, 1991.

[6] M.Damiani and G.De Micheli, "*Don't care* Specifications in Combinational and Synchronous Logic Circuits", *IEEE Transactions on CAD*, (to appear) and *CSL Technical Report*, CSL-TR, 1992.

[7] M.Damiani and G.De Micheli, "Synthesis and optimization of Synchronous Logic Circuits from Recurrence Equations", *Proceedings of EDAC* , 1992.

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

[9] G. De Micheli, " Synchronous Logic Synthesis: Algorithms for Cycle-Time Optimization", *IEEE Transactions on CAD*, vol. 10, pp. 63-73, 1991.

[10] G. De Micheli, "Performance-oriented synthesis in the Yorktown Silicon Compiler", *IEEE Trans on CAD/ICAS*, Vol. CAD-6, NO 5, pp. 751-765, September 1987.

[11] C.Leiserson, F.Rose and J.Saxe "Optimizing Synchronous Circuitry by Retiming", in R.Bryant, Editor *Third Caltech Conference on VLSI*, Computer Science Press, 1983, pp. 87-116.

[12] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, " Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques", *IEEE Transactions on CAD*, vol. 10, pp. 74-84, 1991.

[13] P.McGeer and R.Brayton, *Integrating Functional and Logic Domain in Logic Synthesis*, Kluwer, 1991.

[14] S.Muroga, Y.Kambayashi, H.Lai and J.Culliney, "The Transduction method - Design of Logic networks based on permissible functions", *IEEE Transactions on Computers* Vol. 38, No. 10, pp 1404-1424, October 1989.

[15] K.Sakallah, T.Mudge and K.Olukotun, "Analysis and Design of Latch-Controlled Synchronous Digital Circuits" *Proceedings of DAC*, 1990, pp.111-117

[16] G. Saucier , M. Crastes de Paulet and P. Sicard, "ASYL: A Rule-Based System for Controller Synthesis", *IEEE Transactions on CAD/ICAS*, Vol. CAD-6 No. 6, pp. 1088-1097 November 1987.

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

[18] D.Wong, G. De Micheli and M.Flynn, "A Bipolar Population Counter Using Wave Pipelining to Achieve 2.5X Normal Clock Frequency", *Proceedings of ISSC*, San Francisco, 1992.

[19] D.Wong, G. De Micheli and M.Flynn, "Inserting Active Delay Elements to Achieve Wave Pipelining", *Proceedings of ICCAD*, pp. 270-273, 1989.