An Exact Optimization of Two-Level Acyclic Sequential Circuits

Ellen M. Sentovich and Robert K. Brayton
Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

Memorandum No. UCB/ERL M94/48

1 July 1994

Abstract

Several algorithms for gate-level sequential circuit optimization have been reported in the literature. They perform operations similar to those in the more mature multilevel combinational domain while taking relationships across several time periods into account. These techniques are heuristic and their application ad hoc: there is no guarantee of optimality by any definition beyond "no further improvement". In this paper, we present a technique for producing an *optimum* two-level acyclic sequential circuit. While the circuit restrictions (*e.g.*, two-level, acyclic) and cost function are limiting, the guarantee of optimality is novel and illuminating. The technique presented herein is useful for optimizing sub-circuits of a multilevel sequential circuit just as two-level combinational techniques have been in the combinational domain. Furthermore, the algorithm can be used to detect precisely circuits in which logic sharing across latch boundaries is actually possible – a hitherto unsolved problem.

1 Introduction

Several methods for optimizing gate-level sequential circuits have been reported. [MSBSV91] combines peripheral retiming, which pushes registers to the boundary of a logic block, with standard combinational optimization techniques; [DeM91] considers extraction, factorization, and elimination of synchronous factors across register boundaries; [Lin93] presents a more formal framework for the extraction of synchronous factors. These algorithms are heuristic and hence do not guarantee an optimum result in terms of number of gates, registers, or connections. They are simply iterated until no further improvement is achieved. In fact, there are no theoretical optimum results reported thus far for gate-level sequential circuits.

While only limited results have been reported for exact optimization of multilevel combinational circuits (*e.g.*, [Sas89]), solutions to the exact two-level combinational logic optimization problem are well-developed. The input is a combinational circuit represented by a Boolean function; the output is a minimum-gate, minimal-connection implementation represented by a prime Boolean function with a minimum number of implicants. Traditional exact two-level logic optimization algorithms consider only prime covers and therefore are based on the assumption that an implicant costs no more than any implicant it contains. This is a valid assumption when the target is a minimum-gate solution with implicant cost equal to the number of literals.[1] As a result, a minimum-gate solution can be found by selecting a minimum number of prime implicants. Exact techniques based on this cost function [McC65, Rud89] employ the following algorithm:

Two-level Exact Algorithm:

1. Compute all primes.
2. Form the covering table.
3. Solve the minimum-column cover problem.

[1] See [Rud89] for a brief on cost functions for PLAs.

The covering table contains one row for each minterm of the *ON-set* and one column for each prime. A '1' entry in the table indicates that the column prime covers the row minterm. The minimum-column cover represents a minimum-gate implementation.[2]

This algorithm is useful for producing optimum PLA implementations [BHMSV84], for solving subproblems in multilevel logic optimization [BRSVW87], and for producing good initial implementations for multilevel optimization.

Exact techniques for two-level acyclic sequential circuits can similarly provide a means of local optimization for multilevel sequential circuits. A set of nodes and registers in a multilevel circuit can be clustered together and an optimum implementation found using the algorithm presented herein. The optimum solution considers sharing logic across multiple time frames. The ability to do such local restructuring of logic and registers is important for various cost functions, such as minimum area and minimum delay, but also for other criteria such as re-encoding parts of a state machine. Furthermore, exploration of such techniques provides insight into the logic optimization problem in the synchronous domain.

The main results are summarized in the next section, with detailed development and analysis following. This begins with the underlying assumptions that are made for the development of exact two-level acyclic sequential circuit optimization in section 3. This is followed by the definition of a synchronous Boolean function, which is used to represent a sequential circuit, and operations on synchronous Boolean functions in section 4. The exact optimization algorithm is presented in section 5. A description of the implementation, which involves casting the problem to a multioutput combinational problem, is given in section 6. Conclusions and directions for future work are given in section 7. An expanded version of this work is available in [SB94].

2 Summary of Main Results

The key result of this investigation is the development of an exact optimization procedure for sequential circuits. The input is an acyclic sequential circuit consisting of logic gates and edge-triggered registers controlled by a single global clock. The output is a minimum-gate two-level implementation with an arbitrary number of registers. The procedure effectively combines retiming and logic optimization while guaranteeing the optimality of the result. This was not guaranteed by previously published algorithms which applied the two in an ad hoc fashion.

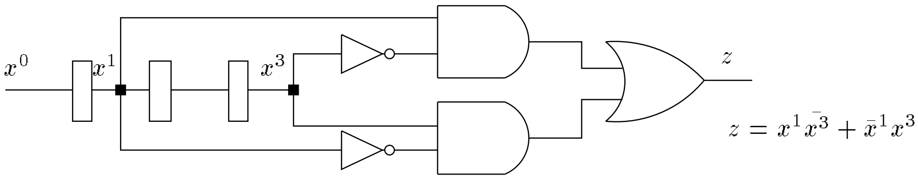
The optimization problem for a single synchronous output is proven to be equivalent to the multiple output two-level combinational problem where each of the outputs represents a particular time frame of the single output. The target implementation (two-level, minimum-gate, arbitrary number of registers) is restrictive yet has application to sub-problems (*e.g.*, sequential node optimization) and certain architectures (*e.g.*, some PLDs, where registers are essentially free). This algorithm is also a useful research tool for determining with certainty when logic can actually be shared across register boundaries. This is demonstrated in section 6.3. This logic sharing determination is an important step since in many cases in [MSBSV91, DeM91, Lin93] no area improvement was obtained. It was not known if this was due to a lack of common logic across time frames in the circuit structure or due to limitations in the (heuristic) algorithms.

3 Preliminaries: Assumptions and Problem Definition

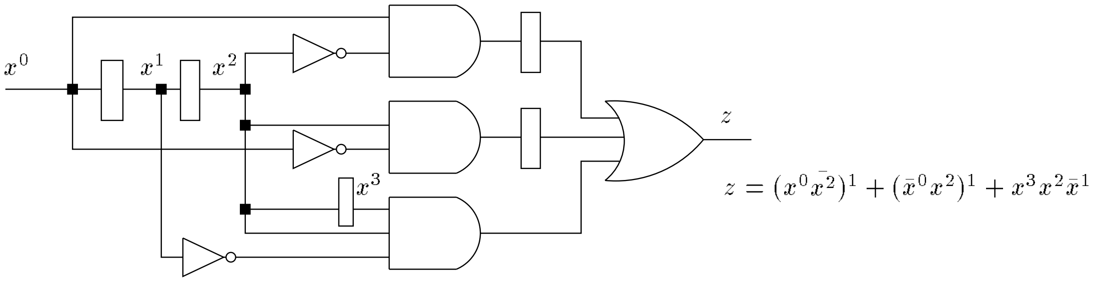
The algorithm developed in the sequel applies to acyclic, synchronous circuits with edge-triggered registers controlled by a single global clock. Such circuits are hereafter referred to as synchronous circuits.[3] The acyclic property ensures that the output signals can be expressed as feedback-free functions of the input signals. The synchronous, edge-triggered, and single-clock properties together ensure that the retiming algorithm can be applied to the circuit while maintaining the correct input/output behavior [LS83]. The initial synchronous circuit can be two-level or multilevel. Each output is later expressed as a single function of the synchronous inputs (*i.e.*, the input signals at different time periods) and represented and optimized using a single *synchronous Boolean function* (a precise definition follows in section 4.1). Structurally, the re-expression process is a combination of combinational logic collapse (which may involve duplication) and retiming. The resulting synchronous Boolean function is equivalent to a two-level circuit with all the registers stacked at the inputs.

[2] As noted in [Rud89], this does not guarantee a minimum-transistor (minimum-literal) implementation as this cost function does not satisfy the assumption above. See [Rot80] for a minimum-transistor optimization algorithm.

[3] In fact, this is a narrower definition than the one generally accepted which requires the sequential elements to be clocked but permits any clocking scheme. We re-use the term here to avoid introducing additional terminology.



(a) A collapsed two-level synchronous circuit



(b) An equivalent two-level synchronous circuit

Figure 1: Two forms of a Two-level Synchronous Circuit

The goal is to obtain an optimum two-level representation for the synchronous circuit. The two-level representation targeted is given in the following definition.

Definition 3.1 *A* **two-level synchronous circuit** *has a level of* AND *gates followed by a level of* OR *gates, and an arbitrary number of registers preceding and following each level of logic.*

It is assumed that all the synchronous inputs are available in both their true and inverted forms. A two-level circuit can be implemented in other ways besides the AND/OR configuration. For simplicity, the structure will always be viewed here as an AND/OR combination.

Example 3.1 *Two two-level versions of the same circuit are shown in Figure 1.*

Exact Optimization Problem: *Given an acyclic synchronous circuit, find a two-level synchronous circuit implementation with a minimum number of logic gates.*

The target implementation is a two-level synchronous circuit as given in Definition 3.1. The solution has a minimum number of AND gates; the number of OR gates is fixed by the number of outputs. Initially, no constraints are placed on the number or location of the registers.

4 Functional Representation: Synchronous Boolean Functions

A synchronous Boolean function is similar to a combinational Boolean function (defined precisely in [BHMSV84]), with an extension to the input space for expression of the sequential nature of the function.

4.1 Definition

Definition 4.1 *Let* $B = \{0, 1\}$, $R = \{0, 1, \ldots, r\}$ *and* $D = \{0, 1, 2\}$. *An n-input, m-output incompletely specified* **synchronous Boolean function** $\mathcal{F}$ *is a mapping*

$$\mathcal{F} : (B \times R)^n \rightarrow D^m$$

The values $i \in R$ represent time points, and r is the *depth* of the function.

A synchronous circuit with n input variables $x_1, x_2, \ldots, x_n$ is represented by a synchronous function of $n(r+1)$ *synchronous input variables* $\mathbf{x} = \{x_1^0, x_2^0, \ldots, x_n^0,\ x_1^1, x_2^1, \ldots, x_n^1,\ \ldots,\ x_1^r, x_2^r, \ldots, x_n^r\}$. A *synchronous literal* x_i^j ($\bar{x}_i^j$) represents the value 1 (0) for the input x_i at time j. The superscript 0 is often omitted: $x_i^0 = x_i$. A *synchronous minterm* of n inputs and depth r is a concatenation of $n(r+1)$ literals and represents an assignment of each input to a value in $\{0,1\}$ for each time point $\{0, 1, \ldots, r\}$. A *synchronous cube* c is a concatenation of synchronous literals. A literal is contained in a cube, $x_i^j \in c$ ($\bar{x}_i^j \in c$), if the value of input x_i at time j is 1 (0) in c. The following are equivalent: $x_i^j \in c$ and $x_i^j = 1$ in c; $\bar{x}_i^j \in c$ and $x_i^j = 0$ in c; x_i^j, $\bar{x}_i^j \notin c$ and $x_i^j = 2$ in c.

It is sometimes convenient to separate the input part of a synchronous cube from the state part. Let $S = \{1, 2, \ldots, r\}$, and x^i be a minterm (recall x_i is a variable, x_i^j is a synchronous variable, x_i^j and $\bar{x}_i^j$ are synchronous literals). $x^i \in B^n$ is an *input minterm*, $s^i \in (B \times S)^n$ is a *state minterm*, and $v^i \in (B \times R)^n$ is a *synchronous input minterm* (a concatenation of x^i and s^i). x^i represents the current values of the inputs $x_1, x_2, \ldots, x_n$, while v^i represents the values of the inputs currently and for r previous time periods. Note that the "state" part of a synchronous input minterm v^i is composed specifically of previous input values; it does not represent symbolic states that may be encoded differently in different implementations of the same circuit. (In other words, two equivalent circuits in their collapsed form will have the same encoding since all registers are stacked at the inputs.)

4.2 Relationship to Synchronous Circuits

Any **acyclic synchronous circuit** *can be represented by a set of synchronous Boolean functions, one for each output.*

An acyclic synchronous circuit can be collapsed so that each output is expressed as a synchronous Boolean function of the synchronous inputs. The collapse operation is possible because the circuit is acyclic; collapse with retiming does not change the logical behavior of the circuit because it is synchronous. The collection of synchronous Boolean functions for the outputs is therefore a valid representation for the synchronous circuit.

Each synchronous variable x_i^j in the function represents a signal in the circuit, and each synchronous literal, x_i^j or $\bar{x}_i^j$ represents the value of the signal x_i^j. The values $j \in R$ represent the values of a global clock, and the depth r of a circuit is the maximum number of registers from a single input to a single output.

4.3 Equivalence of Synchronous Functions

Definition 4.2 *Two completely specified synchronous Boolean functions, $F_1 : (B \times R_1)^n \rightarrow B^m$, $R_1 = \{0, 1, \ldots, r_1\}$, and $F_2 : (B \times R_2)^n \rightarrow B^m$, $R_2 = \{0, 1, \ldots, r_2\}$, are* **equivalent** *(denoted $F_1 = F_2$) if, for each minterm of the smaller synchronous input space v^i, they produce the same output minterm: if $r_1 \leq r_2$, then $F_1 = F_2 \Leftrightarrow \forall v^i \in (B \times R_1)^n,\ F_1(v^i) = F_2(v^i)$.*

The intuition behind this definition of equivalence and the following corollary is that two synchronous circuits are equivalent if and only if they produce the same output minterm given the same sequence of $r_1 + 1$ input minterms. That is, they will behave identically *after* the first r_1 input minterms have been applied. This is independent of initial state. Note if $r_1 < r_2$, $F_1 = F_2$ implies F_2 does not functionally depend on $\mathbf{x}^{r_1+1}, \mathbf{x}^{r_1+2}, \cdots \mathbf{x}^{r_2}$.

Corollary 4.1 *Given two synchronous circuits $\mathcal{C}_1$ and $\mathcal{C}_2$ with depths r_1 and r_2, where $r_1 \leq r_2$ ($r_2 < r_1$), and represented by synchronous Boolean functions F_1 and F_2, if $F_1 = F_2$ then $\mathcal{C}_1$ and $\mathcal{C}_2$ are guaranteed to produce the same output sequence given the same input sequence after the first $r_1 + 1$ ($r_2 + 1$) input minterms have been applied.*

While it appears that definition 4.2 requires F_1 and F_2 to produce the same output given the same input values *and* state values, in fact, equivalence is based on input values only since the state values in synchronous Boolean functions (representing collapsed synchronous circuits) are precisely the input values at previous time periods. As a result of corollary 4.1, two two-level circuits with equivalent synchronous Boolean functions are guaranteed to have equivalent steady-state behavior. The algorithm in section 5 produces an optimum equivalent synchronous function.

4.4 Synchronous Cubes and Synchronous Implicants

A number of definitions and some explanation is needed as a foundation for proving the exact result given in section 5. The details of these definitions are in Appendix A. The key results are the definitions of a synchronous implicant, synchronous implicant containment, and a synchronous implementation set (unlike the combinational case, there can be several implementations of a synchronous implicant).

A *synchronous implicant* $c(I_c)$, where c is a synchronous cube and I_c is a set of integers, represents several synchronous cubes that are implicants of the given function. Example: $x_1 x_2^1(0,2) \Rightarrow x_1 x_2^1$ and $x_1^2 x_2^3$ are implicants. A synchronous implicant $c(I_c)$ *contains* $d(I_d)$, $c(I_c) \overset{s}{\supseteq} d(I_d)$, if each synchronous cube in $d(I_d)$ is contained in the set of synchronous cubes represented by $c(I_c)$. The *cost* of a synchronous cube c is number of literals in c. A *synchronous implementation set* represents all implementations (retimings) of a synchronous implicant.

5 Optimizing Synchronous Functions

In Appendix A section A.1, an example is given which illustrates the need for exact two-level techniques, *i.e.,* it illustrates that combining retiming and resynthesis techniques is not enough to guarantee optimality.

Exact two-level combinational logic optimization techniques generate a solution containing a minimum number of prime implicants using the algorithm given in section 1. This guarantees a minimum-gate solution. Exact two-level synchronous logic optimization techniques will likewise generate a solution with a minimum number of gates, and with an unconstrained number and placement of registers around the gates. The solution will take the form of a minimum number of prime implicants, with the notion of a prime implicant extended for synchronous circuits.

5.1 Exact Two-level Synchronous Logic Optimization

Definition 5.1 *A* **synchronous prime** *is a synchronous implicant that is synchronously contained by no other implicant than itself:* $c(I_c) \overset{s}{\subseteq} d(I_d)$ *and* c *is prime* $\Rightarrow c(I_c) = d(I_d)$.

Proposition 5.1 *All implementations of a synchronous cube have the same cost with respect to a minimum-gate implementation.*

Proof. See Appendix B. □

Proposition 5.2 *A synchronous implicant costs no more than any synchronous implicant it contains with respect to a minimum-gate implementation.*

Proof. See Appendix B. □

Corollary 5.1 *There exists a minimum-gate two-level synchronous circuit that is represented by a set of synchronous primes.*

The key result of Corollary 5.1 is that an exact solution can be found by considering only synchronous primes. This is analogous to the combinational case, so the two-level exact algorithm given in section 1 can be used for synchronous circuits.

5.2 Prime Generation

The consensus operation can be expanded to generate synchronous primes. Intuitively, the combinational consensus of two implicants, if not empty, generates

- an implicant that contains both implicants (a larger implicant), or
- an implicant that contains parts of both implicants (an implicant that "bridges the gap" between two implicants), or
- an implicant that is contained by both implicants and is an implicant for a larger set of output functions.

Synchronous consensus will generate in addition

- an implicant that contains parts of both implicants in several time frames.

All implementations of a synchronous implicant must be considered for prime generation. Detailed definitions of an implementation set associated with a synchronous implicant, and the consensus of two such sets, denoted $\overset{ss}{\odot}$, is given in Appendix A. The key theorem is the following:

Theorem 5.1 *Iterated synchronous implementation set consensus generates all synchronous primes.*

Proof. See Appendix A. □

5.3 Prime Covering and Implementation

The covering table is built with a row per synchronous minterm of the *ON-set* and a column per synchronous prime. The *ON-set* minterms are combinational cubes in the synchronous space $(B \times R)^n$ and are generated from $f - DC(f)$. A prime $c(I_c)$ covers a minterm m if $\exists i_c \in I_c$ *s.t.* $m \subseteq^{i_c} [c]$. (Note that the *time shift by* k of c, denoted $^k[c]$, is obtained by adding k to all superscripts in c.) The synchronous primes are initially expanded into a set of combinational primes in the synchronous space:

$$\text{combinational primes of prime } c(I_c) = \left\{ d \mid d = ^{i_c}[c]\,,\; i_c \in I_c \right\}$$

Each column therefore represents a set of cubes, and minterm m is covered by $c(I_c)$ if $m \subseteq$ combinational primes of $c(I_c)$. The complete table is formed and the covering problem solved. See [Rud89] for efficient minimum column-covering algorithms.

Once the minimum set of synchronous primes has been selected, an implementation is generated directly by implementing each prime $c(I_c)$ by its implicit structure $(c)^{i_{c_{max}}}$ (see Appendix A).

Example 5.1

$$\begin{aligned} f &= x_1x_3 + x_2^1x_3^1 \\ DC(f) &= x_1\bar{x}_2x_3 + \bar{x}_1^1x_2^1x_3^1 \end{aligned}$$

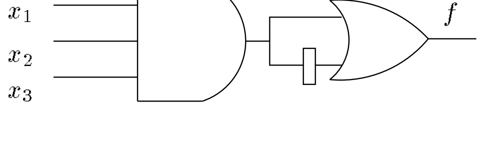
Simple Synchronous Implicants	*Implementations*	
$x_1x_3(0)$	$x_1x_3(0)$	
$x_2x_3(1)$	$x_2x_3(1)$	$x_2^1x_3^1(0)$
$x_1\bar{x}_2x_3(0)$	$x_1\bar{x}_2x_3(0)$	
$\bar{x}_1x_2x_3(1)$	$\bar{x}_1x_2x_3(1)$	$\bar{x}_1^1x_2^1x_3^1(0)$

Remove contained cubes: $x_1\bar{x}_2x_3(0) \overset{s}{\subseteq} x_1x_3(0)$, $\bar{x}_1x_2x_3(1) \overset{s}{\subseteq} x_2x_3(1)$. *Synchronous consensus:* $x_1x_3(0) \overset{ss}{\odot} x_2x_3(1) = x_1x_2x_3(0,1)$. *No cubes can be removed by containment, and subsequent iterations of consensus produce no new cubes.*
Primes: $x_1x_3(0)$, $x_2x_3(1)$, $x_1x_2x_3(0,1)$. $ON(f) = x_1x_2x_3(0) + x_1x_2x_3(1)$.

Covering table:

	x_1x_3	$x_2^1x_3^1$	$x_1x_2x_3 + x_1^1x_2^1x_3^1$
$x_1x_2x_3$	1	0	1
$x_1^1x_2^1x_3^1$	0	1	1

(The rows above actually contain cubes rather than minterms. This creates a more compact but not complete covering table. The result is the same for the complete table.)
Solution: $f = x_1x_2x_3(0,1)$.



6 Implementation using Combinational Prime Generation

The theory presented in the last few sections treats synchronous functions in a slightly different way than combinational functions. As shown in the following section, a synchronous function can be partitioned into a combinational multioutput function, where each output represents a different time frame. This formulation provides the basis for a simple implementation using existing, efficient ESPRESSO code.

6.1 Synchronous Functions as Multioutput Functions

Solving the two-level exact synchronous circuit optimization problem as a multioutput two-level combinational optimization problem involves three steps:

1. Partition the synchronous Boolean function into a set of combinational Boolean functions that together represent the synchronous function.
2. Compute the primes for the multioutput combinational function and translate them to synchronous primes.
3. Form the covering table and find an optimum solution.

In the following, f is a synchronous Boolean function, $\mathbf{x}^i = \{x_1^i, x_2^i, \ldots, x_n^i\}$ is the set of input variables in the i^{th} time frame, and $\theta_i f$ is the time-shift of all literals in all cubes of f by i (*e.g.*, $\theta_1(x_1^1 x_2^1 + x_1) = x_1^2 x_2^2 + x_1^1$). $\mathcal{C}_{\mathbf{X}^i} f$ is the consensus of f with respect to all the variables in $\mathbf{x}^i$.

A synchronous Boolean function f with depth r can be partitioned into $r+1$ functions, $f_0, f_1, \ldots, f_r$, where

$$\begin{aligned} f_0 &= f \\ f_1 &= \theta_{-1}(\mathcal{C}_{\mathbf{X}^0} f) \\ f_2 &= \theta_{-2}(\mathcal{C}_{\mathbf{X}^0\mathbf{X}^1} f) \\ &\vdots \\ f_r &= \theta_{-r}(\mathcal{C}_{\mathbf{X}^0\mathbf{X}^1\ldots\mathbf{X}^{r-1}} f) \end{aligned}$$

Proposition 6.1 $f = f_0 +^1 [f_1] +^2 [f_2] + \cdots +^r [f_r]$.

Proof. $f_0 = f$ by definition. ${}^i[f_i] =^i [\theta_{-i}\mathcal{C}_{\mathbf{X}^0\mathbf{X}^1\ldots\mathbf{X}^{i-1}} f] = \mathcal{C}_{\mathbf{X}^0\mathbf{X}^1\ldots\mathbf{X}^{i-1}} f \subseteq f$. □

Clearly, the functions f_i are computed efficiently by noting that $f_{i+1} = \theta_{-1}(\mathcal{C}_{\mathbf{X}^0} f_i)$ for $i = 0, 1, \ldots, r-1$.

$$\begin{aligned} f_0 &= f \\ f_1 &= \theta_{-1}(\mathcal{C}_{\mathbf{X}^0} f_0) \\ f_2 &= \theta_{-1}(\mathcal{C}_{\mathbf{X}^0} f_1) \\ &\vdots \\ f_r &= \theta_{-1}(\mathcal{C}_{\mathbf{X}^0} f_{r-1}) \end{aligned}$$

Example 6.1 *Let* $f = x_1 x_3^2 + \bar{x}_1 x_3^2 + x_2^1$. *Then* $f_0 = x_1 x_3^2 + \bar{x}_1 x_3^2 + x_2^1$, $f_1 = x_2 + x_3^1$, $f_2 = x_3$. *The partitioned function is* $f = x_1 x_3^2 + \bar{x}_1 x_3^2 + x_2^1 + (x_2 + x_3^1)^1 + (x_3)^2$.

Once the synchronous Boolean function has been partitioned into its corresponding multioutput function, all the synchronous primes can be generated by generating the combinational primes of the multioutput function. A combinational prime p that is an implicant for outputs $f_i, f_{i+1}, \ldots, f_j$, for example, is equivalent to a synchronous prime $p(i, i+1, \ldots, j)$.

Proposition 6.2 *Given a synchronous function f, iterated combinational consensus of the corresponding partitioned multioutput function F produces all synchronous primes.*

Proof. See Appendix B. □

This method of generating synchronous primes also generates some non-primes and duplicates of some primes in different time frames. A final time-shift operation is performed on each cube to generate simple cubes and synchronous containment can be used to remove any non-primes. In practice, a judicious ordering of the combinational prime generation and containment operations, interspersed with simple synchronous operations (*e.g.*, creating simple implicants, removing implicants with support outside of $f + DC(f)$) obviates the need for performing the more expensive synchronous containment check.

Despite the overhead of partitioning the function, converting it to and from a synchronous representation, and generating extra cubes when generating primes, this method for prime generation has a distinct advantage since efficient combinational prime generation techniques already exist [Rud89].

```
synch_exact(C)
{
    /* Break cycles (if necessary) with a minimum feedback arc set */
    make_acyclic(C);

    /* Create the partitioned function and generate the primes */
    F = ON(C); D = DC(C);
    FDC = F + D;
    F_part = partition(FDC);
    D_part = partition(D);
    generate_combinational_primes(F_part);

    /* Remove duplicates, remove non-primes */
    trim_primes(F_part);

    /* Split each synchronous prime into a set of combinational cubes */
    /* and form the covering table */
    split_primes(F_part);
    T = generate_covering_table(F_part, D_part);

    /* Merge back to synchronous primes and find a minimum cover */
    merge_columns(T);
    find_minimum_cover(T);
    C = generate_implementation(T, F_part);
}
```

Figure 2: Algorithm

6.2 Don't Care Conditions

External don't care conditions may be present if, for example, the acyclic synchronous circuit is part of a larger sequential design. In this case, there may be several types of flexibility that can be exploited in determining an optimal implementation, *e.g.*, don't care conditions, Boolean relations, and flexibility based on the cyclic sequential nature of the machine. Some of these can be expressed as don't care conditions at the logic level and expressed as a synchronous Boolean function.

This don't care function can easily be incorporated into the optimization algorithm presented herein: given f and $DC(f)$, the primes are generated from the function $f + DC(f)$. If the partitioned-function method is being used, $f + DC(f)$ is partitioned before prime generation (*i.e.*, f and $DC(f)$ are not partitioned separately).

6.3 Experimentation

The final algorithm as implemented is outlined in figure 2. Some examples have been run and the initial results are illuminating. We have compared the results of the exact sequential algorithm with the exact combinational one. The reason for this is that one open question in the area of gate-level sequential circuit optimization has been the utility of algorithms that exploit sharing of logic across several time frames. Results reported thus far [MSBSV91, DeM91] have suggested that logic functions of this form are uncommon in real circuits. After testing our algorithm on 10 of the smaller MCNC benchmark finite state machines, we have found the presence of this logical structure in 3 of them, as evidenced by a better combined result for the synchronous algorithm (total of 73 cubes) than the combinational one (total of 79 cubes). The logic sharing exists in these circuits but is not common. We expect this logical feature to appear more in data path-type circuits. This suggests that gate-level sequential techniques do in fact have their utility. Our experiments are ongoing, including applications to local re-encoding of finite state machines (applying the algorithm after state assignment, with various choices for breaking the cycles) and to various types of acyclic sequential circuits.

7 Conclusions and Ongoing Work

We have presented an algorithm for obtaining an exact minimum-gate implementation of an acyclic circuit. Theoretically, this provides insight into sharing logic across several time frames. Initial experiments are encouraging are we are continuing to apply the algorithm at a variety of levels in the sequential synthesis process.

Minimizing the number of registers is an important related problem. Ignoring register cost, as done in the initial algorithm, is unrealistic for some circuit design styles since the cost of a register is usually several times the cost of a simple gate. (However, for some design styles, notably PLDs, the registers are built-in and hence incur no additional cost.) We are currently exploring techniques for minimizing the number of registers. There is a partial analogy between minimizing literals in a combinational circuit and minimizing registers in a sequential circuit. This means that a technique similar to the MAKE_SPARSE procedure in ESPRESSO can be used to heuristically reduce the number of registers. In addition, we are investigating expanding the covering procedure to take register cost into account. We are also exploring extensions to cyclic circuits by unrolling cycles and applying these techniques to the unrolled circuit. This may lead to a re-encoding technique that produces smaller FSM implementations. Finally, we are investigating extensions of this technique to produce limited forms of multilevel circuits with some guarantee of optimal sharing across time frames.

8 Acknowledgements

Many useful discussions with Richard Rudell about two-level exact combinational techniques are gratefully acknowledged. This research was supported by NSF under grant number EMC-8419744.

References

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BRSVW87] R.K. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli, and A.R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[DeM91] G. DeMicheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE Transactions on Computer-Aided Design*, CAD-10(1):63–73, January 1991.

[Lin93] B. Lin. Restructuring of Synchronous Logic Circuits. In *Proceedings of the European Conference on Design Automation*, pages 205–209, Paris, France, February 1993.

[LS83] C.E. Leiserson and J.B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[McC65] E.J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill Book Company, 1965.

[MSBSV91] S. Malik, E.M. Sentovich, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques. *IEEE Transactions on Computer-Aided Design*, CAD-10(1):74–84, January 1991.

[Rot80] J.P. Roth. *Computer Logic, Testing and Verification.* Digital System Design. Computer Science Press, 1980.

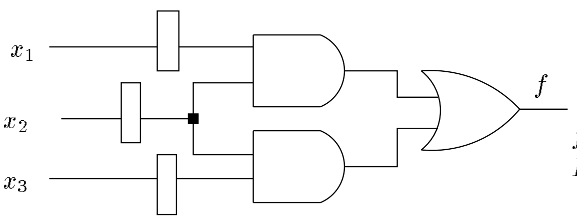
[Rud89] Richard L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, April 1989. Memorandum No. UCB/ERL M89/49.

[Sas89] T. Sasao. On the Complexity of Three-Level Logic Circuits. In *Proceedings of the International Workshop on Logic Synthesis*, North Carolina, May 1989.

[SB94] E.M. Sentovich and R.K. Brayton. An Exact Optimization of Two-Level Acyclic Sequential Circuits. Technical Report Memorandum No. UCB/ERL, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, July 1994.

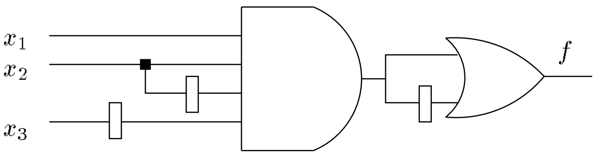
A Operations on Synchronous Cubes

All combinational operations (*e.g.*, intersection, union, containment, consensus) can be applied to synchronous cubes. The synchronous variables are treated independently (*i.e.*, the synchronous relationship between variables is not considered). Precise definitions for these operations are given in [BHMSV84]. Synchronous operations take into account the time-delay relationship between synchronous variables. The definitions in this section are given for single-output cubes for ease of exposition; the extension to multiple-output cubes is straight forward.



$$
\begin{aligned}
f &= x_1^1 x_2^1 + x_2^1 x_3^1 \\
DC(f) &= x_1^1 x_2^1(\bar{x}_2^2 + \bar{x}_3^2) + \\
&\quad x_2^1 x_3^1(\bar{x}_1 + \bar{x}_2)
\end{aligned}
$$

(a) Original circuit



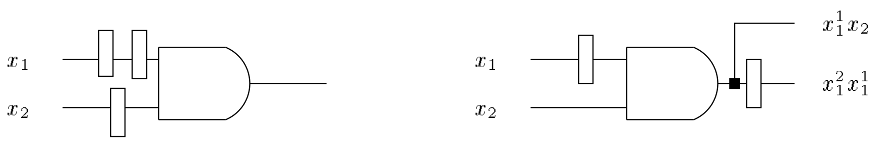
(b) Minimum-gate implementation

Figure 3: Optimizing a Two-level Synchronous Circuit

A.1 Synchronous Cube Implementation

There is an important difference between combinational and synchronous cubes with respect to the minimum AND-gate optimization goal. While a combinational cube has exactly one implementation, a synchronous cube has several implementations depending on the placement of the registers.

Example A.1 *The synchronous cube $x_1^2 x_2^1$ has two implementations:*



(a) $x_1^2 x_2^1$ (b) $(x_1^1 x_2)^1$

All implementations have the same cost (a single AND gate with the same number of inputs), but the maximally forward-retimed structure (*e.g.*, (b) in example A.1) implements a maximum number of synchronous cubes.

One might consider retiming all gates *forward* as much as possible before translating the circuit into Boolean functions and beginning the process of generating primes and finding an optimum cover. However, such an algorithm would require multiple forward retimings interspersed with prime generation techniques. In general, it is tempting to combine existing techniques, namely structural retiming and algebraic combinational logic optimization, to optimize a sequential circuit. In practice, this approach is not robust as it is difficult to determine the correct sequence of operations. A more direct optimization algorithm, which will be described in section 5, dispenses with retiming (a structural operation) and instead considers all implementations of a synchronous cube algebraically to find an optimum solution.

Example A.2 *In the circuit shown in Figure 3, $ON(f) = x_1 x_2 x_2^1 x_3^1 + x_1^1 x_2^1 x_2^2 x_3^2$ and the minimum-gate implementation contains a single* AND *gate. Obtaining this through a combination of retiming and combinational logic optimization would entail a complex combination of partial forward retiming, retiming the don't care set, and combinational optimization.*

While the best implementation for an individual cube is the maximally forward-retimed structure, all implementations must be considered when implementing several cubes of a function. In example A.2, the final cube $x_1 x_2 x_2^1 x_3^1$ is obtained by using the *DC-set* and considering the implicant $x_1^1 x_2^1$ in its forward-retimed structure $(x_1 x_2)^1$ while considering $x_2^1 x_3^1$ in its original structure. Synchronous "primes" are generated by considering all possible implementations.

In the following definitions, literal superscripts imply registers at the AND-gate inputs, and cube superscripts imply registers at the AND-gate output. This structural notation is illustrated in example A.1. A *synchronous cube* c is assumed to have only input registers and is the generic representation for cubes in the synchronous Boolean space. A *structural synchronous cube* $(c)^r$ has input registers and r registers at the output.

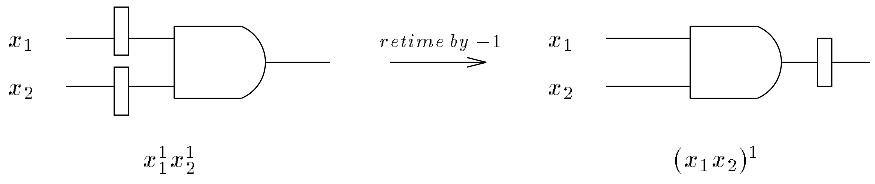


Figure 4: Retiming

A.2 Basic Definitions

Cube Cost: The *cost* of a synchronous cube $(c)^r$ is $|c|$, which is the number of literals in c. The number of registers needed to implement the cube is not included in the cost.

Cube Time-Shift: The *time shift by k* of a synchronous cube c is the synchronous cube d, $d = ^k[c]$, which is obtained by adding k to the superscripts of all the literals in c. It is used here to facilitate the remaining definitions. It is assumed that a time-shift operation always results in a cube with positive superscripts.

Let $j_{min}(c)$ be the minimum superscript in c.

Synchronous Simple Cube: A *simple cube* c has a minimum superscript of 0: $j_{min}(c) = 0$.

Retiming: The *retiming by k* of a synchronous cube $(c)^r$, denoted $[(c)^r]^{retime\ by\ k}$, is the synchronous cube d, where

$$d = (^k[c])^{r-k}$$

Retiming by k is equivalent to the movement of k registers from the output of c to its inputs. An example of retiming is shown in figure 4.

Note that a structural synchronous cube $(c)^r$ implements several synchronous cubes (as in example A.1):

$$(c)^r \text{ implements } c, ^1[c], ^2[c], \ldots, ^r[c]$$

A synchronous cube c that is maximally forward retimed implements

$$^{-j_{min}(c)}[c], ^{-(j_{min}(c)-1)}[c], \ldots, ^0[c] = c$$

A.3 Definitions for Synchronous Prime Generation

A structural synchronous cube $(c)^r$ implements several synchronous cubes, only some of which may be implicants of the given function. The notation $(c)^r$ is sufficient for indicating the implementation structure, but not for indicating which cubes are implicants. A synchronous cube is annotated to create a synchronous implicant.

Synchronous Implicant: A *synchronous implicant* is an annotated cube $c(I_c)$, where

$$\begin{array}{lcl} c & & \text{is a cube} \\ I_c & = & \left\{ i_c \mid ^{i_c}[c] \text{ is an implicant} \right\} \end{array}$$

The maximum integer in I_c is denoted $i_{c_{max}}$:

$$i_{c_{max}} = \max_{I_c} i_c$$

Example: $x_1 x_2^1(\{0, 2\}) \Rightarrow x_1 x_2^1$ and $x_1^2 x_2^3$ are implicants.

Each synchronous implicant represents several implicants of the function and is sometimes called a *synchronous implicant set*. A synchronous implicant is not a canonical form: $x_1^1 x_2^1$ can be represented by either $x_1^1 x_2^1(0)$ or $x_1 x_2(1)$. One implementation of $c(I_c)$, the *implicit structure* for the implicant, is the structural synchronous cube $(c)^{\max_{i_c \in I_c} i_c}$. A synchronous implicant $c(I_c)$ that represents only one implicant, $|I_c| = 1$ with $i_c \in I_c$, will be denoted $c(i_c)$. The implicants $\phi(I_c)$ and $c(\phi)$ represent the null cube.

Simple Synchronous Implicant: A *simple synchronous implicant* is an annotated simple cube $c(I_c)$, where

$$\begin{array}{lcl} c & & \text{is a simple cube} \\ I_c & = & \left\{ i_c \mid ^{i_c}[c] \text{ is an implicant} \right\} \end{array}$$

It is a canonical form for an implicant. The implicit structure for a simple synchronous implicant is a maximally forward-retimed cube.

The remaining definitions operate on simple synchronous implicants since they are canonical. However, they implicitly represent only one implementation of an implicant. For prime generation, which is described in section 5.2, each implicant is expanded to its set of implementations.

Synchronous Intersection: The *synchronous intersection* of synchronous implicants $c(i_c)$ and $d(i_d)$ is $e(i_e) = c(i_c) \overset{s}{\cap} d(i_d)$, where

$$\begin{array}{ll} e = c \cap {}^{i_d - i_c}[d], i_e = i_c & \text{if } i_c < i_d \\ e = {}^{i_c - i_d}[c] \cap d, i_e = i_d & \text{if } i_c > i_d \\ e = c \cap d, i_e = i_c & \text{if } i_c = i_d \end{array}$$

Synchronous intersection of simple implicants produces a simple implicant. Example: $x_1(1) \overset{s}{\cap} x_2(2) = x_1 x_2^1(1)$.

Synchronous Set Intersection: The *synchronous set intersection* of synchronous implicants $c(I_c)$ and $d(I_d)$ is the set of synchronous implicants obtained by pairwise intersection of each element in $c(I_c)$ with each element in $d(I_d)$:

$$c(I_c) \overset{s}{\cap} d(I_d) = \left\{ e(i_e) \mid e(i_e) = c(i_c) \overset{s}{\cap} d(i_d);\ i_c \in I_c, i_d \in I_d \right\}$$

Example: $x_1 x_2^1(0,1) \overset{s}{\cap} x_1 x_2(1) = \{x_1 x_1^1 x_2^1(0), x_1 x_2 x_2^1(1)\}$.

Synchronous Containment: A synchronous implicant $c(i_c)$ *synchronously contains* another implicant $d(i_d)$, $c(i_c) \overset{s}{\supseteq} d(i_d)$, if and only if ${}^{i_c}[c] \supseteq {}^{i_d}[d]$. A synchronous implicant set $c(I_c)$ contains another $d(I_d)$, $c(I_c) \overset{s}{\supseteq} d(I_d)$, if and only if

$$\forall i_d \in I_d, {}^{i_d}[d] \subseteq \bigcup_{i_c \in I_c} c(i_c)$$

The following two results ensure that a minimum-implicant solution has minimum cost.

Proposition A.1 $c(i_c) \overset{s}{\supseteq} d(i_d) \Rightarrow |c| \leq |d|$.

Proof. ${}^{i_c}[c] \supseteq {}^{i_d}[d] \Rightarrow \left|{}^{i_c}[c]\right| \leq \left|{}^{i_d}[d]\right|$. The time-shift operator preserves cost ($|c| = |{}^{x}[c]|$), so $|c| \leq |d|$. □

Proposition A.2 $c(I_c) \overset{s}{\supseteq} d(I_d) \Rightarrow |c| \leq |d|$.

Proof. Assuming $I_d \neq \phi$, ${}^{i_d}[d] \subseteq \cup_{i_c \in I_c} {}^{i_c}[c]$ and Proposition A.1 together imply that $|c| \leq |d|$. □

Functional Equality of Synchronous Implicants: $c(I_c) = d(I_d)$ if and only if $c(I_c) \overset{s}{\supseteq} d(I_d)$ and $d(I_d) \overset{s}{\supseteq} c(I_c)$. If $c(I_c)$ and $d(I_d)$ are simple, then $c(I_c) = d(I_d) \Longleftrightarrow c = d$ and $I_c = I_d$.

Note that this is functional equality in the synchronous Boolean space. That is, if $c(I_c)$ and $d(I_d)$ are equal, they contain precisely the same minterms in the space $(B \times R)^n$.

Implementation Set: A simple synchronous implicant $c(I_c)$ has $i_{c_{max}} + 1$ implementations to be considered. The implementations are denoted $Imp[c(I_c)]$. Let $I_c - i$ denote the set I_c of integers obtained by subtracting i from each and removing negative integers:

$$I_c - i = \{j \mid i_c \in I_c,\ j = i_c - i,\ j \geq 0\}$$

Example: $\{0, 2\} - 1 = \{1\}$.

All implementations can be generated as follows:

$$Imp[c(I_c)] = \{d(I_d) \mid d(I_d) = {}^{i}[c]\,(I_c - i), i = 0, 1, \ldots, i_{c_{max}}\}$$

Example: $x_1 x_2^1(1,2)$ represents two synchronous implicants, $x_1^1 x_2^2$ and $x_1^2 x_2^3$. The implementations of these implicants are $(x_1^1 x_2^2)^0$, $(x_1 x_2^1)^1$, $(x_1^2 x_2^3)^0$, $(x_1^1 x_2^2)^1$, $(x_1 x_2^1)^2$. These are represented by the implicants $x_1 x_2^1(1,2)$, $x_1^1 x_2^2(0,1)$, $x_1^2 x_2^3(0)$ which are generated by the formula above.

Synchronous Distance: The *synchronous distance* between two implicants $c(I_c)$ and $d(I_d)$ is $\delta(c(I_c), d(I_d))$, where

$$\delta(c(I_c), d(I_d)) = \delta(c, d) + \delta(I_c, I_d), \text{ and}$$

$$\delta(I_c, I_d) = \begin{cases} 1 & \text{if } I_c \cap I_d = \phi \\ 0 & \text{otherwise} \end{cases}$$

$\delta(i_c, i_d)$ is the same as the combinational case.

Synchronous Implementation Consensus: The *synchronous implementation consensus* of implementations $c(I_c)$ and $d(I_d)$, $e(I_e) = c(I_c) \overset{s}{\odot} d(I_d)$, is given by

$$\begin{array}{ll} e(I_e) = \phi & \text{if } \delta(c(I_c), d(I_d)) \geq 2 \\ e(I_e) = \{(c \odot d)(I_c \cap I_d)\} & \text{if } \delta(c(I_c), d(I_d)) = 1 \text{ and } \delta(c, d) = 1 \\ e(I_e) = \{(c \cap d)(I_c \cup I_d)\} & \text{if } \delta(c(I_c), d(I_d)) = 0 \text{ or} \\ & \delta(c(I_c), d(I_d)) = 1 \text{ and } \delta(I_c, I_d) = 1 \end{array}$$

Examples: $x_1(1) \overset{s}{\odot} x_2(2) = x_1 x_2(1,2)$; $x_1(0,1) \overset{s}{\odot} x_2(1) = x_1 x_2(0,1)$; $x_1 x_2^2(0) \overset{s}{\odot} \bar{x}_1 x_2^2(0,1) = x_2^2(0) = x_2(2)$.

Synchronous Implementation Set Consensus: The *synchronous implementation set consensus* of $c(I_c)$ and $d(I_d)$, $c(I_c) \overset{ss}{\odot} d(I_d)$ is the synchronous implicant consensus of each implementation in $Imp[c(I_c)]$ with each implementation in $Imp[d(I_d)]$:

$$c(I_c) \overset{ss}{\odot} d(I_d) = \{g(I_g) \quad | \quad g(I_g) = e(I_e) \overset{ss}{\odot} f(I_f) \\ e(I_e) \in Imp[c(I_c)],\ f(I_f) \in Imp[d(I_d)]\}$$

Theorem 5.1 Iterated synchronous implementation set consensus generates all synchronous primes.
Proof. Suppose

$$\begin{aligned} f &= c_1(I_{c_1}) + c_2(I_{c_2}) + \ldots + c_n(I_{c_n}) \\ Imp[f] &= Imp[c_1(I_{c_1})] + Imp[c_2(I_{c_2})] + \ldots + Imp[c_n(I_{c_n})] \end{aligned}$$

Suppose $c(I_c)$ is a prime and $I_c = \{i_{c_1}, i_{c_2}, \ldots, i_{c_k}\}$. Then $\forall i_{c_j} \in I_c$, $c(i_{c_j}) = ^{i_{c_j}}[c] \subseteq f$. Consider all the implementations in time frame i_{c_j}, that is, all implicants $d(i_{c_j})$ such that $d(i_{c_j}) \in d(I_d) \in Imp[f]$.

Since synchronous consensus is identical to combinational consensus when all implicants are in a single, common time frame, the iterated consensus of all $d(i_{c_j})$ above must generate some cube $e_j(i_{c_j})$ that contains $c(i_{c_j})$ since $c(i_{c_j}) \subseteq f$. $e_j(i_{c_j}) \supseteq c(i_{c_j})$ and $e \supseteq c$. This is true for all $i_{c_j} \in I_c$, and the consensus of all of these $e_j(i_{c_j})$ cubes is $e_1 \cap e_2 \cap \cdots \cap e_k(i_{c_1} \cup i_{c_2} \cup \ldots \cup i_{c_k}) = g(I_c)$, where g is a cube and $g \supseteq c$. Since $c(I_c)$ is a prime, $g = c$ and consensus produces the prime $c(I_c)$. □

In practice, not all pairs of implementations of two implicants need to be considered. In fact, there are a number of ways of improving the efficiency of synchronous prime generation using synchronous containment and synchronous consensus. These are not given here as the actual implementation was done using combinational prime generation, which is well-developed and very efficient. The implementation details are in section 6.

B Proposition Proofs

Proposition 5.1: All implementations of a synchronous cube have the same cost with respect to a minimum-gate implementation.
Proof. A synchronous cube c can be implemented in $1 + j_{min}(c)$ ways:

$$(c), \left(^{-1}[c]\right)^1, \left(^{-2}[c]\right)^2, \ldots, \left(^{-j_{min}(c)}[c]\right)^{j_{min}(c)}$$

Each requires implementation of cube c with cost $|c|$. □

Proposition 5.2: A synchronous implicant costs no more than any synchronous implicant it contains with respect to a minimum-gate implementation.
Proof. Suppose $c(I_c) \overset{s}{\supseteq} d(I_d)$ and $I_d \neq \phi$. All implementations of $c(I_c)$ cost $|c|$ and all implementations of $d(I_d)$ cost $|d|$. By Proposition A.2, $|c| \leq |d|$. □

Proposition 6.2: All implementations of a synchronous cube have the same cost with respect to a minimum-gate implementation.
Proof. Suppose $c(I_c)$ is a prime and c is simple. Then $\forall i_c \in I_c$, $^{i_c}[c]$ is an implicant of f and is independent of variables $\mathbf{x}^i$, $i = 0, 1, \ldots, i_c - 1$. Furthermore, c is an implicant of f_{i_c} since f_{i_c} is the largest function contained by f and independent of variables x^i, $i = 0, 1, \ldots, i_c - 1$. Iterated consensus of the cubes in f_{i_c} must produce either c or some cube $d \supseteq c$.

Now consider all implicants in $c(I_c)$. Let $I_c = \{i_1, i_2, \ldots, i_m\}$ and let d_{i_1} be a prime of f_{i_1} that contains c. Iterated consensus produces $d_{i_1} \in f_{i_1}, d_{i_2} \in f_{i_2}, \ldots, d_{i_m} \in f_{i_m}$. Since $d_{i_j} \supseteq c$, $j = 1, 2, \ldots, m$, the consensus of these cubes is contained by the synchronous cube $c(i_1, i_2, \ldots, i_m) = c(I_c)$. In fact, it is precisely equal to since $c(I_c)$ since $c(I_c)$ was assumed prime *a priori.* □