An Efficient Implementation of Boolean Functions as Self-Timed Circuits

Ilana David, Ran Ginosar, *Member, IEEE*, and Michael Yoeli

Abstract— **Self-timed logic provides a method for designing logic circuits such that their correct behavior depends neither on the speed of their components nor on the delay along the communication wires. This paper proposes a general synthesis method for *efficiently* implementing any family of Boolean functions over a set of variables, as a self-timed logic module. Interval temporal logic is used to express the constraints that are formulated for the self-timed logic module. A method is provided for proving the correct behavior of the designed circuit, by showing that it obeys all the functional constraints. The resulting circuit is compared with alternative proposed self-timed methodologies. Our approach is shown to require less gates than the other methods. Our proposed method is appropriate for automatic synthesis of self-timed systems.**

Index Terms— **Asynchronous systems, combinational logic, delay-insensitive, self-timed, temporal logic, verification.**

I. Introduction

SELF-TIMED logic provides a method for designing asynchronous logic circuits such that their correct behavior depends neither on the speed of their components nor on the delay along the communication wires. We refer to [18] for an extensive discussion of self-timed logic and its advantages as compared with globally clocked, or synchronous, logic. Seitz [18] illustrates a possible approach to the self-timed implementation of Boolean functions by showing the design of a self-timed adder circuit. Other methods of designing self-timed circuits can be found in [19] and [1]. Both methods, as well as Seitz's example, employ an excessive number of gates. Also, no formal proof for the correctness of the constructed circuit is provided. An interesting framework for applying temporal logic to the formalization of the approach of [18] is proposed in [13]. However, no design method is suggested, nor is any correctness proof established.

In this paper we propose a general synthesis method for *efficiently* implementing any family of Boolean functions over a set of variables, as a self-timed logic module. Temporal logic is employed to express the constraints that we formulate for the self-timed logic module. We provide a method for proving the correct behavior of our circuit, by showing that it obeys all the functional constraints. The proposed method is appropriate for automatic synthesis of self-timed systems. The method is extended to the synthesis of finite-state machines (FSM) in the companion paper [6].

The problem of implementing self-timed Boolean functions is informally described in Section II. In Section III, we introduce a formal approach to the problem specification. We use a modified version of temporal logic for this purpose. In Section IV we formulate the concept of "double-rail self-timed" implementation in a precise way. In Section V, we describe a method for implementing a given set of Boolean functions, as an efficient double-rail, self-timed circuit. The implementation is obtained by the interconnection of four subnets. A detailed description of each subnet using temporal logic is given. An example of this method and its comparison with previously published methods is given in Section VI. In Section VII, a formal proof of correctness is provided. Sections VIII and IX provide a detailed discussion of the merits of our approach.

Manuscript received July 13, 1989; revised April 23, 1991.

I. David is with the Department of Electrical Engineering, Technion–Israel Insitute of Technology, Haifa 32000, Israel.

R. Ginosar is with the Department of Electrical Engineering and the Department of Computer Science, Technion–Israel Institute of Technology, Haifa 32000, Israel.

M. Yoeli is with the Department of Computer Science, Technion–Israel Institute of Technology, Haifa 32000.

IEEE Log Number 9103032.

II. Double-Rail Self-Timed Implementation of Boolean Functions

Let $f_1, \cdots, f_m$ be $m \geq 1$ Boolean functions, each on $n \geq 1$ variables, i.e., $f_i : \{0,1\}^n \rightarrow \{0,1\}$, for $1 \leq i \leq m$. We call a ternary, asynchronous network $\hat{N}$ a *ternary ST* (self-timed) *implementation* of $f_1, \cdots, f_m$ if the following conditions are met:

1) $\hat{N}$ has n three-valued inputs $\hat{x}_1, \cdots, \hat{x}_n$ and m three-valued outputs $\hat{f}_1, \cdots, \hat{f}_m$. Each input and output may assume any value from the set $\{0, 1, U\}$. We refer to U as the *undefined* value and to $0, 1$ as the *defined* values.
2) If all the inputs and outputs are *defined* (i.e., 0 or 1), then
$$\hat{f}_i = f_i(\hat{x}_1, \cdots, \hat{x}_n), 1 \leq i \leq m.$$
3) The sequential behavior of the network $\hat{N}$ and of its environment is specified by the following cycle of activities. The E_i's are environment (domain) constraints. The S_i's are network (functional) constraints. S_i is to follow E_i $(1 \leq i \leq 4)$ and $E_{(i+1)}$ is assumed to follow S_i $(1 \leq i \leq 3)$. Each cycle is started by $E1$ and terminates with $S4$. Thus, $S4$ is to be followed by $E1$, to start a new cycle:

$E1$. All inputs are set to *undefined*.
$S1$. All outputs become *undefined*.
($E1$ and $S1$ constitute the inactive steady state.)

$E2$. Some (but not all) inputs become *defined*.
$S2$. All outputs remain *undefined*.
(Partial outputs should be blocked, in order to assure proper operation of composite ST system.)
$E3$. All inputs become *defined*.
$S3$. All outputs become *defined*.
($E3$ and $S3$ constitute the active steady state.)
$E4$. Some (but not all) inputs become *undefined*.
$S4$. All outputs remain *defined*.

$S4$, in spite of $E4$, assures proper ST operation of other ST systems to which $\hat{N}$ may be connected. The outputs may be required, and should be retained, until all inputs become undefined. The intermediate steps of the environment $E2$ and $E4$ may be skipped. Thus, the network $\hat{N}$ can also behave in accordance with the cycles $(E1, S1, E3, S3, E1)$, or $(E1, S1, E2, S2, E3, S3, E1)$, or $(E1, S1, E3, S3, E4, S4, E1)$.

A ternary network such as $\hat{N}$ may be implemented physically by applying three-valued logic (cf. [5]), where three voltage ranges are used to represent the logic values $0, U, 1$. Instead, we follow [18] and employ double-rail code, in which the three logic values $0, U, 1$ are represented by $10, 00, 01$, respectively. Let N be a binary asynchronous network representing $\hat{N}$ in accordance with the above double-rail code. Thus, N has $2n$ binary inputs and $2m$ binary outputs, representing the n ternary inputs and m ternary outputs of $\hat{N}$. Furthermore, N satisfies the above conditions 2) and 3) of $\hat{N}$. We shall refer to N as a DR (double-rail)-ST implementation of $f_1, \cdots, f_m$. Thus, this paper is concerned with obtaining an efficient double-rail ST implementation N for any given set of n-variable Boolean functions $f_1, \cdots, f_m$.

III. Definitions

In this section we introduce a formal approach which will enable us to state the problem and its solution in a precise way and to provide a suitable proof of correctness. Our approach is based on a modified and simplified version of the Interval Temporal Logic (ITL) introduced in [9] and [15]. ITL is a generalization of the syntax and semantics of standard temporal logics [16], [10]. Our version is particularly tailored toward the description and analysis of self-timed networks. We employ the usual propositional connectives NOT, AND, OR, IMPLIES (with descending priorities) and denote them by $\neg, \wedge, \vee, \rightarrow$, respectively. Furthermore, we introduce two temporal operators. The first one is $\trianglelefteq$, taken over from [10]. The formula $P \trianglelefteq Q$ is interpreted to mean "Q remains true at least as long as P does." The operator $\trianglelefteq$ is related to the more familiar until operator by $P \trianglelefteq Q \equiv Q$ *until* $\neg P$. The until operator in question is the so called *weak* version [10], namely P *until* Q does not assert that Q will eventually occur. If J is a continuous time interval starting at time t_o and terminating at time t_1 ($> t_o$), then $P \trianglelefteq Q$ is valid for J (notation: $J \models P \trianglelefteq Q$) iff the following holds: If P is true continuously from time t_0 up to time t' ($t_o \leq t' < t_1$) then Q is also true continuously from time t_o up to time t' or longer.

Note that the known temporal operators $\Box$ ("henceforth") and $\Diamond$ ("eventually") may be expressed by means of the operator $\trianglelefteq$ [10]. Namely $\Box P \equiv$ *true* $\trianglelefteq P$ and $\Diamond P \equiv \neg\Box\neg P$.

The other temporal operator we introduce is "$\mapsto$". This operator is used, in a somewhat different sense, in [11]. We interpret the formula $P \mapsto Q$ to mean the following: If P is true and "remains true long enough," then Q will eventually become true and stay true at least as long as P remains true. The formula $P \mapsto Q$ could be approximated by the following formula, using conventional temporal logic: $\Box(\Box P \rightarrow \Diamond\Box Q)$. However, the formula $P \mapsto Q$ does not require that P stays true forever.

To illustrate the use of the operator $\mapsto$ consider a multilevel combinational network with an unknown, but finite, propagation delay. If we apply a particular input combination, say X, then after the propagation delay in question, the output will obtain the correct value, say Z. This output value will not change, at least as long as the input does not change. We describe this situation by

$$\textit{input} = X \mapsto \textit{output} = Z.$$

In addition to the temporal operators introduced so far, we shall also need means to reason about finite time intervals. Although the intervals we consider can be viewed as closed continuous time intervals, we take over the following concepts and notations from ITL (Interval Temporal Logic [15]), where the time intervals considered are discrete.

Let J be a closed (continuous) time interval of finite, nonzero length. We write $J \models P$ to state that assertion P holds at the beginning of J. Similarly, $J \models finP$ asserts that P is true at the end of J, and $J \models \Box P$ asserts that P is true throughout the interval J. We write $J \models x \approx y$ instead of $J \models \Box x = y$. Assume now that x is a binary variable with value 0 at the start of J (i.e., $J \models x = 0$) and final value 1 (i.e., $J \models finx = 1$). Furthermore assume that x changes its value only once (from 0 to 1) during the interval J. This is simply denoted by $J \models\uparrow x$. We define $J \models\downarrow x$ similarly.

We now illustrate the concepts and notations introduced in this section by means of the two-input C-element [14,] [18]. The C-element (also called "rendezvous" element) is an essential building block of various kinds of speed-independent and self-timed circuits. It behaves as follows: When all its inputs assume the same value, the output also assumes this value; otherwise, the output value does not change. Let $CE2(a, b; z)$ denote a two-input C-element with inputs a, b and output z. Then the conjunction of the following properties can be considered as a definition of $CE2(a, b; z)$. $PCE1$ states that if the two inputs are equal and stay equal long enough, then the output will assume the same value. $PCE2$ and $PCE3$ state that if the output value equals the value of one of the inputs, then the output value will not change as long as the corresponding input value does not change.

$$(PCE1) \quad a = b \mapsto z = a$$
$$(PCE2) \quad a = 0 \wedge z = 0 \rightarrow (a = 0 \trianglelefteq z = 0)$$
$$(PCE3) \quad a = 1 \wedge z = 1 \rightarrow (a = 1 \trianglelefteq z = 1).$$

In $(PCE2)$ and $(PCE3)$ a may be replaced by b.

Properties $(PCE2)$ and $(PCE3)$ can also be formulated by means of intervals. Let J be an arbitrary, finite interval of positive length. Then

$(PCE2')$ $(J \models a = z = 0) \wedge (J \models a \approx 0) \rightarrow (J \models z \approx 0)$

$(PCE3')$ $(J \models a = z = 1) \wedge (J \models a \approx 1) \rightarrow (J \models z \approx 1)$

$PCE2'$ assumes that at the beginning of the interval J $a = z = 0$ and that a stays at the value 0 throughout the interval. Under this assumption, the output z will remain at the value 0 throughout the interval. $PCE3'$ can be explained similarly.

IV. Formal Specification of DR-ST Implementation

In this section we rephrase the concept of DR-ST implementation in a more formal way. Let N be a *double-rail* (n, m) network, i.e., a binary network with input set $I = \{x_1^0, x_1^1, \cdots, x_n^0, x_n^1\}$, $n \geq 1$, and output set $O = \{f_1^0, f_1^1, \cdots, f_m^0, f_m^1\}$, $m \geq 1$. We set $\underline{x}_i = (x_i^0, x_i^1)$, $1 \leq i \leq n$, and $\underline{f}_j = (f_j^0, f_j^1)$, $1 \leq j \leq m$. Let $0, 1$, and U be the ternary equivalent of the 2-bit ("double-rail") combination $10, 01$, and 00, respectively. We assume that always $\underline{x}_i \neq 11$ and denote by $\hat{x}_i$ the ternary equivalent of $\underline{x}_i$. $\hat{f}_j$ is defined similarly. $\underline{x}_i$ is "undefined" iff $\underline{x}_i = 00$ and "defined" iff $\underline{x}_i \in \{10, 01\}$. This terminology also applies to $\underline{f}_j$. Following [13], we write $D(I)$ to state that all the $\underline{x}_i$'s are "defined"; $D(O)$ is used similarly for the $\underline{f}_j$'s.

Speaking informally, we say that the network N is stable [notation: *stable*(N)] iff its "internal state" and outputs do not change as long as its inputs remain unchanged. The internal state is the set of values of all the nodes of the network N which are neither inputs nor outputs. We consider the concept "stable" to be defined implicitly by the following stability rules, viewed as axioms. Let *incond*, *intcond*, and *outcond* be conditions met by the inputs, internal state, and outputs of N, respectively. Then the following "stability rules" evidently hold:

$(STR1)$ **if** *incond* $\mapsto$ *outcond*
then (*incond* $\wedge$ *stable*(N)) $\rightarrow$ *outcond*

$(STR2)$ **if** *incond* $\mapsto$ *intcond*
then (*incond* $\wedge$ *stable*(N)) $\rightarrow$ *intcond.*

$STR1$ states the following: if some input condition eventually leads to some output condition, and if this input condition presently holds and the network is already stable, then the relevant output condition also holds presently. $STR2$ is explained similarly.

In the informal behavioral description of network $\hat{N}$ in Section II, no explicit reference to stable states has been made. However, we require that the environment will apply activities $E2$ and $E4$ to the network only after it has reached a stable state. This mode of operation is known as *fundamental* mode, versus *input–output* mode, in which the environment may change the inputs as soon as the correct output values have been obtained [2]. Under realistic circumstances, it may be assumed that the two modes coincide (see also Section VIII).

In order to achieve a formal, behavioral specification of the network N of Section II, we now introduce the concepts of defining interval ($D\uparrow$-interval) and undefining interval ($D\downarrow$-interval). Given a set X of binary variables and a binary constant $c \in \{0, 1\}$, we write $AX = c$ instead of $\forall x \in X : x = c$. Thus, when $AI = 0$, all inputs are undefined. An interval J is a defining interval (with respect to N), iff the following conditions are met.

$(D\uparrow 1)$ $J \models AI = 0 \wedge$ *stable* (N)

$(D\uparrow 2)$ $J \models$ *fin*$[D(I) \wedge$ *stable*$(N)]$

$(D\uparrow 3)$ $\forall x \in I : J \models x \approx 0 \vee \uparrow x.$

$D\uparrow 1$ states that at the beginning of the interval J all inputs are undefined and the network is stable. $D\uparrow 2$ states that at the end of the interval all inputs are defined and the network has again reached stable state. $D\uparrow 3$ requires that any binary input either stays at the value 0 during the whole interval, or changes its value exactly once.

An *undefining interval* J is specified by

$(D\downarrow 1)$ $J \models D(I) \wedge D(O) \wedge$ *stable*(N)

$(D\downarrow 2)$ $\forall x \in I : J \models x \approx 0 \vee \downarrow x.$

$D\downarrow 2$ means that during the undefining interval each binary input either stays at the value 0, or changes from 1 to 0 exactly once.

We now have the necessary tools for a formal behavioral specification of N.

Given m Boolean functions $f_1, \cdots, f_m$ on n variables, we say that the double-rail (n, m) network N is a DR-ST implementation of $f_1, \cdots, f_m$ iff the following conditions are met:

$(DR1)$ $AI = 0 \mapsto AO = 0$

$(DR2)$ $AI = 0 \wedge$ *stable*$(N) \rightarrow \neg D(I) \trianglelefteq AO = 0$

$(DR3)$ Let J be a *defining interval* (with respect to N).
Then $J \models$ *fin*$[D(O) \wedge \forall j \in \{1, \cdots, m\} :$
$\hat{f}_j = f_j(\hat{x}_1, \cdots, \hat{x}_n)]$

$(DR4)$ Let J be an *undefining interval* (with respect to N)
Then $J \models [\neg AI = 0 \trianglelefteq D(O)]$.

Stated informally, the above conditions may be interpreted as follows:

$DR1$: If all inputs are undefined, eventually all outputs will become undefined.

$DR2$: If at some instant of time all the inputs are undefined and the network is stable, then the outputs are undefined and will remain undefined until all the inputs become defined.

$DR3$: If J is a defining interval, then at the end of the interval J, all the outputs are defined and assume the required value.

$DR4$: If J is an undefining interval, then all the outputs remain defined until all the inputs become undefined.

V. Efficient DR-ST Implementation

In this section we describe a method for obtaining an efficient DR-ST implementation for any given finite set of n-variable Boolean functions $f_1, \cdots, f_m$. The implementation of N consists of four subnets interconnected as shown in Fig. 1.

Subnet ORN detects, for each input, whether this input is defined or undefined; subnet CEN indicates that all the inputs became defined or all the inputs became undefined; subnet DRN actually implements the combinational functions; subnet OUTN ensures that the outputs remain defined as long as not all inputs have reached their undefined state. A detailed description of each of the four subnets follows.

Subnet ORN

This subnet consists of n two-input OR-gates. The ith OR-gate $(1 \leq i \leq n)$ has inputs x_i^0 and x_i^1, and output w_i. Thus,

$$I(ORN) = I(N) = I$$
$$O(ORN) = \{w_1, \cdots, w_n\} = W.$$

The following conditions are met by the subnet ORN (recall the assumption that $x_i^0 = x_i^1 = 1$ never occurs).

$$(CORN1) \quad AI = 0 \mapsto AW = 0$$
$$(CORN2) \quad \neg D(I) \trianglelefteq \neg AW = 1$$
$$(CORN3) \quad D(I) \mapsto AW = 1$$
$$(CORN4) \neg AI = 0 \trianglelefteq \neg AW = 0.$$

$CORN1$ and $CORN3$ state that if all inputs are undefined (defined), then eventually all the outputs of ORN will become 0 (1).

$CORN2$ and $CORN4$ state that as long as all the inputs are not yet defined (undefined), not all the outputs of ORN equal 1 (0).

Subnet CEN

CEN consists of a single n-input C-element. Its inputs are $w_1, \cdots, w_n = W$ and its output is y. It satisfies the following conditions.

$$(CCEN1) \quad AW = 0 \mapsto y = 0$$
$$(CCEN2) \quad AW = 1 \mapsto y = 1$$
$$(CCEN3) \quad \neg AW = 1 \wedge y = 0 \rightarrow (\neg AW = 1 \trianglelefteq y = 0)$$
$$(CCEN4) \quad \neg AW = 0 \wedge y = 1 \rightarrow (\neg AW = 0 \trianglelefteq y = 1).$$

$CCEN1$ and $CCEN2$ state that if all the inputs of CEN are equal, then the output y will eventually assume the common value of the inputs. $CCEN3$ and $CCEN4$ indicate that under any other condition, the output remains unchanged.

Subnet DRN

This is a combinational double-rail (n, m) network with input $I(DRN) = I(N) = I$ and output $O(DRN) = \{F_1^0, F_1^1, \cdots, F_m^0, F_m^1\} = F$. DRN is to satisfy the following conditions:

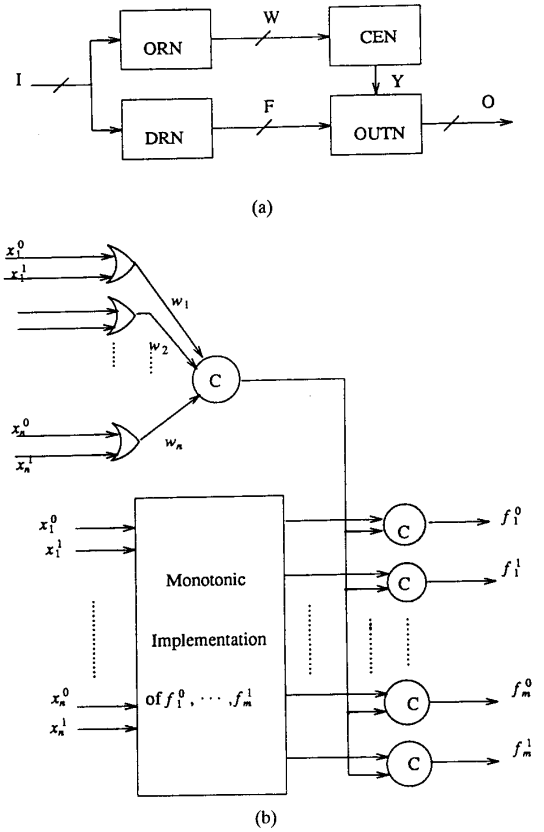


Fig. 1. The general structure of an efficient implementation of self-timed combinational circuit. Subnet DRN is a monotonic implementation of all the required Boolean functions and their inverses. The other subnets do not depend on the functions themselves, only on the number of inputs and functions; their task is to guarantee self-timed operation.

$$(CDRN1) \quad AI = 0 \mapsto AF = 0$$
$$(CDRN2) \quad D(I) \mapsto D(F) \wedge \forall j \in \{1, \cdots, m\} : \hat{F}_j = f_j(\hat{x}_1, \cdots, \hat{x}_n)$$
$$(CDRN3) \quad \text{Let } J \text{ be a } \textit{defining interval} \text{ and let } j \in \{1, \cdots, m\}, i \in \{0, 1\}. \text{ Then: } (J \models finF_j^i = 0) \rightarrow (J \models F_j^i \approx 0)$$
$$(CDRN4) \quad \text{Let } J \text{ be an } \textit{undefining interval} \text{ and let } j \in \{1, \cdots, m\}, i \in \{0, 1\}.$$

Then

$$(J \models F_j^i = 0) \rightarrow (J \models F_j^i \approx 0).$$

$CDRN1$ states that once all the inputs of DRN become undefined, then eventually all its outputs (F) will become undefined. $CDR2$ states that if all the inputs of DRN become defined, then eventually all the outputs of DRN (F) will become defined and assume the required function values. $DRN3$ stipulates that if at the end of a defining interval some binary output equals 0, then it was 0 throughout the whole interval (this condition ensures that there are no static hazards), while $CDRN4$ stipulates that if one of the binary outputs of

DRN is 0 at the beginning of an undefining interval, this binary output stays 0 throughout the whole interval.

A network satisfying conditions $CDRN3$ and $CDRN4$ is called *monotonic*. The monotonicity of DRN is essential for the correct behavior of the overall network (as explained in the proof of Lemma 3 below).

An efficient implementation of DRN may be obtained by means of the following procedure. Let $\tilde{N}$ be a binary network having as inputs the variables $x_1, \cdots, x_n$, as well as their complements $x_1', \cdots, x_n'$. $\tilde{N}$ consists of OR gates and AND gates only and produces $2m$ outputs, namely $f_j(x_1, \cdots, x_n)$ and $f_j'(x_1, \cdots, x_n)$ for $j = 1, \cdots, m$. Such a network $\tilde{N}$ evidently exists. We use known techniques to obtain a minimal implementation of $\tilde{N}$. Then, DRN is derived from $\tilde{N}$ by renaming:

1) Inputs x_i and x_i' $(1 \leq i \leq n)$ are renamed x_i^1 and x_i^0, respectively.
2) Outputs f_j and f_j' $(1 \leq j \leq m)$ are renamed F_j^1 and F_j^0, respectively.

The above procedure produces the function f_j and its dual f_j' as independent AND–OR circuits. By the above renaming we obtain the desired double-rail implementation. The double-rail inputs provide the binary variables and their complements, thus no inverters are required inside the DRN circuit. One easily verifies that the subnet DRN thus derived from $\tilde{N}$ indeed satisfies the conditions $(CDRN1) - (CDRN4)$. Note that DRN is the only subnet in this implementation that depends on the functions $f_1, \cdots, f_m$.

Subnet OUTN

This subnet consists of the following $2m$ two-input C-elements:

$$CE2(F_j^0, y; f_j^0)\ ,\ 1 \leq j \leq m$$
$$CE2(F_j^1, y; f_j^1)\ ,\ 1 \leq j \leq m.$$

Thus, $I(OUTN) = F \cup \{y\}$ and $O(OUTN) = O(N) = O$.

In view of the properties of $CE2(a, b; z)$ discussed in Section III, $OUTN$ satisfies the following conditions:

$(COUTN1)$ $AF = 0 \wedge y = 0 \mapsto AO = 0$

$(COUTN2)$ $y = 0 \wedge AO = 0 \rightarrow y = 0 \trianglelefteq AO = 0$

$(COUTN3)$ Let J be an interval and let $j \in \{1, \cdots, m\}$, $i \in \{0, 1\}$. Then
$(J \models F_j^i \approx 0 \wedge J \models f_j^i = 0) \rightarrow J \models f_j^i \approx 0$

$(COUTN4)$ $y = F_j^i \mapsto f_j^i = F_j^i$

$(COUTN5)$ $y = 1 \wedge f_j^i = 1 \rightarrow y = 1 \trianglelefteq f_j^i = 1.$

These conditions state the following: if all the binary inputs to $OUTN$ are 0, then the outputs will become 0 and stay at this value as long as $y = 0$ ($COUTN1$, $COUTN2$). Furthermore, if at beginning of some interval J some f output equals 0, and the corresponding F is 0 throughout this interval, then the f output remains 0 throughout the interval ($COUTN3$). If y equals one of the F inputs, then the corresponding f output will assume the value of the F input ($COUTN4$). If y and one of the f outputs both equal 1 at some instant, then this f output remains 1 as long as y remains 1 ($COUTN5$).

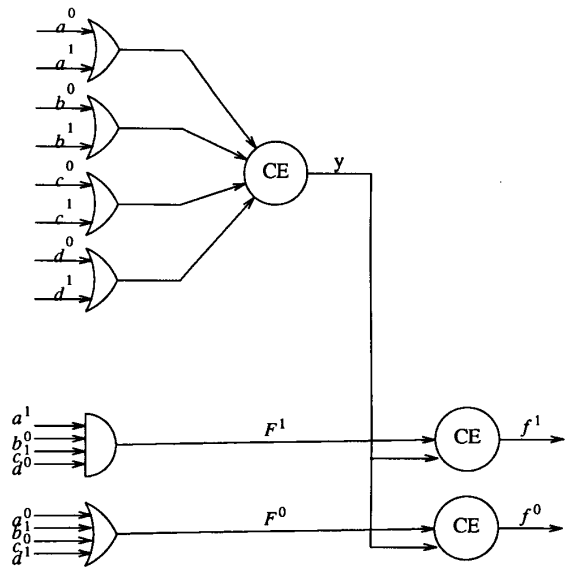


Fig. 2. Example of an efficient implementation of a self-timed combinational circuit of the function $f(a, b, c, d) = a \cdot b' \cdot c \cdot d'$.

Note that the y signal contains forks. Under extremely unlikely conditions, requiring skewed asymmetric delays and long sequences of specific inputs, such forks may cause illegal operation of the network. These circumstances are so unreasonable that it is safe to assume that these forks are isochronic. With respect to the network N, this assumption is a special case of the *fundamental* mode assumption on which our model is based, as further discussed in Section VIII.

VI. An Example

As an example we implement the function $f(a, b, c, d) = a \cdot b' \cdot c \cdot d'$. Following the procedure for efficient implementation of DRN we get

$$F^1 = a^1 \cdot b^0 \cdot c^1 \cdot d^0$$
$$F^0 = a^0 + b^1 + c^0 + d^1.$$

The circuit is shown in Fig. 2. Other methods for implementing DR-ST modules, using double-rail code, can be found in [19] and [1]. Fig. 3 shows the implementations of the above function, using

1) Seitz's example outline [18]
2) Anantharaman's method [1]
3) Singh's method [19].

The four implementations are compared by number of gates in Table I. In order to compare the number of gates used in each implementation, we count the gates and CE's in each circuit and replace each gate or C-element by its two-input-gate equivalent. Since the CE function is associative, we use

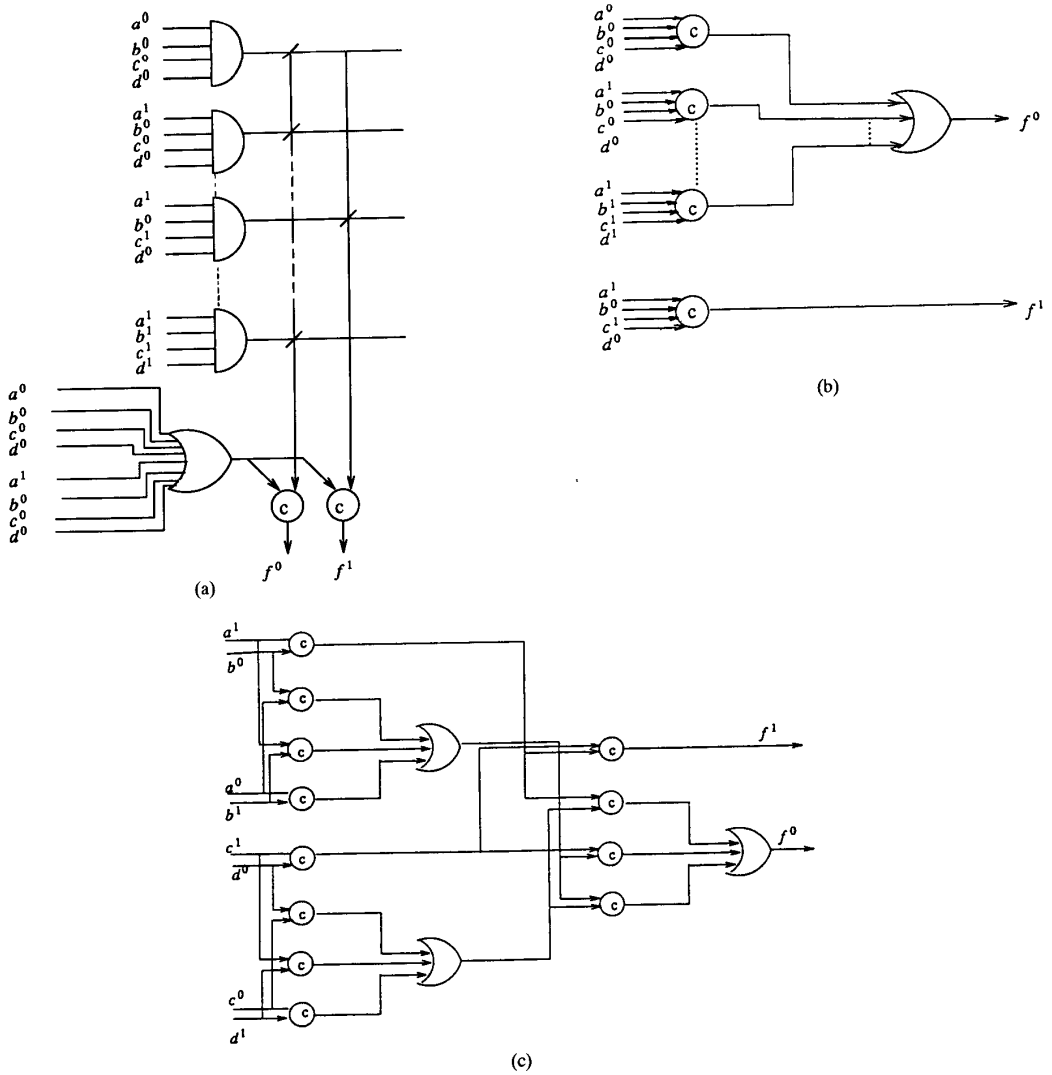


Fig. 3. Implementing the function of Fig. 2, using methods proposed by (a) Seitz [18], (b) Anantharaman [1], and (c) Singh [19]. The implementation of Fig. 2 requires a smaller number of gates (Table I).

the scheme

$$1CE_n \equiv (n-1)CE_2$$
$$1CE_2 \equiv 4G_2$$
$$1G_n \equiv (n-1)G_2.$$

(G_2 denotes two-input gate; G_n denotes n-input gate.)

This example shows that our method uses considerably less gates than the other methods. We discuss the reasons for this advantage in Section VIII below. The non-self-timed implementation of this example would require only three two-input gates; however, the difference in the gate count between conventional implementations and our self-timed circuits becomes less significant for large circuits.

VII. Proof of Correctness

In this section we provide a formal proof of correctness. The proof is based on the following rules of inference:

$(R1)$ $P \mapsto Q, Q \mapsto R \vdash P \mapsto R$

$(R2)$ $P \mapsto Q, P \mapsto R \vdash P \mapsto Q \wedge R$

$(R3)$ $P \trianglelefteq Q, Q \trianglelefteq R \vdash P \trianglelefteq R.$

As for intervals, we have

$(RJ1)$ $J \models P, P \rightarrow Q \vdash J \models Q$

$(RJ2)$ $J \models finP, P \rightarrow Q \vdash J \models finQ$

$(RJ3)$ $J \models P, J \models Q \vdash J \models P \wedge Q$

$(RJ4)$ $J \models finP, J \models finQ \vdash J \models fin(P \wedge Q).$

Theorem: The implementation network N of Section V is correct, i.e., satisfies the conditions $(DR1)$ – $(DR4)$ of Section IV.

Proof: We need the following lemmas.

TABLE I
GATE COUNT COMPARISON OF FOUR SELF-TIMED COMBINATIONAL LOGIC IMPLEMENTATIONS

	Gate and Element Count	Two-Input Gate Equivalent
Seitz's circuit	2 CE_2 + 16 AND_4 + 1 OR_{15} + OR_8	77
Anantharaman's circuit	16 CE_4 + 1 OR_{15}	206
Singh's circuit	12 CE_2 + 3 OR_3	54
Our circuit	1 CE_4 + 1 CE_2 + 1 AND_4 + 1 OR_4	30

Lemma 1: The implementation network N of Section V satisfies conditions $(DR1), (DR2)$.

The formal proof given below is based on the following sequence of arguments [see Fig. 1(b)]: Let all inputs of the network be undefined. Then, all the w_i's will become 0 and consequently y will also become 0. In view of $CDRN1$, all F's will become 0. $y = 0$ and all F's equal 0 will eventually force all f outputs to become 0 $(DR1)$. The f outputs will remain 0 as long as $y = 0$; $y = 0$ as long as not all w_i's have become 1, i.e., not all inputs have become defined $(DR2)$.

Proof:

$[1]AI = 0 \mapsto AW = 0$ $\{(CORN1)\}$
$[2]AW = 0 \mapsto y = 0$ $\{(CCEN1)\}$
$[3]AI = 0 \mapsto y = 0$ $\{[1], [2], (R1)\}$
$[4]AI = 0 \mapsto AF = 0$ $\{(CDRN1)\}$
$[5]AI = 0 \mapsto y = 0 \wedge AF = 0$ $\{[3], [4], (R2)\}$
$[6]AF = 0 \wedge y = 0 \mapsto AO = 0$ $\{(COUTN1)\}$
$[7]AI = 0 \mapsto AO = 0$ $\{[5], [6], (R1)\}$

$(DR1) = [7]$ is thus proven. □

$[8]AI = 0 \wedge$ *stable*(N) {*Assumption*}
$[9]AW = 0$ $\{[1], [8], (STR2)\}$
$[10]y = 0$ $\{[3], [8], (STR2)\}$
$[11]AO = 0$ $\{[7], [8], (STR1)\}$
$[12]\neg D(I) \trianglelefteq \neg AW = 1$ $\{(CORN2)\}$
$[13]\neg AW = 1 \trianglelefteq y = 0$ $\{[9], [10], (CCEN3)\}$
$[14]y = 0 \trianglelefteq AO = 0$ $\{[10], [11], (COUTN2)\}$
$[15]\neg D(I) \trianglelefteq AO = 0$ $\{[12], [13], [14], (R3)\}$
$[16]AI = 0 \wedge$ *stable*$(N) \rightarrow$
$\neg D(I) \trianglelefteq AO = 0$ $\{[8], [15]\}$.

Since $(DR2) = [16]$, Lemma 1 is proven. □

Lemma 2: The implementation network N of Section V satisfies condition $(DR3)$.

The formal proof of this lemma is based on the following sequence of arguments:

Let J be a defining interval. Then, at the beginning of J all inputs are undefined, and, by Lemma 1, all outputs are undefined. Thus, all the f_j^i equal 0. By $(D \uparrow 2)$ all inputs are defined at the end of the interval J, and the network N is stable. It follows that all F_j are defined. Assume for example that $F_j^0 = 0$ and $F_j^1 = 1$. Thus, F_j^0 was 0 during the whole interval. Consequently, f_j^0 was also 0 during the whole interval, and therefore has the value 0 at the end of the interval. At the end of interval J, all inputs are defined, and therefore all the w_i equal 1, yielding $y = 1$. Also, at the end of the interval, $F_j^1 = 1$. It follows that $f_j^1 = 1$. Thus, the output f_j is defined at the end of the interval and assumes the correct value. The case $F_j^0 = 1$ and $F_j^1 = 0$ is treated in the same way.

Proof: Let J be a *defining interval* (with respect to N). Thus,

$[1]J \models [AI = 0 \wedge$ *stable*$(N)]$ $\{D \uparrow 1\}$
$[2]J \models fin[D(I) \wedge$ *stable*$(N)]$ $\{D \uparrow 2\}$
$[3]\forall x \in I : J \models x \approx 0 \vee \uparrow x$ $\{D \uparrow 3\}$
$[4]D(I) \wedge$ *stable*$(N) \rightarrow D(F)$ $\{(CDRN2), (STR2)\}$
$[5]J \models finD(F)$ $\{[2], [4], (RJ2)\}$.

Let now $j \in \{1, \cdots, m\}$. $D(F)$ implies that either $F_j^0 = 0 \wedge F_j^1 = 1$ or $F_j^0 = 1 \wedge F_j^1 = 0$. We consider the first case. The other case is treated in the same way.

$[6]J \models fin(F_j^0 = 0 \wedge F_j^1 = 1)$ $\{[5], Case1\}$
$[7]J \models F_j^0 \approx 0$ $\{[1], [2], [3], [6], (CDRN3)\}$
$[8]AI = 0 \wedge$ *stable*$(N) \rightarrow f_j^0 = 0$ {*Lemma 1*}
$[9]J \models f_j^0 = 0$ $\{[1], [8], (RJ1)\}$
$[10]J \models f_j^0 \approx 0$ $\{[7], [9], (COUTN3)\}$
$[11]J \models finf_j^0 = 0$ $\{[10]\}$
$[12]D(I) \mapsto AW = 1$ $\{(CORN3)\}$
$[13]AW = 1 \mapsto y = 1$ $\{(CCEN2)\}$
$[14]D(I) \wedge$ *stable*$(N) \rightarrow y = 1$ $\{[12], [13], (R1), (STR2)\}$
$[15]J \models finy = 1$ $\{[2], [14], (RJ2)\}$
$[16]J \models finF_j^1 = 1$ $\{[6], (RJ2)\}$
$[17]J \models fin(y = 1 \wedge F_j^1 = 1)$ $\{[15], [16], (RJ4)\}$

$[18]y = 1 \wedge F_j^1 = 1 \mapsto f_j^1 = 1$ $\{(COUTN4)\}$
$[19]J \models finf_j^1 = 1$ $\{[2], [18], (STR1)\}$
$[20]J \models fin(F_j^0 = f_j^0 \wedge F_j^1 = f_j^1)$ $\{[6], [11], [19], \}$.

Assertion [20] holds for both cases considered above (see remarks following assertion [5]).

$[21]D(I) \wedge$ *stable*$(N) \rightarrow \hat{F}_j$
$= f_j(\hat{x}_1, \cdots, \hat{x}_n)$ $\{(CDRN2)\}$
$[22]J \models finD(O)$ $\{[5], [20]\}$
$[23]J \models fin[\forall j \in \{1, \cdots, m\} :$
$\hat{f}_j = f_j(\hat{x}_1, \cdots, \hat{x}_n)]$ $\{[2], [20], [21]\}$.

In view of the assumptions [1], [2], [3] (i.e., J is a *defining interval*) and $(RJ3)$ applied to [22], [23], Lemma 2 is proven. □

Lemma 3: The implementation network N of Section V satisfies condition $(DR4)$.

The formal proof of this lemma is based on the following sequence of arguments:

Let J be an undefining interval. Then at the beginning of J all inputs and all outputs are defined, the network is stable and $y = 1$. The value of y remains 1 as long as one of the inputs is still defined. Assume now that $f_j^0 = 0$ and $f_j^1 = 1$. It follows that f_j^1 equals 1 as long as one of the inputs is still defined. In view of the monotonicity of DRN, F_j^0 equals 0 during the whole interval, thus f_j^0 also retains the value 0. It follows that f_j remains defined as long as not all inputs become undefined.

Proof: Let J be an *undefining interval* (with respect to N). Thus,

$$
\begin{array}{ll}
[1]J \models [D(I) \wedge D(O) & \\
\quad \wedge\ \mathit{stable}(N)] & \{D \downarrow 1\} \\
[2]\forall x \in I : J \models (x \approx 0 \vee \downarrow x) & \{D \downarrow 2\} \\
[3]D(I) \mapsto AW = 1 & \{(CORN3)\} \\
[4]AW = 1 \mapsto y = 1 & \{(CCEN3)\} \\
[5]D(I) \mapsto y = 1 & \{[3],[4],(R1)\} \\
[6]D(I) \wedge\ \mathit{stable}(N) \rightarrow & \\
\quad AW = 1 \wedge y = 1 & \{[3],[5],(STR2)\} \\
[7]J \models (AW = 1 \wedge y = 1) & \{[1],[6],(RJ1)\} \\
[8]AW = 1 \wedge y = 1 \rightarrow & \\
\quad (\neg AW = 0 \trianglelefteq y = 1) & \{(CCEN4)\} \\
[9]\neg AI = 0 \trianglelefteq \neg AW = 0 & \{(CORN4)\} \\
[10]J \models (\neg AI = 0 \trianglelefteq y = 1) & \{[7],[8],[9],(R3),(RJ1)\}
\end{array}
$$

Let $j \in \{1, \cdots, m\}$

$$[11]J \models (f_j^0 = 0 \wedge f_j^1 = 1) \vee (f_j^0 = 1 \wedge f_j^1 = 0) \quad \{[1]\}.$$

We assume the case

$$[12]J \models (f_j^0 = 0 \wedge f_j^1 = 1) \quad \{[11],\ \mathit{Case\ 1}\}$$

The other case can be treated similarly.

$$
\begin{array}{ll}
[13]y = 1 \wedge f_j^1 = 1 \rightarrow & \\
\quad y = 1 \trianglelefteq f_j^1 = 1 & \{(COUTN5)\} \\
[14]J \models (\neg AI = 0 \trianglelefteq f_j^1 = 1) & \{[10],[12],[13],(R3), \\
 & \quad (RJ1),(RJ3)\} \\
[15]\ \mathit{stable}(N) \wedge y = 1 & \\
\quad \wedge f_j^0 = 0 \rightarrow F_j^0 = 0 & \{(COUTN4),(STR1)\} \\
[16]J \models F_j^0 = 0 & \{[1],[7],[12],[15], \\
 & \quad (RJ1),(RJ3)\} \\
[17]J \models F_j^0 \approx 0 & \{[1],[2],[16],(CDRN4)\} \\
[18]J \models f_j^0 \approx 0 & \{[12],[17],(COUTN3)\} \\
[19]J \models (\neg AI = 0 \trianglelefteq f_j^0 & \\
\quad = 0 \wedge f_j^1 = 1) & \{[14],[18],(RJ3)\} \\
[20]J \models (\neg AI = 0 \trianglelefteq f_j^0 \neq f_j^1) & \{[19]\}.
\end{array}
$$

One easily verifies that assertion [20] also holds for the other case mentioned above. Hence,

$$[21]J \models [\neg AI = 0 \trianglelefteq D(O)].$$

Thus, Lemma 3 and the above theorem are proven. □

VIII. Discussion

The terms *self-timed, delay-insensitive*, and *speed independent* systems are used to mean different things by different authors. We distinguish the following three attributes:

1) Independence of delays along wires and inside elements.
2) Generation of a completion signal.
3) Operation without requiring external timing signals (such as synchronizing clocks).

Delay insensitive systems actually guarantee only the first attribute, and often include the third as well. The proposed CL provide all three attributes, and we employ the adjective self-timed to mean that. Extensive research on the synthesis of delay-insensitive circuits is also reported in [17], [20], [8], [11], and [2].

As was shown by means of the example in Section VI, our approach carries the potential of generating smaller circuits than other methods. There are two main reasons for this advantage:

1) We do not generate all minterms (unlike Seitz and Anantharaman). Rather, we employ a minimized implementation of each function and its dual.
2) We do not combine small self-timed modules to achieve a large self-timed module (unlike Singh). A self-timed circuit incurs a certain amount of overhead, as compared to non-self-timed circuits. Combining small self-timed combinational circuits in order to construct a large one, such as proposed, for instance, in [18], duplicates the overhead and results in an inflated amount of circuitry.

We adopt a different approach for circuit construction: Combine the multiple Boolean equations into a single set, and implement that set directly. Thus, we manage to incur the self-timed overhead only once. In addition, separate CL modules may be freely interconnected to create larger combinational logic blocks as long as no loops are generated. In each case, the correct operation of the overall network is ensured, provided that the environment adheres to the given behavioral constraints.

Our synthesis algorithm does not specify what type of Boolean minimization should be carried on the DRN subnet. This subnet may consist of two-level (AND–OR) logic, or of multilevel logic. The latter form enables reasonable application of advanced minimization algorithms [4] which generate multilevel logic for very large combinational circuits, which are impractical to minimize with exponential algorithms.

The DRN subnet can be implemented in a number of different manners. Indeed, all that matters is that the DRN subnet is monotonic, as expressed by condition CDRN1, and behaves "rationally," as expressed by the other three conditions. The simplest materialization of those conditions is a combinational network consisting exclusively of AND and OR gates, but other possibilities may also exist.

An important aspect of our method is that it strives toward complete delay-insensitivity. Other proposals (e.g., [18]) have suggested various shortcuts in order to simplify self-timed logic. Most importantly, an assumption of "equipotential" regions is often resorted to. Within an equipotential region it is assumed that the delays are well understood and that the circuit functions correctly, without providing the additional circuitry to guarantee self-timed operation.

The equipotential regions approach carries with it a twofold shortcoming. First, the correct operation of the circuit depends on loosely defined, intuitive concepts of approximation, which often lead to missed estimates and design errors. Second, this methodology does not scale very well: As the implementation technology improves in speed and density, the equipotential regions reduce in size at a faster rate than the linear contraction of the circuit. Thus, a subcircuit which is designed within an equipotential region at some point in time may not scale with the advance of technology [7]. In contrast, our proposed method relies exclusively on double-rail implementation. It is completely delay-insensitive under the assumption that *fundamental* mode coincides with the *input–output* mode [2]; in the fundamental mode, inputs to a circuit are allowed to change only after both outputs and all internal nodes have stabilized, whereas in the input–output mode inputs may change as soon as the outputs have stabilized. As shown in Section VII, our proof assumes the former mode. Note that it has been pointed out that completely delay-insensitive designs are theoretically impossible, if the *input–output* mode is adhered to [3], [12]. Note further that the *fundamental* mode assumption renders the isochronic fork assumption (as referred to in Section V) redundant.

We have developed the proposed method as part of our effort to construct a silicon compiler which generates self-timed logic. The implementation method presented in this paper is straightforward, and is readily embedded it in a computer program which can generate self-timed combinational circuits automatically out of Boolean equations. Having proved the correct self-timed operation of the resultant circuit, our method now generates combinational self-timed circuits which are mathematically-proven "correct-by-construction," a prerequisite to employing them in a silicon compiler.

It has been claimed that developing self-timed combinational logic is an unnecessary step, and that self-timed methodology is only useful for sequential (asynchronous) machines and for structures of higher complexity. However, we pursue this effort for two reasons. First, self-timed combinational circuits, as described in this paper, constitute an important building block in our synthesis methodologies for finite state machines and for more complex structures, which are currently being developed. Second, we believe that with the advent of extremely fast technologies, such as GaAs, where many known clocking and timing schemes fail, self-timed techniques may turn out to be mandatory even at the smallest combinational circuit level. In addition, this method is extended to self-timed FSM's as described in [6].

IX. Conclusions

In this paper we have presented a method by which any finite set of Boolean functions can be implemented efficiently as a self-timed combinational circuit. We have seen that if the initial conditions of the circuit are zeroed and stable, then the circuit satisfies the conditions we formulated, and thus behaves as a self-timed system.

Other methods for implementing self-timed logic use an excessive number of gates, whereas our method yields an efficient circuit.

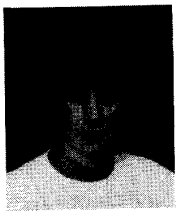
We have rephrased the informal specification of [18] and employed temporal logic to express it in a formal manner. We have provided a formal proof of the correctness of any self-timed circuit designed by our method. The simplicity of the synthesis algorithm, and the fact that the resultant circuits are mathematically proven to be "correct by construction," make this method appropriate for automatic synthesis of self-timed combinational circuits within a silicon compiler. The synthesis of self-timed FSM's is discussed in [6].

Acknowledgment

The authors wish to express their appreciation to the referees for their detailed and helpful comments that improved the quality of this paper.

References

[1] T. S. Anantharaman, "A delay insensitive regular expression recognizer," *IEEE VLSI Tech. Bull.*, Sept. 1986.
[2] J. A. Brzozowski and J. C. Ebergen, "Recent developments in the design of asynchronous circuits," in *Proc. Seventh Int. Conf. Fundamental Computat. Theory FCT'89*, Hungary, Aug. 89.
[3] ——, private communication, 1990.
[4] R. K. Brayton *et al.*, *Logic Minimization Algorithm for VLSI Synthesis*. Norwell, MA: Kluwer Academic, 1984.
[5] J. T. Butler, Guest Editor, Special Issue on Multiple Valued Logic, *IEEE Comput. Mag.*, Apr. 1988.
[6] I. David, R. Ginosar, and M. Yoeli, "Implementing sequential machines as self-timed circuits," *IEEE Trans. Comput.*, this issue, pp. 12–17.
[7] A. L. Davis, private communication, 1988.
[8] J. C. Ebergen, "Translating programs into delay-insensitive circuits," Ph.D. dissertation, Eindhoven Univ. of Technology, 1987.
[9] J. Halpern, Z. Manna, and B. Moszkowski, "A hardware semantics based on temporal intervals," in *Proc. 10th Int. Colloq. Automata, Languages and Programming*, Barcelona, Spain. Berlin, Germany: Springer-Verlag, 1983, pp. 278–291.
[10] L. Lamport, "What good is temporal logic?," in *Proc. Inform. Processing 83*, R. E. A. Mason, Ed. Amsterdam, The Netherlands: North-Holland, pp. 657–668.
[11] A. J. Martin, "Compiling communicating processes into delay insensitive VLSI circuits," *Distributed Comput.*, vol. 1, no. 3, 1986.
[12] ——, "Limitations to delay-insensitivity in asynchronous circuits," Tech. Rep. CS-TR-90-02, Dep. Comput. Sci., California Institute of Technology, 1990.
[13] Y. Malachi and S. Owicki, "Temporal specifications of self-timed systems," in *VLSI Systems and Computations*, H. T. Kung B. Sproul, and G. Steel, Eds. Rockville, MD: Computer Science Press, 1981, pp. 203–212.
[14] R. E. Miller, *Switching Theory, Vol. 2*. New York: Wiley, 1965.
[15] B. Moszkowski, "A temporal logic for multilevel reasoning about hardware," *IEEE Comput. Mag.*, pp. 10–19, Feb. 1985.
[16] Z. Manna and A. Pnueli " Verification of concurrent programs: The temporal framework," in *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, Eds., International Lecture Series in Computer Science. New York: Academic, 1981, pp. 215–273.
[17] M. Rem, "Concurrent computations and VLSI circuits," in *Control Flow and Data Flow; Concepts of Distributed Computing*, M. Broy Ed. Berlin, Germany: Springer-Verlag, 1985, pp. 399–437.
[18] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Reading, MA: Addison-Wesley, 1980, pp. 218–262.
[19] N. P. Singh, "A design methodology for self-timed systems," M.Sc. Thesis, MIT Laboratory for Computer Science Tech. Rep. TR-258, MIT, Cambridge, MA, Feb. 1981.
[20] J. L. A. Van de Snepscheut, *Trace Theory and VLSI Design*, LNCS 200, 1985.

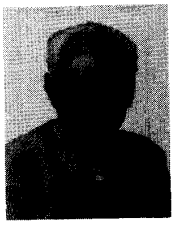


Ilana David received the B.Sc., M.Sc. and D.Sc. degrees in electrical engineering from the Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa, in 1972, 1975, and 1991, respectively.

She is currently a Software Engineer at the Department of Electrical Engineering, Technion. Her research interests are in the area of logic design, computer architectures, and asynchronous systems.

Ran Ginosar (S'79–M'82) received the B.Sc. degree (electrical engineering and computer science) (summa cum laude) from the Technion–Israel Institute of Technology, Haifa, in 1978 and the M.A. and Ph.D. degrees from Princeton University, Princeton, NJ, in 1979 and 1982, respectively.

He has served as a Member of the Technical Staff at Bell Laboratories Research, Murray Hill, NJ, until 1983, and has since been with the faculty of Electrical Engineering and Computer Science at the Technion, and (as a visitor) the Department of Computer Science at the University of Utah. His research interests focus on high-performance, highly parallel VLSI architectures and self-timed design and architectures.



Michael Yoeli received the M.Sc. degree in mathematics from the Hebrew University, Jerusalem, Israel, in 1957, and the D.Sc. degree in mathematics from the Technion–Israel Institute of Technology, Haifa, in 1960.

He is currently Professor Emeritus at the Department of Computer Science, Technion. He joined the faculty of the Department of Electrical Engineering at the Technion in 1955, where he became Professor in 1968. In 1969 he was one of the founding fathers of the Department of Computer Science at the Technion, and served as its Chairman during 1973–1975. In 1982 he was awarded the Bank of Leumi Chair in Computer Science. He published many papers in the following areas: theory of automata, multivalued switching, cellular logic, theory and applications of Peti nets, switch-level modeling of CMOS circuits, and verification and synthesis of delay-insensitive networks. He is also a co-author of *Digital Networks* (Englewood Cliff, NJ: Prentice-Hall, 1976) and editor of *Formal Verification of Hardware Design* (Los Alamitos, CA: IEEE Computer Society Press).