Vol. 45 No. 5 IPSJ Journal May 2004

Regular Paper

Logic Optimization of Asynchronous Speed-Independent Circuits Using Transduction Methods

Hiroshi Saito,† Hiroshi Nakamura,† Masahiro Fujita†† and Takashi Nanya†

In this paper, we present an optimization method of asynchronous speed-independent circuits based on transduction methods. Although transduction methods are well used for the optimization of multi-level combinational circuits, the direct application to speed-independent circuits may leads to a malfunction of circuits because the property of hazard-freeness guaranteed by circuits is broken. Therefore, we discuss how to optimize speed-independent circuits by using transduction methods without leading any hazardous behavior. For instance, we extend the gate substitution algorithm in transduction methods. Finally, we evaluate the efficiency of the extended gate substitution algorithm by applying it to several benchmark circuits.

1. Introduction

Signal transition graphs (STGs)[2] are well used to specify the behavior of asynchronous circuits. Logic synthesis tool `Petrify`[3] accepts an STG and produces a hazard-free asynchronous speed-independent (SI) circuit. The SI circuit can operate correctly under arbitrary gate delay and zero wire delay[6].

Although `Petrify` is well established for the synthesis of SI circuits, global optimization using the relationships among the logic functions of gates is restricted to gate substitution after logic decomposition[5]. Namely, if a function is decomposed, `Petrify` exploits whether the gate corresponding to the decomposed function can substitute for other gates. However, it implies that there is no chance of optimization if no function is decomposed because of the appearances of hazardous behavior. Because the logic function of each non-input (output or internal) signal is derived separately from the binary encoded state graph of a given STG, the failure of decomposition remains redundancy in the synthesized circuit such that the identical function appears on the logic network of different signals.

Thus, in this paper, we present an alternative approach to optimize SI circuits globally based on transduction methods[4),8]. Transduction methods are well used for the optimization of multi-level combinational circuits. By calculating *permissible functions* which represent don't care space in a given circuit, transduction methods optimize circuits by merging common gates, substituting gates, or pruning gates so that the total number of gates and/or connections are minimized.

In this paper, we extend *the gate substitution algorithm*[8] to optimize SI circuits based on *the standard-C architecture*[5]. Because the gate substitution algorithm does not concern the property of hazard-freeness guaranteed in SI circuits, there is a possibility that the property is broken during transformation. Therefore, to optimize SI circuits correctly, the gate substitution algorithm is extended in the following two aspects. First, we extend the traditional calculation method of permissible functions. The key point is that the assignment of don't cares is carried out based on *monotonous cover conditions*[5] which guarantee hazard-freeness in SI circuits based on the standard-C architecture. Second, we introduce valid conditions of substitution to preserve the property of hazard-freeness. Although these conditions are also based on monotonous cover conditions, we modify them so that the evaluation of monotonous cover conditions is carried out on calculated logic functions and permissible functions.

Figure 1 shows how our proposed extended gate substitution algorithm works. At first, the original STG is synthesized to generate a SI circuit based on the standard-C architecture and the new STG corresponding to the synthesized circuit. The new STG is different from the original one in that it includes the behavior of not only input/output signals

† Research Center for Advanced Science and Technology, The University of Tokyo
†† Department of Electrical Engineering, The University of Tokyo

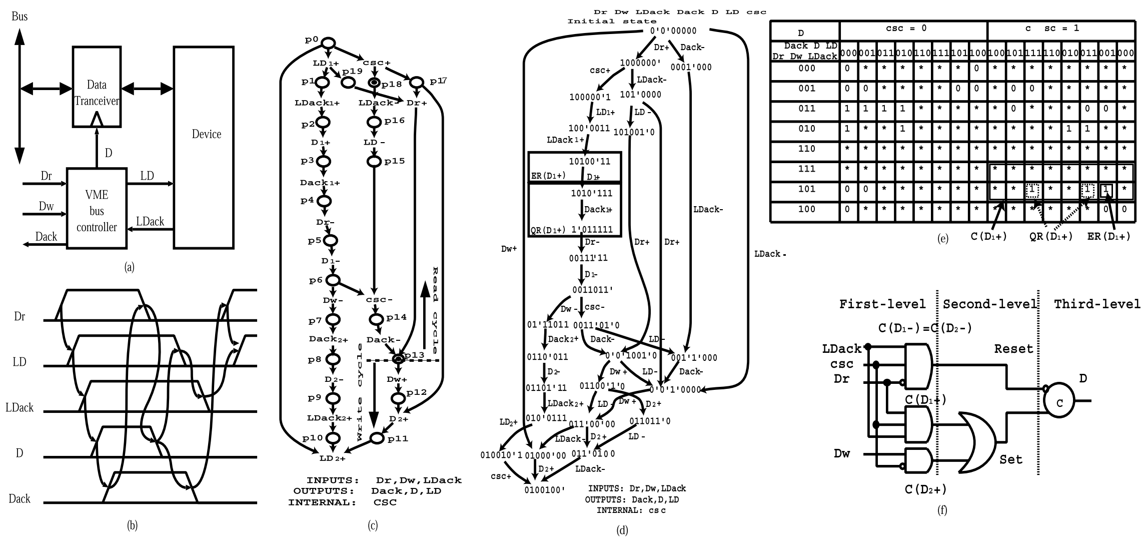


Fig. 2 VME bus controller: (a) Interface, (b) Timing diagram of a read cycle, (c) STG, (d) SG, (e) Karnaugh map for signal *D*, (f) Standard-C architecture for signal *D*.

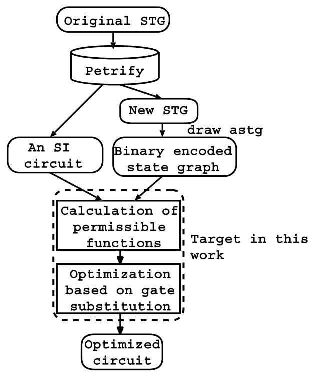

Original STG
Petrify
New STG
draw astg
An SI circuit
Binary encoded state graph
Calculation of permissible functions
Optimization based on gate substitution
Target in this work
Optimized circuit

Fig. 1 Optimization flow.

but also internal signals inserted by Petrify. Then, by using draw_astg command distributed in Petrify package, the binary encoded state graph is translated from the new STG. After both the SI circuit and the state graph are generated as inputs of the extended gate substitution algorithm, permissible functions are calculated throughout the circuit. Finally, the circuit is optimized without violating the property of hazard-freeness.

The rest of this paper is organized as follows. The basic notions for SI circuits are shown in Section 2. In Section 3, we describe a traditional calculation method of permissible functions at first, and then we describe how to calculate permissible functions in SI circuits. In Section 4, we describe conditions of valid substitution and the extended gate substitution algorithm. Finally, we show experimental results in Section 5 and conclude this work in Section 6.

2. Basic Notions of Asynchronous SI Circuits

2.1 Signal Transition Graph

Figure 2 (a) shows the interface of a VME bus controller. The controller controls the flow of data between Bus and Device via a Data Transceiver. There are two input signals *Dr* and *Dw* for bus side which indicate the beginning of a bus read and write cycle. After the completion of cycles, these signals are acknowledged by output signal *Dack*. On the other hand, there is a pair of handshake signals *LD* and *LDack* for device side. Data Transceiver is controlled by signal *D*. The timing diagram of a read cycle is shown in Fig. 2 (b).

Signal transition graphs (STGs)[2] known as interpreted Petri Nets[7] are well used to specify the behavior of asynchronous circuits. The STG shown in Fig. 2 (c) represents the behavior of the VME bus controller, which is generated by Petrify[3] after the logic circuit is synthesized. The STG has one internal signal *csc* which is inserted during the synthesis. Note all signals in an STG are partitioned into the set of input signals and the set of non-input (internal and output) signals.

Rising transitions of signal a are labeled with "$a+$" and falling transitions with "$a-$". We also use the notation $a*$ if the sign of the transition is not specific. Multiple transitions for a signal are distinguished by an index i (e.g., D_1+ and D_2+ in Fig. 2 (c)). Places (denoted by circles) can be marked with tokens (black dots). The set of all places currently marked is called a *marking*. On the other hand, all arcs in STGs represent the ordering relation between vertices.

A signal transition is *enabled* if all its input places contain a token. After the firing of an enabled transition, one token is removed from every input place and added to every output place of the transition. For example, in the initial marking $\{p13, p18\}$ of the STG in Fig. 2 (c), transitions of input signals, $Dw+$ and $Dr+$, are enabled. In that marking, however, only one of them can be fired because there is only one token in place $p13$ (i.e., the removal of the token by the firing of one transition will disable the other). The choice is decided by the environment. If transition $Dw+$ is fired, a new marking, $\{p12, p18\}$, where D_2+ becomes enabled is achieved. In that case, transition $Dr+$ is not enabled any more (i.e., transition $Dr+$ is *disabled*).

2.2 State Graph

By analyzing all of the reachable markings in a given STG, one can generate the *state graph* (SG) (i.e., each marking in the STG corresponds to a state in the SG). In SGs, vertices labeled with a vector of signal values represent binary encoded states and arcs between pairs of vertices labeled with a signal transition represent the ordering relation between states. Figure 2 (d) is the SG corresponding to the STG in Fig. 2 (c). In state 0'0'00000 of the SG where all of the signals have value 0, transitions $Dr+$ and $Dw+$ are enabled. After the firing of $Dw+$, the corresponding circuit achieves to state 01000'00 where $D+$ is enabled. Note that if a transition is enabled in a state, the corresponding signal value of the state is labeled by symbol '.

2.2.1 Speed-Independence Property

A synthesized asynchronous SI circuit is hazard-free if the SG satisfies the speed-independence property[6]. The property consists of the following two conditions:

- no non-input signal transition can be disabled by another signal transition
- no input signal transition can be disabled by a non-input signal transition

The former ensures that there are no glitches, known as hazards, at the output of gates, while the latter ensures that there are no hazards at the inputs of circuits. According to the speed-independence property, only transitions for input signals can be disabled by transitions for different input signals (i.e., a choice by environment as in $Dr+$ and $Dw+$ of Fig. 2 (c)).

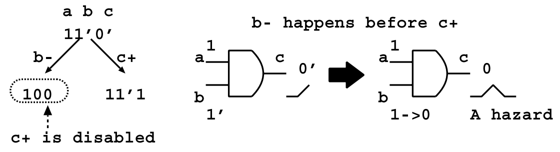


Fig. 3 Violation of speed-independence.

Figure 3 shows a violation of the speed-independence property. Initially, in state 11'0', the rising transition of non-input signal c, $c+$, and the falling transition of input signal b, $b-$, are enabled. If $b-$ happens before $c+$, the enabled transition $c+$ is disabled in state 100 (i.e., $c+$ cannot be fired any more). This implies a hazard at the output of the AND gate.

Note an SG must satisfy not only the speed-independence property but also other properties to be implementable as a hazard-free SI circuit. These are described in the literature[6].

2.2.2 Next-State Function

The next-state function on a non-input signal is defined as follows[5]. It maps the binary code of each SG state s into:

- 1 if the signal has value 0' or 1 in the binary code of s
- 0 if the signal has value 1' or 0 in the binary code of s
- * (don't care) for all binary codes that do not correspond to any reachable SG state

The next-state function of signal D in the SG of Fig. 2 (d) is represented in Fig. 2 (e) with the Karnaugh map.

2.2.3 Excitation Region and Quiescent Region

Given a signal a, we can classify the states of the SG into the following sets[5]:

- Positive and negative excitation regions (ERs)
- Positive and negative quiescent regions (QRs)

A maximally connected set of states in which a signal transition a_i* is enabled is called an ER for the transition a_i* (denoted by ER(a_i*)). The QR of a signal transition a_i* (denoted by QR(a_i*)), with the corresponding ER(a_i*), is a maximal set of states s reachable from ER(a_i*) such that $a*$ is stable (not enabled) in s and s is not reachable from any other ER(a_j*) ($i \neq j$) without going through ER(a_i*). Examples of

ER and QR for transition D_1+ are shown in Fig. 2 (d).

2.3 Standard-C architecture

In this work, we focus on the optimization of asynchronous SI circuits based on the standard-C architecture. In the standard-C architecture, the logic circuit of a non-input signal consists of three-level logics as shown in Fig. 2 (f). The first-level AND-OR gates called *monotonous covers* (MCs) correspond to a signal transition in a given STG. An AND-OR gate represents not only a complex gate but also an AND or OR gate with multiple input inverters. They are configured by covering SG states based on the following MC conditions. The second-level OR gates gather MCs for all rising or falling transitions of a signal to organize a set or reset function of the signal. The third-level C-element corresponds to the output of a non-input signal. The C-element is equal to the function, $a = a \cdot (b + c) + b \cdot c$, and behaves as follows. The output signal a is equal to 1 if both inputs b and c are equal to 1 and 0 if both b and c are equal to 0. Otherwise, it keeps the previous value of signal a.

The definition of MC conditions are as follows[5].

Definition 2.1 $C(a_i*)$ denotes one of the first-level AND-OR gates in the standard-C architecture. $C(a_i*)$ is a correct monotonic polyterm cover for the $\mathsf{ER}(a_i*)$ if the following three conditions are satisfied:

(1) Cover condition: $C(a_i*)$ covers all states of $\mathsf{ER}(a_i*)$, i.e., $C(a_i*)$ is logic 1 in all states of $\mathsf{ER}(a_i*)$

(2) One-hot condition: $C(a_i*)$ does not cover any state outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$, i.e., $C(a_i*)$ is logic 0 in all states outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$

(3) Monotonicity condition: $C(a_i*)$ can change the value from 1 to 0 at most once along any state sequence within $\mathsf{QR}(a_i*)$

In the standard-C architecture, the speed-independence property must be satisfied not only for the transitions of input and non-input signals but also for the transitions of all MCs. The MC conditions satisfy this requirement[5].

Let us explain the reason briefly. Suppose that the synthesized circuit enters a state in $\mathsf{ER}(a_i+)$. The cover condition ensures that $C(a_i+)$ changes the logic value from 0 to 1 in that state (i.e., the rising transition of $C(a_i+)$ is not disabled). The one-hot condition guarantees that no other gates $C(a_j+)(i \neq j)$ in the set function of signal a and no gates $C(a_i-)$ in the reset function of signal a can be at 1 in that moment. Therefore, $C(a_i+)$ is the only gate to change the value of signal a to 1 through the OR gate and the C-element (i.e., the enabled transition a_i+ is not disabled). When signal a changes its value from 0 to 1, the circuit enters a state in $\mathsf{QR}(a_i+)$. According to the last condition, $C(a_i+)$ will be reset the value from 1 to 0 (i.e., the falling transition of $C(a_i+)$ is not disabled) and this change is propagated to signal a when it will go low (i.e., a_i-). As a result, we can find out that there is no disabled transition in the circuit (i.e., the speed-independence property is satisfied).

The generation of MCs is explained by considering the case of transition D_1+ in Fig. 2 (d). $\mathsf{ER}(D_1+)$ consists of state 10100'00 and $\mathsf{QR}(D_1+)$ consists of states 1010'111 and 1'011111. First, cube $Dr \cdot \overline{Dw} \cdot LDack \cdot \overline{Dack} \cdot \overline{D} \cdot LD \cdot csc$ is generated to cover all the states of $\mathsf{ER}(D_1+)$. Then, the cube is expanded to $Dr \cdot LDack \cdot csc$ without violating the one-hot and monotonicity conditions. This cube corresponds to the MC for transition D_1+, respectively (see Fig. 2 (e) and Fig. 2 (f)). Note the cube also covers unreachable states denoted as don't cares.

3. Calculation Methods of Permissible Functions

3.1 A Traditional Calculation Method of Permissible Functions

Before to explain a traditional calculation method of permissible functions, we define permissible functions as follows[8]:

Definition 3.1 If no non-input signals change after replacing the function realized at a gate or net to a function f, the function f is called a permissible function for the gate or net.

Here, a net represents the wire for a signal or a fanout of a gate (a net is denoted as $n_i(i = 1, 2, \ldots)$ in **Fig. 4** (a)).

Permissible functions are calculated for each net and gate based on the following two steps[8]:

(1) The logic function of each gate and net is calculated from the nets of input signals by assigning truth values.

(2) The permissible functions of each gate and net are calculated from the nets of non-input signals by assigning don't cares.

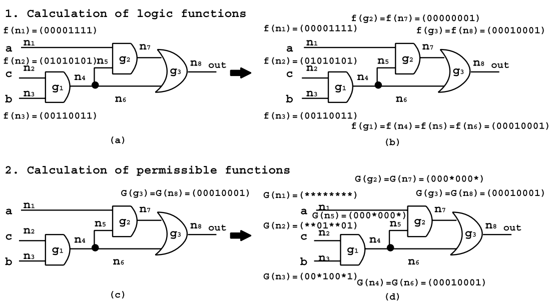


Fig. 4 Calculation of permissible functions.

By assigning truth values to the nets of input signals, the logic function of each gate and net is calculated from the nets of input signals to the nets of non-input signals. 2^n truth values are assigned for each net of input signals if there are n input signals. For example in Fig. 4 (a), because there are three input signals, eight truth values are assigned to each net of input signals. A vector of these values is referred as the logic function of a gate or net. For example, the logic function of the net of input signal b in Fig. 4 (a) is represented as $f(n_3) = (00110011)$.

After logic functions are calculated throughout a given circuit as in Fig. 4 (a), the permissible functions of each gate and net are calculated from the nets of non-input signals to the nets of input signals. Initially, the calculated logic functions of the nets of non-input signals are assigned as the permissible functions of the nets of non-input signals together with external don't cares (Fig. 4 (c)). Then, the permissible functions for all internal gates and nets are calculated by assigning don't cares. In the case of an OR gate, all of the inputs must be 0 if the output is equal to 0. However, if the output is equal to 1, one of its inputs must be 1, while the others can be either 0 or 1 (i.e., don't care, denoted by $*$). Based on the same consideration, we can calculate the permissible functions of other types of gates. Note if a gate or net has multiple fanout, the permissible functions of the gate or net are represented as the common set of the permissible functions for each fanout.

Figure 4 (d) shows the calculated permissible functions for the circuit in Fig. 4 (a). As same as a logic function, the permissible functions for a gate or net are denoted as a vector of truth values with don't cares. For example, $G(n_3) =$ (00*100*1) represents a set of permissible functions of net n_3 (i.e., (00110011), (00110001), (00010001), and (00010011)). The permissible functions of gate g_1 and net n_4, $G(g_1) = G(n_4)$ = (00010001), are the common set of $G(n_5) =$ (000*000*) and $G(n_6) =$ (00010001) because gate g_1 and net n_4 has two fanout (i.e., nets n_5 and n_6).

During the calculation of permissible functions, a gate or net may have several sets of permissible functions. For example in Fig. 4 (d), (****000*) is another set of permissible functions for net n_5 (the first three elements are different from the previous one). The difference comes from the selection of don't care assignment to n_1 and n_5 where the output of gate g_2 is equal to 0. However, since to calculate all of the sets requires a lot of computation time, we select only one set based on priority. A set of permissible functions selected by priority is called *Compatible Set of Permissible Functions* (CSPF)[8].

For the selection of don't care assignment, we simply follow the same manner described in the paper[8]. According to the paper, priority is assigned to each input of gates. After the assignment of priority, don't cares are assigned to the inputs which have higher priority, while don't care is not assigned to the input which has the lowest priority. Priority is assigned as follows:

- If an input is a fanout of a gate and the gate is connected to the net of a non-input signal, the lowest priority is assigned to the input because we cannot remove the gate connected to the non-input signal.
- If an input is a fanout of a gate which has multiple fanout, lower priority is assigned to the input because the reduction of the gate affects many parts of the circuit.

For example, we assign priority to the inputs of gate g_2 (i.e., nets n_1 and n_5) in Fig. 4 (d). Because n_5 is a fanout of gate g_1 which has multiple fanout (n_5 and n_6), we assign higher priority not to n_5 but to n_1 (i.e., $G(n_1)$ has many don't cares in the elements).

3.2 A Calculation Method in Asynchronous SI Circuits

To calculate permissible functions (i.e., CSPF) in asynchronous SI circuits, we extend the calculation method described in Section 3.1. The reasons are as follows:

- The target of the calculation method described in Section 3.1 is combinational circuits. However, SI circuits derived by `Petrify` represent sequential machines which have feedback loops.
- The calculation method described in Sec-

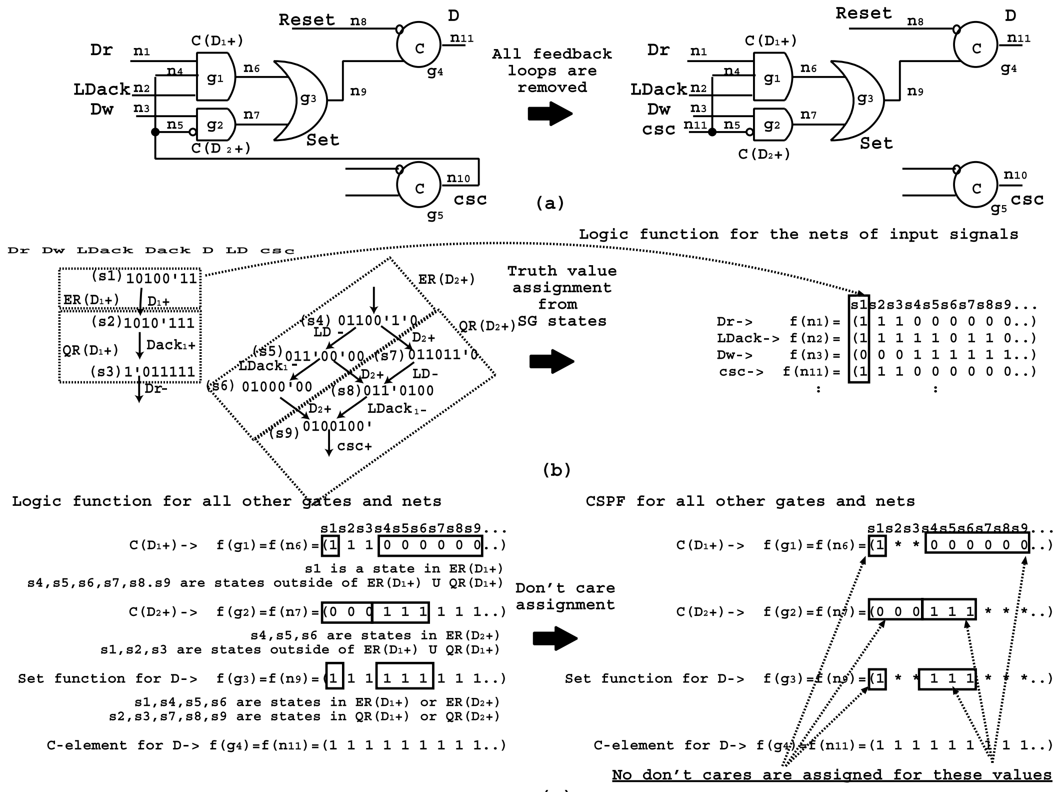

Reset n8
D
n11
C
g4
Dr n1
C(D1+)
n4
g1
n6
LDack n2
Dw n3
g3
n9
n5
g2
n7
Set
C(D2+)
n10
csc
g5
All feedback loops are removed
csc n11
(a)
Logic function for the nets of input signals
Dr Dw LDack Dack D LD csc
(s1) 10100'11
ER(D1+) D1+
(s2)1010'111
QR(D1+) Dack1+
(s3)1'011111
Dr-
ER(D2+)
(s4) 01100'1'0
LD-
D2+
QR(D2+)
(s5) 011'00'00
(s7) 011011'0
LDack1-
D2+
LD-
(s6) 01000'00
(s8) 011'0100
D2+
LDack1-
(s9) 0100100'
csc+
Truth value assignment from SG states
s1s2s3s4s5s6s7s8s9...
Dr-> f(n1)=(1 1 1 0 0 0 0 0 0..)
LDack-> f(n2)=(1 1 1 1 1 0 1 1 0..)
Dw-> f(n3)=(0 0 0 1 1 1 1 1 1..)
csc-> f(n11)=(1 1 1 0 0 0 0 0 0..)
(b)
Logic function for all other gates and nets
s1s2s3s4s5s6s7s8s9...
C(D1+)-> f(g1)=f(n6)=(1 1 1 0 0 0 0 0 0..)
s1 is a state in ER(D1+)
s4,s5,s6,s7,s8.s9 are states outside of ER(D1+) U QR(D1+)
C(D2+)-> f(g2)=f(n7)=(0 0 0 1 1 1 1 1 1..)
s4,s5,s6 are states in ER(D2+)
s1,s2,s3 are states outside of ER(D1+) U QR(D1+)
Set function for D-> f(g3)=f(n9)=(1 1 1 1 1 1 1 1 1..)
s1,s4,s5,s6 are states in ER(D1+) or ER(D2+)
s2,s3,s7,s8,s9 are states in QR(D1+) or QR(D2+)
C-element for D-> f(g4)=f(n11)=(1 1 1 1 1 1 1 1 1..)
Don't care assignment
CSPF for all other gates and nets
s1s2s3s4s5s6s7s8s9...
C(D1+)-> f(g1)=f(n6)=(1 * * 0 0 0 0 0 0..)
C(D2+)-> f(g2)=f(n7)=(0 0 0 1 1 1 * * *..)
Set function for D-> f(g3)=f(n9)=(1 * * 1 1 1 * * *..)
C-element for D-> f(g4)=f(n11)=(1 1 1 1 1 1 1 1 1..)
No don't cares are assigned for these values
(c)

Fig. 5 Calculation of permissible functions in asynchronous SI circuits.

tion 3.1 does not concern the possibility of hazards when we assign don't cares. However, since SI circuits preserve the property of hazard-freeness (i.e., it is guaranteed by Def. 2.1 in the standard-C architecture), we must carefully assign don't cares to reduce the possibility of hazardous behavior.

For the extension, we introduce the following three requirements:

(1) Removal of all feedback loops
(2) Assignment of truth values from SG states
(3) Assignment of don't cares based on Def. 2.1

Because SI circuits represent sequential machines, they contain feedback loops. To prevent iterative calculation of logic functions and permissible functions caused by these loops, we must cut all of the loops before the calculation of logic functions (see **Fig. 5** (a)). After the cut of feedback loops, the calculation of logic functions and CSPF in SI circuits is considered as the cases of multi-level combinational circuits.

Since SI circuits are synthesized from an SG, truth values for each net of input signals are directly assigned from the values of SG states. As a result, we can save memory space and computation time because of the avoidance of calculation in unreachable states. For example in Fig. 5 (b), since *Dr* is 1, *LDack* is 1, *Dw* is 0, and *csc* is 1 in state $s1$, these values are assigned to the nets of input signals such that the first value of $f(n_1)$ is 0, $f(n_2)$ is 1, $f(n_3)$ is 0, and $f(n_{11})$ is 1. After all state values are assigned to the nets of input signals, the logic functions of gates and nets are calculated as described in Section 3.1. Note each value in the logic function of non-input signals represents the next state value with respect to the current state.

For the assignment of don't cares, we refer to MC conditions described in Def. 2.1. According to the cover condition and the one-hot condition in Def. 2.1, an MC $C(a_i*)$ must be logic 1 in all states of $\mathsf{ER}(a_i*)$ and logic 0 in all states outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$. Therefore, we should not assign don't cares to the values of the logic function in $C(a_i*)$ (i.e., the first-level AND-OR gate in the standard-C architecture) if the values are related to those states. In other words, don't cares are assigned only to the values related to the states of $\mathsf{QR}(a_i*)$.

In the case of the set function of signal a (i.e., the second-level OR gate), don't cares are assigned to the values related to the states of all $\mathsf{QR}(a_i+)$ (i.e, states in $\mathsf{QR}(a_1+) \cup \mathsf{QR}(a_2+) \cup \ldots \cup \mathsf{QR}(a_n+) (i = 1 \ldots n)$). On the other hand, in the case of the third-level C-element, no don't cares are assigned to all the values of the logic function.

For example in Fig. 5 (c), we have never assigned don't cares for all the values of the logic function of signal D (i.e., $f(g_4)$). For the set function of signal D (i.e., $f(g_3)$), don't cares are assigned to the values related to states $s2$, $s3$, $s7$, $s8$, and $s9$ because these states are in $\mathsf{QR}(D_1+) \cup \mathsf{QR}(D_2+)$. Finally, for MCs of D_1+ and D_2+ (i.e., $f(g_1)$ and $f(g_2)$), we assign don't cares to the values related to $\mathsf{QR}(D_1+)$ (for $f(g_1)$) and $\mathsf{QR}(D_2+)$ (for $f(g_2)$).

By introducing these requirements, we can calculate CSPF in SI circuits. Figure 5 (c) shows the calculated CSPFs with respect to the circuit in Fig. 5 (a).

4. Transduction Methods in Asynchronous SI Circuits

In transduction methods described in the literature [8)], we concentrate on the gate substitution algorithm. In the literature, the algorithm was applied to the optimization of multi-level combinational circuits. However, because the gate substitution algorithm does not concern the property of hazard-freeness guaranteed in SI circuits, there is a possibility that the property is broken after substitution if we apply it to SI circuits directly. Therefore, to apply it to SI circuits, we must extend the algorithm so that any substitution does not lead to hazardous behavior.

4.1 Conditions of Valid Gate Substitution

To preserve hazard-freeness after substitution, we propose the conditions described in Proposition 4.1. In the proposition, gate g_2 can substitute for gate g_1 only when gate g_2 satisfies the MC conditions (Def. 2.1) satisfied by gate g_1. This is because in SI circuits based on the standard-C architecture, the hazard-freeness of gates is guaranteed by the MC conditions. Therefore, if g_2 satisfies the MC conditions satisfied by g_1, the substitution does not lead to any hazardous behavior.

Different from the conditions in Def. 2.1 which are evaluated on SG, the conditions described in Prop. 4.1 are evaluated on calculated logic functions and CSPFs. Before describing the proposition, we introduce several terminologies for calculated logic functions and CSPFs:

- $f(g)$ - the logic function of gate g
- $f_s(g)$ - the value of $f(g)$ in state s
- $G(g)$ - the CSPF of gate g
- $G_s(g)$ - the value of $G(g)$ in state s
- $pred_s$ - the set of all direct predecessor states of state s

The substitution of gate g_2 for gate g_1 is validated by the following proposition. g_1 and g_2 correspond to the MCs (i.e., first-level AND-OR gates) for the i-th transition of $a*$ and the j-th transition of $b*$ (i.e., a_i* and b_j*).

Proposition 4.1 The substitution of gate g_2 for gate g_1 is hazard-free if the following conditions are satisfied.

(1) $\forall$ state $s : G_s(g_1) = 1 \Rightarrow f_s(g_2) = 1$

(2) $\forall$ state $s : G_s(g_1) = 0 \Rightarrow f_s(g_2) = 0$

(3) $\forall$ state $s : s \in \mathsf{QR}(a_i*) \wedge f_s(g_2) = 1 \Rightarrow f_{s'}(g_2) = 1$ in all states s' of $pred_s$

As the proof of Prop. 4.1, we show that g_2 satisfies MC conditions not only for transition b_j* but also for transition a_i*.

Proof: The first condition in Prop. 4.1 claims that g_2 satisfies the cover condition in Def. 2.1 for both transitions a_i* and b_j*. According to the calculation of CSPFs described in Section 3.2, no don't care is assigned to $f_s(g_1)$ if s is a state in $\mathsf{ER}(a_i*)$. Therefore, $G_s(g_1)$ is equal to 1 in state s. If $f_s(g_2)$ is equal to 1 in all states s where $G_s(g_1)$ is equal to 1, it means that g_2 covers all states in not only $\mathsf{ER}(b_j*)$ but also $\mathsf{ER}(a_i*)$. In these states, the rising transition of g_2 is fired (i.e., not disabled).

The second condition claims that g_2 satisfies the one-hot condition in Def. 2.1 for both transitions a_i* and b_j*. As same as the first condition, no don't care is assigned to $f_s(g_1)$ if s is a state outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$. Therefore, $G_s(g_1)$ is equal to 0 in state s. If $f_s(g_2)$ is equal to 0 in all states s where $G_s(g_1)$ is equal to 0, it means that g_2 does not cover not only the states outside of $\mathsf{ER}(b_j*) \cup \mathsf{QR}(b_j*)$ but also the states outside of $\mathsf{ER}(a_i*) \cup \mathsf{QR}(a_i*)$. g_2 is the only gate to change the value of signals a and b when a_i* and b_j* are enabled.

The third condition claims that g_2 satisfies the monotonicity condition in Def. 2.1 for both transitions a_i* and b_j*. $s \in \mathsf{QR}(a_i*) \wedge f_s(g_2) = 1$ means that state s is in $\mathsf{QR}(a_i*)$ and $f_s(g_2)$ is equal to 1. Under this situation, if all direct predecessor states s' (i.e., s' is a state in $pred_s$) are covered by g_2, there is no additional rising transition between s' and s. It preserves that only a falling transition of g_2 is fired in both $\mathsf{QR}(a_i*)$ and $\mathsf{QR}(b_j*)$. □

Before substitution, Prop. 4.1 must be evaluated to validate hazard-freeness. If one of them is not satisfied, the substitution is prevented because it leads to hazardous behavior.

The substitution for a second-level OR gate is also carried out by using Prop. 4.1. For example, if the first-level MC (denoted as g_2) of transition a_1+ substitutes for the second-level OR gate (denoted as g_1) of transitions b_j+ ($j = 1, 2\ldots$), g_2 must satisfy Prop. 4.1 for all MCs of b_j+. On the other hand, a gate g which substitutes for a C-element must have the same logic function as the C-element because no don't cares are assigned to the C-element (see Section 3.2).

4.2 Extended Gate Substitution Algorithm

To apply the gate substitution algorithm to asynchronous SI circuits based on the standard-C architecture, we extend it so that substitution is realized only when Prop. 4.1 is satisfied. **Figure 6** shows a pseudo code of our extended gate substitution algorithm. It evaluates the possibilities of substitution for all pairs of gates. Currently, there is no particular way to decide the order of substitution. However, since the order may affect the quality of optimization, we are going to consider it in our future work.

In our extended gate substitution algorithm, the evaluation of Prop. 4.1 is classified into three cases by the statements of calculated logic functions and CSPFs.

Case 1. $f(g_1) = f(g_2)$**.** If the logic functions of two gates g_1 and g_2 are equivalent, g_2 substitutes for g_1 without checking Prop. 4.1 because it implies that all conditions of Prop. 4.1 are satisfied. The substitution of g_1 for g_2 is also possible.

Case 2. $f(g_2) \subset G(g_1)$ **(or** $f(g_1) \subset G(g_2)$**).** Suppose $f(g_2) \subset G(g_1)$. In this case, our algorithm evaluates whether g_2 substitutes for g_1 or not. $f(g_2) \subset G(g_1)$ means that in all states s where $G_s(g_1)$ is 0 or 1 $f_s(g_2)$ has the same value. Since it means that the first two conditions of Prop. 4.1 are satisfied, only the last condition of Prop. 4.1 must be checked before the substitution. We call the check of Prop. 4.1 in Case 2 as *Case2_check*.

Similarly, in the case of $f(g_1) \subset G(g_2)$, our algorithm evaluates whether g_1 substitutes for g_2 or not.

```
Input: An initial SI circuit and the corresponding SG
Output: An optimized SI circuit

begin
calculate CSPF $G(g_i)$ for all $i$ using SG
while network is changed do
  foreach pair of gates, $g_i$ and $g_j$ $(i \neq j)$ do
    /* Case1 */
    if $f(g_i) = f(g_j)$
      disconnect all connections to $g_j$
      connect $g_i$ to all fanout gates of $g_j$
      recalculate CSPF $G(g_i)$ for all $i$
      break
    /* Case2 */
    if $G(g_i) \supset f(g_j)$ and $Case2\_check$ of $g_j$ wrt $g_i$
      disconnect all connections to $g_i$
      connect $g_j$ to all fanout gates of $g_i$
      recalculate CSPF $G(g_i)$ for all $i$
      break
    if $G(g_j) \supset f(g_i)$ and $Case2\_check$ of $g_i$ wrt $g_j$
      disconnect all connections to $g_j$
      connect $g_i$ to all fanout gates of $g_j$
      recalculate CSPF $G(g_i)$ for all $i$
      break
    /* Case 3 */
    $NewG = G(g_i) \cap G(g_j)$
    if $NewG \neq \emptyset$
      explore a set of new gates $g$ such that $NewG \supset f(g)$
      if $g \in New$ exists and $Case3\_check$ of $g$ wrt $g_i$ and $g_j$
        disconnect all connections to $g_i$, $g_j$
        connect $g$ to all fanout gates of $g_i$, $g_j$
        recalculate CSPF $G(g_i)$ for all $i$
        break
  endforeach
endwhile
end
```

Fig. 6 Extended gate substitution algorithm.

Case 3. Other Cases. In all other cases, we calculate the conjunction of $G(g_1)$ and $G(g_2)$ denoted by $NewG$ which represents the common set of both $G(g_1)$ and $G(g_2)$. If $NewG$ is not empty, a set of new gates g whose $f(g)$ is included in $NewG$ (i.e., $NewG \supset f(g)$) is explored. Such a set is denoted as New. If a gate g in New satisfies all of the conditions of Prop. 4.1 with respect to g_1 and g_2, the newly cleated gate g substitutes for both g_1 and g_2. Different from Case2_check, we call the check of Prop. 4.1 in Case3 as *Case3_check*.

The exploration of a new gate g is explained by using an example in **Fig. 7**. In this example, $G(g_1) = (**10)$, $G(g_2) = (*1*0)$, and $NewG = (*110)$ $(G(g_1) \cap G(g_2))$. Since the inputs of the new gate g is configured by the inputs of both g_1 and g_2 (i.e., n_1 and n_2), we explore the new gate g by the combination of n_1 and n_2 such that $f(g)$ is equal to either (1110) or (0110). In this case, because $f(n_1) = (0101)$ and $f(n_2) = (0011)$, we can identify that $f(g)$ will be (1110) when we take NAND of $x1$ and $x2$. Therefore, a NAND gate g is created in New.

4.2.1 Example

To demonstrate how our extended gate substitution algorithm works, we apply it to an example circuit.

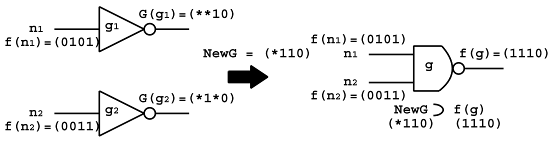


Fig. 7 Creation of a new gate g.

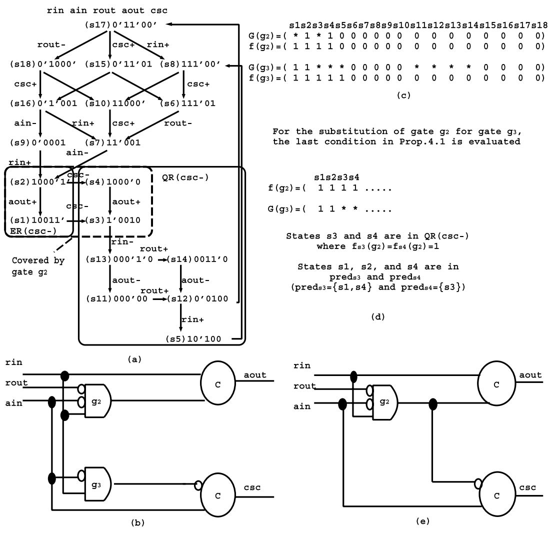

rin ain rout aout csc
(a)
(b)
(c)
(d)
(e)
QR(csc-)
ER(csc-)
Covered by gate g2
For the substitution of gate g2 for gate g3, the last condition in Prop.4.1 is evaluated
States s3 and s4 are in QR(csc-) where fs3(g2)=fs4(g2)=1
States s1, s2, and s4 are in preds3 and preds4 (preds3={s1,s4} and preds4={s3})
aout
csc

Fig. 8 Example of gate substitution.

Figure 8 (a) shows the SG of this example and Fig. 8 (b) shows a part of the corresponding SI circuit. In this example, we focus on the substitution of gate g_2 for gate g_3 at first. Because of $G(g_3) \supset f(g_2)$ (i.e., $G(g_3) = (11{*}{*}{*}0000000000000) \supset f(g_2) = (111100000000000000)$), the substitution of gate g_2 for g_3 is evaluated based on Case2_check.

In Case2_check, only the last condition of Prop. 4.1 is evaluated. For the last condition of Prop. 4.1, states s in QR($csc-$) and $f_s(g_2) = 1$ are enumerated. States $s3$ and $s4$ correspond to such states. In these states, we check whether all of the direct predecessor states of $s3$ and $s4$ are covered by g_2 or not. In states $s1$, $s2$, and $s4$ which are the states of $pred_{s3}$ and $pred_{s4}$ ($pred_{s3} = \{s1, s4\}$ and $pred_{s4} = \{s2\}$), since g_2 covers these states (i.e., $f_{s1}(g_2)$, $f_{s2}(g_2)$, and $f_{s4}(g_2) = 1$), the last condition of Prop. 4.1 is satisfied (see Fig. 8 (d)) As a result, the substitution of gate g_2 for gate g_3 is carried out without leading any hazardous behavior as in Fig. 8 (e).

As another case, to demonstrate how our algorithm blocks an invalid substitution, we consider the substitution of gate g_3 for gate g_2. In this case, we can see a violation of the second condition of Prop. 4.1 in state $s5$. In state $s5$ which is outside of ER($aout+$) ∪ QR($aout+$), both $f_{s5}(g_2)$ and $G_{s5}(g_2)$ are equal to 0. However, because $f_{s5}(g_3)$ is equal to 1 in state $s5$, there exists an additional rising transition of $aout+$ in that state. Since this transition is not specified in the given SG (i.e., $aout+$ is not enabled in state $s5$), we find out that this is a hazard. Because of the violation of Prop. 4.1, our algorithm prevents the substitution.

5. Experimental Results

To evaluate our extended gate substitution algorithm, we applied it to several benchmark circuits derived from asynchronous logic synthesis tool `Petrify`. For this purpose, we implemented our algorithm in JAVA. The experiment was carried out on an Windows 2000 machine which has a Pentium III processor (800 MHz) and a 320 Mbyte memory.

Table 1 shows the results of optimization after our algorithm was applied. The first column shows the name of circuits. The second column shows the number of signals in the original STGs. The third column represents the number of newly inserted signals (i.e., internal signals) by `Petrify` because of logic decomposition. The fourth column represents the number of states in SGs after the new signals are inserted. The fifth and sixth ones show the number of gates and literals in the original SI circuits obtained by `Petrify`. Note gate substitution by using newly inserted signals was carried out on circuits *nak-pa*, *slave*, and *sfir*. The seventh and eighth ones show the results after our algorithm was applied to the original SI circuits. The final column shows the calculation time taken by our algorithm.

The experimental results show that the area reduction ratio by our algorithm is about 14% for both the number of literals and gates. In these optimized circuits, there exist circuits which were not optimized by `Petrify` (*exp* and *pcc* in this experiment) and circuits which were already optimized by `Petrify` (*nak-pa*, *slave*, and *sfir*). Therefore, by using our algorithm as a post-optimizer of `Petrify`, we may get more optimum circuits.

In the following, we describe the current limitations of our algorithm. In addition to them, we briefly show the future direction of this work.

- Our algorithm can handle circuits which have 20,000 states. However, explicitly as-

Table 1 Optimization result.

name	signals	inserted	states	Original SI		Optimized SI		time
	(STG)	signals		gates	lits.	gates	lits.	(sec.)
nak-pa	10	1	60	7	19	6	17	0.19
slave	14	3	311	11	45	9	36	1.45
sfir	13	2	238	11	41	10	38	0.94
exp	5	0	16	5	19	4	14	0.11
pcc	16	0	2240	9	27	8	25	3.37

signing these states to input signals to calculate CSPFs faces to computational difficulties if the number of states is increased. Therefore, to solve this problem, an implicit method by using binary decision diagram (BDD) [1] will be considered in our future work.

- Our algorithm based on Prop. 4.1 is applicable only for the standard-C architecture. To apply our algorithm to other implementations of SI circuits, we need to modify the assignment method of don't cares and the conditions of valid substitution based on the implementability conditions of such implementations.

6. Conclusions

Because in `Petrify` global optimization is restricted to gate substitution after logic decomposition, redundancy may be remained in the synthesized asynchronous SI circuits. To solve this problem, we proposed an alternative optimization method for SI circuits based on transduction methods. Because the property of hazard-freeness must be satisfied in optimized circuits, the gate substitution algorithm in transduction methods was extended by introducing a calculation method of permissible functions and valid conditions of substitution. The extended gate substitution algorithm was implemented in JAVA.

The experimental results showed that the area reduction ratio by our algorithm was about 14% for both the number of literals and gates. In addition, the results showed the usefulness of our algorithm in that it could optimize the circuits which were not optimized by `Petrify` and the circuits which were already optimized by `Petrify`. From this reason, by using our algorithm as a post-optimizer of `Petrify`, we may get more optimum circuits.

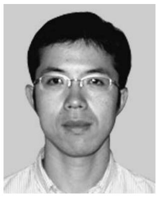
As future work, to optimize large circuits efficiently, we are going to extend our algorithm by using BDD. Another work is to extend our algorithm so that other implementation styles of SI circuits can be optimized.

Acknowledgments We would like to thank Dr. Alex Kondratyev (Cadence Barkley Lab.) and Prof. Jordi Cortadella (Universitat Politecnica de Catalunya) for their useful comments. This work was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (B)13480076 and Special Coordination Fund for Science and Technology.

References

1) Bryant, R.E.: Graph-based algorithms for boolean function manipulation, *IEEE Trans. Comput.*, Vol.C-35, No.8, pp.677–691 (Aug. 1986).
2) Chu, T.A.: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*, Ph.D. thesis, MIT, Boston (1987).
3) Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagnumber, L. and Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronumberus controllers. *IEICE Trans. Inf. Syst.*, Vol.E80-D, No.3, pp.314–325 (March 1997).
4) Futita, M.: A logic synthesis system with multi-level logic circuit minimization mechanism based on transduction methods, *IPSJ Trans.*, Vol.30, No.5, pp.613–623 (May 1989).
5) Kondratyev, A., Cortadella, J., Kishinevsky, M., Lavagnumber, L. and Yakovlev, A.: Logic decomposition of speed-independent circuits, *Proc. IEEE*, Vol.87, No.2, pp.347–362 (Feb. 1999).
6) Muller, D.E. and Bartky, W.S.: A theory of asynchronous circuits, *Proc. International Symposium on the Theory of Switching*, pp.204–243, Harvard University Press (Apr. 1959).
7) Murata, T.: Petri nets: Properties, analysis and applications, *IEEE Press*, Vol.77, pp.541–580 (Apr. 1989).
8) Muroga, S., Kambayashi, Y., Lai, H.C. and Culliney, J.N.: The transduction method—design of logic networks based on permissible functions, *IEEE Trans. Comput.*, Vol.38, No.10, pp.1404–1424 (Oct. 1989).

(Received October 15, 2003)
(Accepted March 5, 2004)



Hiroshi Saito received the B.S and M.S degrees in computer science from the University of Aizu in 1998 and 2000. In 2003, he received the Ph.D. degree in engineering from the University of Tokyo. He is currently a research associate of Research Center for Advanced Science and Technology at the University of Tokyo. His research interests include the computer aided design of asynchronous systems. He is a member of the IEEE and IEICE.

Hiroshi Nakamura received the B.E., M.E., and Ph.D. degree in Electrical Engineering from the University of Tokyo in 1985, 1987, and 1990 respectively. From 1990 to 1996, he was a faculty of Institute of Information Sciences and Engineering at University of Tsukuba, where he was a member of CP-PACS project. He is currently an Associate Professor of Research Center for Advanced Science and Technology at the University of Tokyo. His research interests include computer architecture, high-performance and dependable computing, VLSI design, and bioinformatics. He received the Best Paper Award from IPSJ in 1994 and Sakai Special Researcher Award from IPSJ in 2002. He is a member of the IEEE, the ACM, and the IPSJ.

Masahiro Fujita received the B.S. degree in electrical engineering in 1980, and the M.S. and Ph.D. degrees in information engineering from the University of Tokyo, Tokyo, Japan, in 1982 and 1985, respectively. From 1985 to 1993, he was a Research Scientist with Fujitsu Laboratories, Kawasaki, Japan. From 1994 to 1999, he was the Director of the Advanced Computer-Aided Design Research Group, Fujitsu Laboratories of America, Sunny-vale, CA. He is currently a Professor in the Department of Electrical Engineering, the University of Tokyo, Tokyo, Japan. He has been on program committees for many conferences dealing with digital design and is an Associate Editor of Formal Methods on Systems Design. His primary research interest is in the computer-aided design of digital systems. Dr. Fujita received the Sakai Award from IPSJ in 1984.

Takashi Nanya received the B.E. and M.E. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1969 and 1971, respectively, and the Ph.D. degree in electrical engineering from the Tokyo Institute of Technology, Tokyo, Japan, in 1978. He was with the NEC Central Research Laboratories from 1971 to 1981, and on the faculty of Tokyo Institute of Technology from 1981 to 1996. In 1996, he joined the University of Tokyo where he is a professor. From 2001 to 2004, he served as the director of the Research Center for Advanced Science and Technology. His research interests include dependable computing, VLSI design and asynchronous computing. He received the IEICE Best Paper Award in 1987, the Okawa Publication Award in 1994, and ASP-DAC Best Paper Award in 1998. He has been serving as a vice-chair of IFIP-TC10 "Computer Systems Technology", and IFIP WG10.4 "Dependable Computing and Fault Tolerance". He is a fellow of IEEE and IEICE.