JOURNAL OF ELECTRONIC TESTING: Theory and Applications 18, 487–501, 2002

Design for Consecutive Testability of System-on-a-Chip with Built-In Self Testable Cores

TOMOKAZU YONEDA AND HIDEO FUJIWARA
Graduate School of Information Science, Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara, 630-0101, Japan
tomoka-y@is.aist-nara.ac.jp
fujiwara@is.aist-nara.ac.jp

Received August 28, 2001; Revised January 8, 2002

Editor: Krishnendu Chakrabarty

Abstract. This paper introduces a new concept of testability called consecutive testability and proposes a design-for-testability method for making a given SoC consecutively testable based on integer linear programming problem. For a consecutively testable SoC, testing can be performed as follows. Test patterns of a core are propagated to the core inputs from test pattern sources (implemented either off-chip or on-chip) consecutively at the speed of system clock. Similarly the test responses are propagated to test response sinks (implemented either off-chip or on-chip) from the core outputs consecutively at the speed of system clock. The propagation of test patterns and responses is achieved by using interconnects and consecutive transparency properties of surrounding cores. All interconnects can be tested in a similar fashion. Therefore, it is possible to test not only logic faults but also timing faults that require consecutive application of test patterns at the speed of system clock since the consecutively testable SoC can achieve consecutive application of any test sequence at the speed of system clock.

Keywords: consecutive testability, consecutive transparency, test access mechanism, system-on-a-chip, design for testability, built-in self test

1. Introduction

A fundamental change has taken place in the way digital systems are designed. It has become possible to design an entire system, containing millions of transistors, on a single chip. In order to cope with the growing complexity of such modern systems, designers often use pre-designed, reusable megacells knows as cores. Core-based systems-on-a-chip (SoC) design strategies help companies significantly reduce the time-to-market and design cost for their new products.

However, SoCs are difficult to test after fabrication [16]. In order to make SoC testable, the following three conditions have to be satisfied. (1) *There exist test pattern source (TPS) and test response sink (TRS) for each core*. The TPS generates the test patterns for the embedded core and the TRS observes the test responses. TPS as well as TRS can be implemented either off-chip or on-chip. (2) *There exists test access mechanism for each core*. The test access mechanism propagates test patterns and responses. It can be used for on-chip propagation of test patterns from a TPS to the core-under-test, and for on-chip propagation of test responses from the core-under-test to a TRS. (3) *Interconnects that exist between cores are testable*.

A major difficulty to make SoC testable concerns accessibility of embedded cores. Several techniques of design-for-testability (DFT) have been proposed. There are three main approaches to achieve accessibility of embedded cores. The first approach is based on *test bus architectures* by which the cores are isolated from each other in test mode using a dedicated bus [2–4, 10, 13] or flexible *TESTRAIL* [8] around the cores to propagate test data. Test time reduction is the main objective in the majority of these methods. For example, [3] used an integer linear programming formulation to find the best test assignment and optimize the bandwidth distribution among various test buses to minimize time. The second approach uses *boundary scan architectures* [12, 14] to isolate the core during test. The third approach uses *transparency* [6, 7, 11] or *bypass* [9] mode for embedded cores to reduce the problem to one of finding paths from TPS to core inputs and from core outputs to TRS.

Under the design environment for SoCs, pre-computed test sets are provided for every core. These test sets may contain functional vectors, scan vectors or ordered test sequences for non-scan sequential circuits. They may be for logic faults such as stuck-at faults or timing faults such as delay faults. Moreover, some cores may be able to be at-speed testable in order to increase the coverage of non-modeled and performance-related defects. For that reason, it is necessary to be capable of applying any test sequence to each core and observing any response sequence from the core consecutively at the speed of system clock. We call such a test access *consecutive test access*. Although test bus approach is consecutively test accessible for cores, it is difficult to perform consecutive test access for interconnects. On the other hand, boundary scan, transparency, and bypass mode approaches can test interconnects. However, they are not consecutively test accessible.

There have been reported two works [5, 15] to realize the consecutive test accessibility for both cores and interconnects. In [15], assuming that TPS and TRS are implemented only off-chip (i.e., embedded cores are tested by using external automatic test equipment), we proposed a new testability of SoCs called *consecutive testability*. A consecutively testable SoC consists of *consecutively transparent* cores and can achieve consecutive test access to all cores and all interconnects. Consecutive transparency of a core guarantees consecutive propagation of any test/response sequence from the core input to the core output with some latency. In [5], a synthesis-for-transparency approach was presented to make cores *single-cycle transparent* by embedding multiplexers. This single-cycle transparency is a special case of consecutive transparency of [15] such that the latency of the consecutive transparency is restricted to zero, i.e., single-cycle transparency is the consecutive transparency with zero latency. Therefore, area overhead for making cores consecutively transparent with some latency is generally lower than that for making cores single-cycle transparent (i.e., transparent with zero latency).

In this paper, we consider SoCs that include *BISTed* (Built-In Self Tested) cores and *opaque* cores as well as non-BISTed cores and consecutively transparent cores, and extend the concept of *consecutive testability* of SoCs so that TPS and TRS implemented both on-chip and off-chip can be dealt with. Then, we present a DFT method to make a given SoC consecutively testable. Consecutive testability of an SoC guarantees that, for each core (for each interconnect), by using interconnects and consecutive transparency properties of surrounding cores, test patterns can be fed into the core (the interconnect, respectively) from TPS and the responses can be propagated to TRS consecutively at the speed of system clock. Therefore, consecutively testable SoCs can achieve high quality of test since any test sequence for a core can be applied to the core from TPS and any response sequence can be observed at TRS consecutively at the speed of system clock.

This paper is organized as follows. We introduce an SoC model in Section 2. In Section 3, we introduce the consecutive transparency, the consecutive testability, and present a new test methodology for testing SoCs. We present a graph model for an SoC in Section 4. In Section 5, we present a DFT method for consecutive testability. The experimental results are discussed in Section 6. Finally, Section 7 concludes this paper.

2. System-on-a-Chip

An SoC consists of *cores, primary inputs, primary outputs* and *interconnects* (Fig. 1). For the sake of uniformity, user-defined logic can be considered as another core. Each individual core is testable by either external test or built-in self test. In case a core is testable by external test, a pre-computed test set is available for the core which, if applied to the core, will result in a very high fault coverage. We introduce *ports* of each core as interface points in a natural fashion: signals enter into a core through its *input ports*, and exit through its *output ports*. An interconnect connects an output port with an input port, a primary input with an input port, or an output ports with a primary output. Any number of interconnects can connect to the same output port (i.e., fanout is allowed), but only one

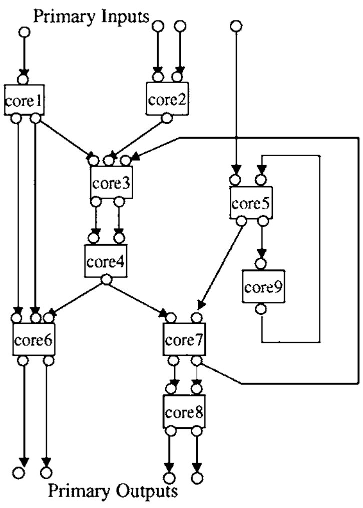


Fig. 1. System-on-a-chip.

interconnect can connect to the same input port. It is not necessary that interconnects are of the same bit width.

3. A Test Methodology for System-on-a-Chip Based on Consecutive Testability

We present a new test methodology for SoCs based on *consecutive testability*. Fig. 2 illustrates a consecutively testable SoC and the consecutive test access to Core 3. A control signal is provided for each core by a test controller (either off-chip or on-chip). Each control signal of a core determines the current test mode of the core called a *configuration*. The types of configurations are consecutive transparencies and functions as a TPS and a TRS. Core 1 works as a TPS for Core 3. Core 2 realizes a consecutive transparency of shaded output port and Core 4 realizes a consecutive transparency of shaded input port. Consecutive transparency of an input port of a core guarantees that any input sequence applied to the input port can propagate to some output ports of the core consecutively at the speed of system clock. Consecutive transparency of an output port of a core guarantees that any output sequence that appears at the output port can propagate from some input ports of the core consecutively at the speed of system clock. Consecutive testability of an SoC guarantees that, for each core (for each interconnect) in the SoC, by selecting configurations of other cores, any test sequence can be consecutively fed into the core (the interconnect, respectively) from TPSs and any response sequence can

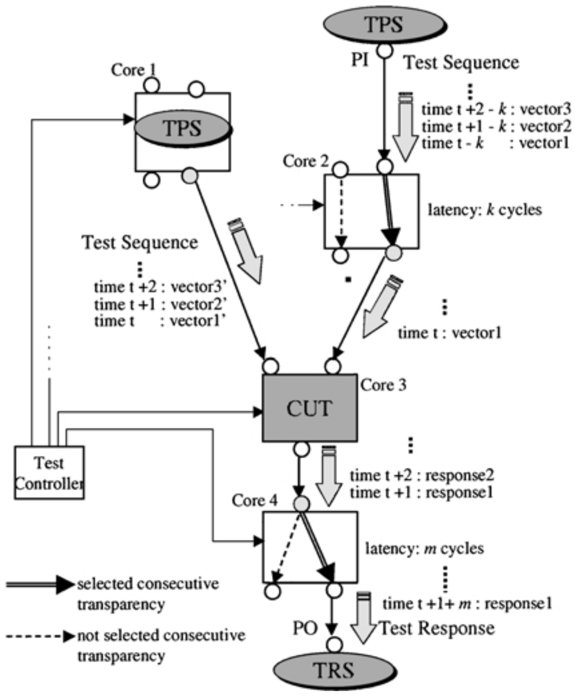


Fig. 2. Consecutive test access.

be consecutively propagated to TRSs through consecutive transparencies of other cores and interconnects. We define the consecutive transparency of a core and the consecutive testability of an SoC in the following subsections.

3.1. Consecutive Transparency of a Core

Definition 1 (Consecutive transparency of a core). Let $I(i)$ be the ith bit of an input port I, and $O(j)$ be the jth bit of an output port O. Suppose that there exists a configuration of a core which can realize a path P between $I(i)$ and $O(j)$. P is called a *consecutively transparent path* if any input sequence applied to $I(i)$ can be consecutively observed at $O(j)$ after some latency, and then $I(i)$ and $O(j)$ are said to be *consecutively transparent*. Moreover, a core is called to be *consecutively transparent* if, for each port of the core, there exists a configuration that can make all bits of the port consecutively transparent.

Fig. 3 illustrates various configurations of a consecutively transparent core. A consecutively transparent core has generally several configurations, and each configuration can be identified by an ID number. By

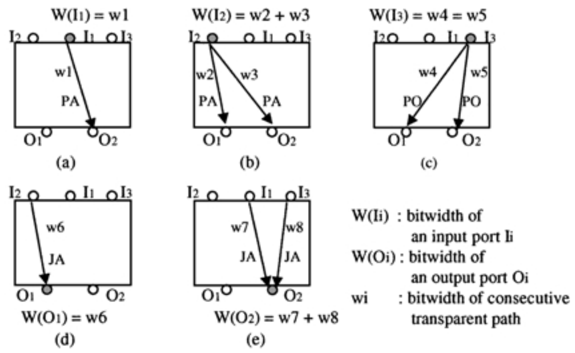

W(I1) = w1
W(I2) = w2 + w3
W(I3) = w4 = w5
PA
PO
JA
W(O1) = w6
W(O2) = w7 + w8
W(Ii) : bitwidth of an input port Ii
W(Oi) : bitwidth of an output port Oi
wi : bitwidth of consecutive transparent path

Fig. 3. Various configurations of a consecutively transparent core. (a) Configuration ID 1; (b) Configuration ID 2; (c) Configuration ID 3; (d) Configuration ID 4; (e) Configuration ID 5.

selecting a configuration of a core, consecutively transparent paths of an I/O port are realized and the I/O port can be made consecutively transparent. For each configuration, all consecutively transparent paths between an input port and an output port are represented as one consecutively transparent path.

We classify consecutively transparent paths into three types, *PA* (Propagation AND), *PO* (Propagation OR), and *JA* (Justification AND). *PA* is a type for a consecutively transparent path of an input port that propagates part of bit-width of test responses applied to the input port. On the other hand, *PO* is a type for a consecutively transparent path of an input port that propagates all bit-width of test response applied to the input port. For an input port, all consecutively transparent paths of type *PA* are necessary to make the input port consecutively transparent. However, only one consecutively transparent path of type *PO* is sufficient to make the input port consecutively transparent. *JA* is a type for a consecutively transparent path of an output port that propagates all or part of bit-width of test sequence which appears at the output port. For an output port, all consecutively transparent paths are necessary to make the output port consecutively transparent.

Fig. 3(a) illustrates type *PA* such that any input sequence applied to an input port I_1 propagates to only one output port O_2. Fig. 3(b) illustrates type *PA* such that any input sequence applied to an input port I_2 propagates to two output ports (O_1 and O_2), where any input sequence of bit width $W(I_2)$ is bit-sliced ($W(I_2) = w2 + w3$) and observed at two output ports (O_1 and O_2). Fig. 3(c) illustrates type *PO* such that any input sequence applied to I_3 propagates to two output ports (O_1 and O_2), where any input sequence of bit width $W(I_3)$ is fanouted ($W(I_3) = w4 = w5$) and observed at two output ports (O_1 and O_2). Fig. 3(d) illustrates type *JA* such that any output sequence that appears at the output port O_1 is propagated from only one input port I_2. Fig. 3(e) illustrates type *JA* such that any output sequence that appears at the output port O_2 is propagated from two input ports (I_1 and I_3), where any output sequence of bit width $W(O_2)$ is constructed by the two input sequences ($W(O_2) = w7 + w8$).

3.2. *Test Pattern Source and Test Response Sink*

The test pattern source (TPS) generates test patterns for cores and interconnects, and the test response sink (TRS) observe the test responses. TPS and TRS can be implemented either off-chip or on-chip. In this paper, we classify TPS and TRS into the following three types (Fig. 4).

1. S_{BIST}. S_{BIST} is a type of TPS and TRS implemented inside of a core (i.e., on-chip) and used for testing the core itself (Fig. 4(c)). A core which has this type of TPS and TRS can be self-testable.
2. S_{off}. S_{off} is a type of TPS and TRS implemented off-chip by external automatic test equipment (ATE) (Fig. 4(a)). TPS of type S_{off} can generate any test sequence of any length, and TRS of type S_{off} can observe any response sequence of any length consecutively at the speed of ATE system clock, which is usually slower than SoC system clock.
3. S_{on}. S_{on} is a type of TPS and TRS implemented inside of a core (i.e., on-chip) and used for testing other cores and interconnects (Fig. 4(b)). Since TPS and TRS of type S_{on} are implemented on-chip, memory spaces for them are limited. Therefore, TPS and TRS of type S_{on} cannot deal with arbitrary long sequences like TPS and TRS of type S_{off}. However,

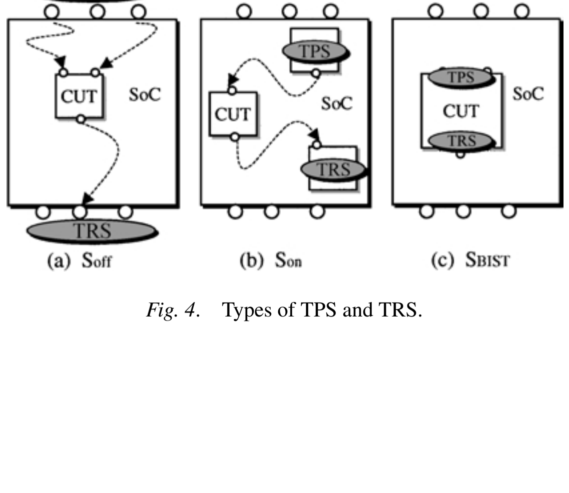
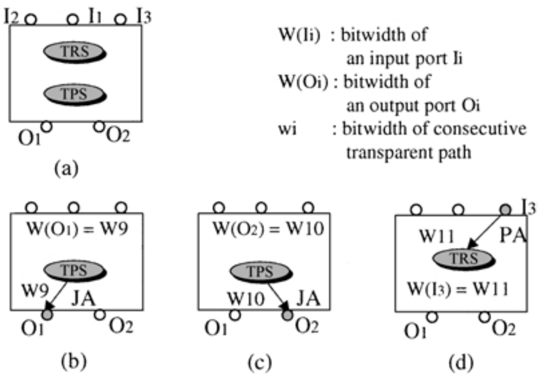


Fig. 4. Types of TPS and TRS.



I2 I1 I3
TRS
TPS
O1 O2
(a)
W(Ii) : bitwidth of an input port Ii
W(Oi) : bitwidth of an output port Oi
wi : bitwidth of consecutive transparent path
W(O1) = W9
W9 JA
(b)
W(O2) = W10
W10 JA
(c)
W11 PA
W(I3) = W11
(d)

Fig. 5. Various configurations of a core that has TPS and TRS of type S_{on}. (a) Son; (b) Configuration ID 6; (c) Configuration ID 7; (d) Configuration ID 8.

within the limited memory spaces, TPS of type S_{on} can generate any test sequence and TRS of type S_{on} can observe any response sequence consecutively at the speed of system clock. A core which can be tested by TPS and TRS of type S_{on} can also be tested by TPS and TRS of type S_{off}. A core which has TPS and TRS of type S_{on} has several configurations (Fig. 5), and each configuration can be identified by an ID number. By selecting a configuration of the core, the core can realize functions as a TPS and a TRS.

3.3. *Consecutive Testability of a System-on-a-Chip*

In this subsection, we introduce a new testability of an SoC called *consecutive testability*. In this paper, we assume that the following informations are given as an SoC.

- Connectivity information between cores
- Test informations of each core
  - type of TPS/TRS that can test the core (S_{BIST} or S_{off} or S_{on})
  - configurations if the core is consecutively transparent
  - configurations if the core has TPS/TRS of type S_{on}

The length of test sequence required to test an interconnect is usually much shorter than that required to test a core. Hence, we assume all interconnect can be tested by TPS/TRS of type S_{on}. In order to test a core, it is necessary to apply test patterns consecutively to all input ports of the core simultaneously. On the other hand, it is not necessary to observe all output ports of the core simultaneously. It is sufficient only to observe one output port at a time. Therefore, we define the consecutive test accessibility of a core and the consecutive test accessibility of an interconnect as follows.

Definition 2 (Consecutive test accessibility of a core). A core C is said to be *consecutively test accessible* if the following two conditions are satisfied at the same time for each output port O of C.

1. Any test sequence generated by the TPS required to test C can be applied to all input ports of C consecutively at the speed of system clock (*consecutive controllability of C for TPS*).
2. Any response sequence appeared at O can be propagated to the TRS required to test C consecutively at the speed of system clock (*consecutive observability of O for TRS*).

Definition 3 (Consecutive test accessibility of an interconnect). For an interconnect E that connects an output port O with an input port I, E is said to be *consecutively test accessible* if O and I satisfies the following two conditions at the same time.

1. Any test sequence generated by the TPS required to test E can be applied to O consecutively at the speed of system clock (*consecutive controllability of E for TPS*).
2. Any response sequence appeared at I can be propagated to the TRS required to test E consecutively at the speed of system clock (*consecutive observability of I for TRS*).

Then, we define the consecutive testability of an SoC as follows.

Definition 4 (Consecutive testability of an SoC). An SoC is said to be *consecutively testable* if all cores and all interconnects in the SoC are consecutively test accessible.

4. Graph Modeling

In this section, we define a core connectivity graph to represent an SoC, and consider the consecutive testability on the graph.

Definition 5 (Core connectivity graph). We define a *core connectivity graph* $G = (V, E, \lambda)$ as a following directed graph to represent an SoC.

- $V = V_{PI} \cup V_{PO} \cup V_{in} \cup V_{out} \cup V_{source} \cup V_{sink}$ where V_{PI} is the set of all PIs of the SoC, V_{PO} is the set of all POs of the SoC, V_{in} is the set of all input ports of cores in the SoC, and V_{out} is the set of all output ports of cores in the SoC. V_{source} is the set of all TPSs of type S_{on} in the SoC. V_{sink} is the set of all TRSs of type S_{on} in the SoC.
- $E = E_{core} \cup E_{net}$ where $E_{core} = \{(x, y) \in V_{in} \times V_{out} \mid$ input port x is connected to output port y by a consecutively transparent path$\}$, and $E_{net} = \{(y, x) \in V_{out} \times V_{in} \mid$ output port y is connected to input port x by an interconnect$\}$.
- Labeling function $\lambda : E \rightarrow 2^{C \times I \times T \times W}$ where C is the set of all cores in the SoC, I is the set of all ID numbers of configurations, $T = \{JA, JO, PA, PO \mid$ types of consecutively transparent path (JO is for fanouted interconnects)$\}$, and W is the set of all bit widths of $e \in E$. Especially for $e \in E_{net}$, $\lambda(e) = \{\{\phi, \phi, JO,$ bit width of $e\}, \{\phi, \phi, PO,$ bit width of $e\}\}$.

Fig. 6 illustrates a core connectivity graph G which corresponds to the SoC of Fig. 1. Fig. 7 illustrates edges labeled by λ which correspond to the core of Figs. 3 and 5.

We refer to a vertex that has no input edge as a *source*, and a vertex that has no output edge as a *sink*. For a core connectivity graph G, selecting a configuration of a core is to leave edges which have labels of the configuration and to remove other edges from the core.

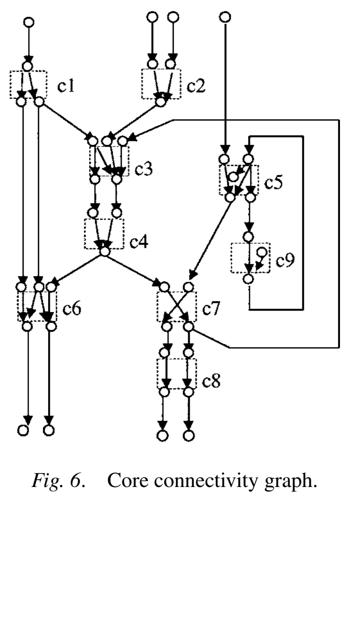


Fig. 6. Core connectivity graph.

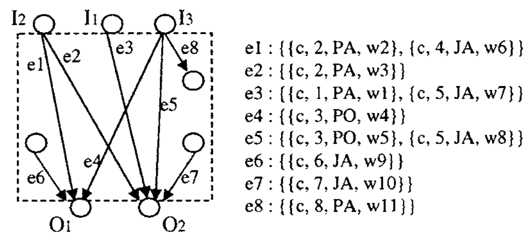


Fig. 7. Label by λ.

Then, we define a justification subgraph of a core, a justification subgraph of an interconnect and a propagation subgraph of a port as subgraphs of a core connectivity graph.

Definition 6 (Justification subgraph of a core). Let $G = (V, E, \lambda)$ be a core connectivity graph of an SoC and $G_J = (V_J, E_J, \lambda)$ be an acyclic subgraph of G. For a core $c \in C$, G_J is called a *justification subgraph of c* if G_J satisfies all the following conditions.

1. All input ports of c are sinks in G_J and there exists no sink except for all input ports of c in G_J.
2. For each edge $u \in E_J$, u has a label of either JO or JA.
3. Let $G' = (V', E', \lambda)$ be a subgraph of G obtained by selecting a configuration for each core. For each edge $u \in E_J$,
   (a) u contains all input edges of u in G', and
   (b) u contains only one output edge of u in G' when output edges have labels of JO in G'.

Lemma 1. *Let V_S be the set of all source vertices in G_J of core c. Then c is consecutively controllable for V_S.*

Proof: By Definition 5 and condition 2 of Definition 6, all edges in G_J can be used to apply test patterns consecutively at the speed of system clock since each edge in G_J represents either a consecutively transparent path or an interconnect, and has a label of either JO or JA. By condition 1 of Definition 6, there exist simple paths from more than one element in V_S to each input port of c. By condition 3 of Definition 6, all edges in the same core have the same ID number of configuration since only one configuration is selected for each core. Let v be the vertex in V_{out} (i.e., v is an output port of a core). If a configuration to realize a consecutive transparency of v is selected, all consecutively transparent paths for v exist in G_J (condition 3(a) of Definition 6). If a configuration to realize a consecutive transparency of v is not selected, v is a source vertex in G_J. By condition 3(b) of Definition 6, it is possible to apply any test sequence for all simple paths at the same time.

Therefore, we can see that any test sequence generated at V_S can be applied to all input ports of c along all simple paths in G_J consecutively at the speed of system clock. This completes the proof. □

Definition 7 (Justification subgraph of an interconnect). Let $G = (V, E, \lambda)$ be a core connectivity graph of an SoC and $G_J = (V_J, E_J, \lambda)$ be an acyclic subgraph of G. For an interconnect $e = (y, x) \in E_{net}$, G_J is called a *justification subgraph of e* if G_J satisfies all the following conditions.

1. Only y is a sink in G_J.
2. For each edge $u \in E_J$, u has a label of either *JO* or *JA*.
3. Let $G' = (V', E', \lambda)$ be a subgraph of G obtained by selecting a configuration for each core. For each edge $u \in E_J$,
   (a) u contains all input edges of u in G', and
   (b) u contains only one output edge of u in G' when output edges have labels of *JO* in G'.

Lemma 2. *Let V_S be the set of all source vertices in G_J of interconnect e. Then e is consecutively controllable for V_S.*

Proof: The proof is similar to the proof of Lemma 1. □

Definition 8 (Propagation subgraph of a port). Let $G = (V, E, \lambda)$ be a core connectivity graph of an SoC and $G_P = (V_P, E_P, \lambda)$ be an acyclic subgraph of G. For a vertex $v \in V$, G_P is called *propagation subgraph of v* if G_P satisfies all the following conditions.

1. Only v is a source in G_P.
2. For each edge $u \in E_P$, u has a label of either *PO* or *PA*.
3. Let $G' = (V', E', \lambda)$ be a subgraph of G obtained by selecting a configuration for each core. For each edge $u \in E_P$,
   (a) u contains all output edges of u in G' when the output edges have labels of *PA*, and
   (b) u contains more than one output edge of u in G' when the output edges have labels of *PO* in G'.

Lemma 3. *Let V_E be the set of all sink vertices in G_P of vertex v. Then v is consecutively observable for V_E.*

Proof: By Definition 8 and condition 2 of Definition 6, all edges in G_P can be used to propagate test responses consecutively at the speed of system clock since each edge in G_P represents either a consecutively transparent path or an interconnect, and has a label of either *PO* or *PA*. By condition 1 of Definition 8, there exist simple paths from v to each element in V_E. By condition 3 of Definition 8, all edges in the same core have the same ID number of configuration since only one configuration is selected for each core. Let v' be the vertex in V_{in} (i.e., v' is an input port of a core). If a configuration to realize a consecutive transparency of v' is selected and the consecutively transparent paths for v' are type *PA*, all consecutively transparent paths for v' exist in G_P (condition 3(a) of Definition 8). If a configuration to realize a consecutive transparency of v' is selected and the consecutively transparent paths for v' are type *PO*, there exist at least one consecutively transparent path for v' G_P (condition 3(b) of Definition 8). If a configuration to realize a consecutive transparency of v' is not selected, v' is a sink vertex in G_P.

Therefore, we conclude that any response sequence appeared at v can be propagate to V_E along all simple paths in G_P consecutively at the speed of system clock. This completes the proof. □

Theorem 1. *Let $G = (V, E, \lambda)$ be a core connectivity graph of an SoC. An SoC is said to be* consecutively testable *if the SoC satisfies the following two conditions.*

1. *For each output port $v \in V_{out}$ of each core $c \in C$, there exist one* justification subgraph *G_J of c and one* propagation subgraph *G_P of v where G_J and G_P are disjoint and satisfy the following conditions.*
   - *if TPS/TRS type required to test c is S_{BIST} $G_J = G_P = \phi$*
   - *if TPS/TRS type required to test c is S_{off} $V_S \subseteq V_{PI}$, $V_E \subseteq V_{PO}$*
   - *if TPS/TRS type required to test c is S_{on} $V_S \subseteq (V_{PI} \cup V_{source})$, $V_E \subseteq (V_{PO} \cup V_{sink})$.*
2. *For each interconnect $e = (y, x) \in E_{net}$, there exist one* justification subgraph *G_J of e and one*

propagation subgraph G_P *of* x *where* G_J *and* G_P *are disjoint and satisfy the following conditions.*

- $V_S \subseteq (V_{PI} \cup V_{source})$, $V_E \subseteq (V_{PO} \cup V_{sink})$.

Proof: The proof follows from Definitions 2–4 and Lemmas 1–3. □

5. DFT for Consecutive Testability

This section presents a *design-for-testability* (DFT) method that makes a given SoC consecutively testable. We assume that each individual core is testable by either external test or built-in self test. In case a core is testable by external test, a pre-computed test set is available for the core which, if applied to the core, will result in a very high fault coverage. Additionally, we assume (i) the internal design of the cores cannot be modified by DFT due to IP (Intellectual Property) protection and (ii) control signals for configurations can be controlled independently of normal operations. In the rest of this paper, we consider the DFT under such assumptions.

5.1. Problem Formulation

Each core (interconnect) in a consecutively testable SoC is consecutively controllable for the required TPS and consecutively observable for the required TRS. In other words, for each output port v of each core $c \in C$, a core connectivity graph G that represents a consecutively testable SoC has one justification subgraph G_J of c and one propagation subgraph G_P of v where G_J and G_P are disjoint and satisfy the condition 1 of Theorem 1. Similarly, for each interconnect $e = (y, x) \in E_{net}$, there exist one justification subgraph G_J of e and one propagation subgraph G_P of x where G_J and G_P are disjoint and satisfy the condition 2 of Theorem 1.

When a core (an interconnect) in a given SoC is not consecutively controllable for the required TPS, paths from the TPS are added by using *test multiplexers* (MUXes) in the proposed DFT (Fig. 8(a)). Similarly, when a core (an interconnect) in a given SoC is not consecutively observable for the required TRS, paths to the TRS are added by using test MUXes (Fig. 8(a)). When an interconnect-under-test is directly connected to an input or output port of a core which is not consecutively transparent, it is necessary to isolate the interconnect from the core in order to make the interconnect

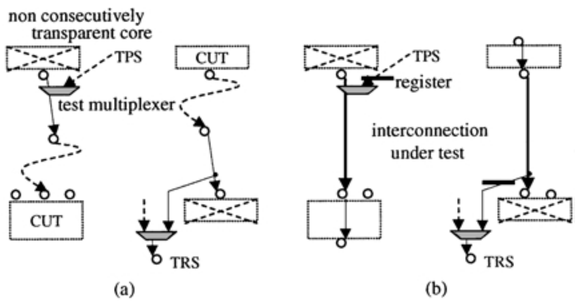

non consecutively transparent core
TPS
CUT
test multiplexer
CUT
TRS
(a)
TPS
register
interconnection under test
TRS
(b)

Fig. 8. DFT elements. (a) DFT for consecutive test access; (b) DFT for isolation of interconnection under test.

consecutively test accessible. This isolation is implemented by using test MUXes and *registers* (Fig. 8(b)). Assuming that any SoC includes enough number of TPSs and TRSs to make each core (each interconnect) consecutively controllable and observable, we formulate a DFT for making the SoC consecutively testable as the following optimization problem.

Definition 9. DFT for consecutive testability
Input: An SoC (a core connectivity graph)
Output: A consecutively testable SoC
Optimization: Minimizing hardware overhead (i.e., total bit width of added MUXes and registers).

5.2. DFT Algorithm

We propose a DFT algorithm for consecutive testability. The algorithm consists of the following four stages.

Stage 1. Augment a given SoC so that all cores are consecutively controllable for the required TPS.
Stage 2. Augment a given SoC so that all cores are consecutively observable for the required TRS.
Stage 3. Augment a given SoC so that all interconnects are consecutively controllable for the required TPS.
Stage 4. Augment a given SoC so that all interconnects are consecutively observable for the required TRS.

5.2.1. DFT for Consecutive Controllability of Cores (Stage 1). The objective of the first stage is to modify a given SoC with minimum hardware overhead so that all cores are consecutively controllable for the required TPS (i.e., each core $c \in C$ has a justification subgraph G_J of c where G_J satisfies the condition 1 of Theorem 1). The strategy of the algorithm is that, for each core, it first creates *control initial graph*, and then, it creates *control middle graph*. After that, it induces conditions such that the control middle graph satisfies the Definition 6 and the core is consecutively controllable for the required TPS. Finally, the algorithm formulates the DFT in this stage as *an integer linear programming problem*. All cores are made consecutively controllable with minimum hardware overhead by solving the integer linear programming problem.

5.2.1.1. Step 1: Creation of Control Initial Graph. The *control initial graph G_{J_c}* of a core $c \in C$ is created from a core connectivity graph G as follows.

1. Remove the edges which have labels of c and let the vertices which correspond to the input ports of c be sinks.
2. Remove the edges which have labels of neither *JA* nor *JO*.
3. We define the control initial graph G_{J_c} as the set of vertices and edges reachable to sinks.

Fig. 9 illustrates a control initial graph $G_{J_{c6}}$. Each edge in $G_{J_{c6}}$ has a label of either *JO* or *JA* and the number beside $e \in E_{core}$ represents a label of configuration ID.

Let A_{J_c} be the set of cores that exist in G_{J_c}. Here, a core $c' \in C$ that exists in G_{J_c} means that there exists more than one edge which has a label of c' in G_{J_c}. For each $a \in A_{J_c}$, let B_{J_a} be the set of all configuration IDs of a. We define K_{J_c} as the following equation.

$$K_{J_c} = \prod_{a \in A_{J_c}} B_{J_a}$$
$$= B_{J_{a1}} \times B_{J_{a2}} \times B_{J_{a3}} \times \dots .$$

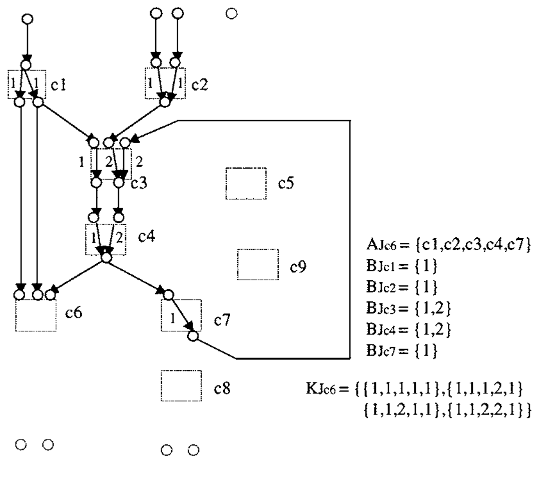


Fig. 9. Control initial graph $G_{J_{c6}}$.

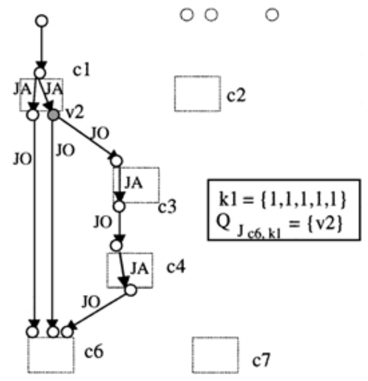


Fig. 10. Control middle graph $G_{J_{c6,k1}}$.

A control initial graph G_{J_c} contains several configurations for each core $a \in A_{J_c}$, and consecutive transparency of each core $a \in A_{J_c}$ is not realized.

5.2.1.2. Step 2: Creation of Control Middle Graph. For each $k \in K_{J_c}$, the *control middle graph $G_{J_{c,k}}$* is created from a control initial graph G_{J_c} as follows.

1. For each $a \in A_{J_c}$, select a configuration that corresponds to k.
2. We define the control middle graph $G_{J_{c,k}}$ as the set of vertices and edges reachable to sinks.

Fig. 10 illustrates a control middle graph $G_{J_{c6,k1}}$. *JO* and *JA* beside $e \in E$ represent types of consecutively transparent path e. A control middle graph $G_{J_{c,k}}$ contains only one configuration for each core $a \in A_{J_c}$, and consecutive transparency of each core $a \in A_{J_c}$ is realized.

For $G_{J_{c,k}}$, let $Q^1_{J_{c,k}}$, $Q^2_{J_{c,k}}$ and $Q^3_{J_{c,k}}$ be the sets of all vertices $q \in G_{J_{c,k}}$ that satisfies the following conditions respectively (Fig. 11).

1. $Q^1_{J_{c,k}}$: q is a source.
2. $Q^2_{J_{c,k}}$: q has more than two output edges which have labels of *JO*.
3. $Q^3_{J_{c,k}}$: There exist cycles which contain q.

We define $Q_{J_{c,k}}$ as follows.

$$Q_{J_{c,k}} = Q^1_{J_{c,k}} \cup Q^2_{J_{c,k}} \cup Q^3_{J_{c,k}}$$

$Q^2_{J_{c,k}}$ and $Q^3_{J_{c,k}}$ are the sets of all vertices that do not satisfy the Definition 6. Moreover, we define $Q^{1,PI}_{J_{c,k}}$ and

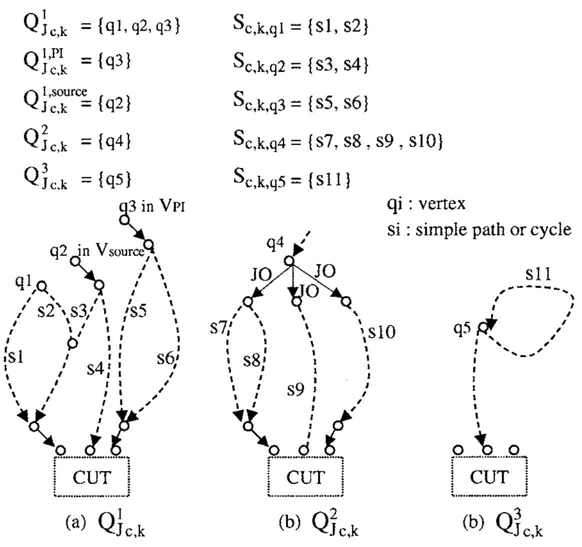

$Q^1_{J_{c,k}} = \{q1, q2, q3\}$
$Q^{1,PI}_{J_{c,k}} = \{q3\}$
$Q^{1,source}_{J_{c,k}} = \{q2\}$
$Q^2_{J_{c,k}} = \{q4\}$
$Q^3_{J_{c,k}} = \{q5\}$
$S_{c,k,q1} = \{s1, s2\}$
$S_{c,k,q2} = \{s3, s4\}$
$S_{c,k,q3} = \{s5, s6\}$
$S_{c,k,q4} = \{s7, s8, s9, s10\}$
$S_{c,k,q5} = \{s11\}$
q3 in VPI
q2 in Vsource
qi : vertex
si : simple path or cycle
q1
q4
q5
JO
s1
s2
s3
s4
s5
s6
s7
s8
s9
s10
s11
CUT
(a) $Q^1_{J_{c,k}}$
(b) $Q^2_{J_{c,k}}$
(b) $Q^3_{J_{c,k}}$

Fig. 11. $Q^1_{J_{c,k}}, Q^2_{J_{c,k}}, Q^3_{J_{c,k}}$.

$Q^{1,source}_{J_{c,k}}$ as follows.

$$Q^{1,PI}_{J_{c,k}} = Q^1_{J_{c,k}} \cap V_{PI}, \quad Q^{1,source}_{J_{c,k}} = Q^1_{J_{c,k}} \cap V_{source}$$

Then, let $S_{c,k,q}$ be the set of all simple paths from q in $Q_{J_{c,k}}$ to each sink vertex in $G_{J_{c,k}}$.

5.2.1.3. Step 3: Integer Linear Programming Formulation. We define the following variables as integer linear programming variables.

$$y_c = \begin{cases} 1 & \text{core } c \text{ is consecutively controllable for TPS} \\ 0 & \text{otherwise} \end{cases}$$

$$a_{c,k} = \begin{cases} 1 & G_{J_{c,k}} \text{ is consecutively controllable for TPS} \\ 0 & \text{otherwise} \end{cases}$$

$$d_{c,k,q} = \begin{cases} 1 & G_{J_{c,k}} \text{ is consecutively controllable for vertex } q \\ 0 & \text{otherwise} \end{cases}$$

$$z_{q,r} = \begin{cases} 1 & \text{output edge } r \text{ of vertex } q \text{ is consecutively controllable for } q \\ 0 & \text{otherwise} \end{cases}$$

$$m_s = \begin{cases} 1 & \text{if MUX is inserted to simple path } s \\ 0 & \text{otherwise} \end{cases}$$

$$x_e = \begin{cases} 1 & \text{if MUX is inserted to interconnect } e \\ 0 & \text{otherwise} \end{cases}$$

The following integer linear programming formulation minimizes the test overhead (i.e., total bit width of MUXes) while making all cores consecutively controllable.

Minimize

$$\sum_{e \in E_{net}} x_e \cdot width(e) \tag{1}$$

Subject to:

1. for each core $c \in C$,

$$y_c \geq 1 \tag{2}$$

2. For each core $c \in C$ which can be tested by either S_{off} or S_{on},

$$\sum_{k \in K_{J_c}} a_{c,k} \geq y_c \tag{3}$$

3. For each element $k \in K_{J_c}$,

$$\sum_{q \in Q_{J_{c,k}}} d_{c,k,q} \geq |Q_{J_{c,k}}| \cdot a_{c,k} \tag{4}$$

$|Q_{J_{c,k}}|$ is a constant value which represents the number of elements in $Q_{J_{c,k}}$.

4. (a) In case TPS type required to test c is S_{off} for each vertex $q \in (Q^1_{J_{c,k}} - Q^{1,PI}_{J_{c,k}})$,

$$\sum_{s \in S_{c,k,q}} m_s \geq |S_{c,k,q}| \cdot d_{c,k,q} \tag{5}$$

(b) In case TPS type required to test c is S_{on}, for each vertex $q \in (Q^1_{J_{c,k}} - (Q^{1,PI}_{J_{c,k}} \cup Q^{1,source}_{J_{c,k}}))$,

$$\sum_{s \in S_{c,k,q}} m_s \geq |S_{c,k,q}| \cdot d_{c,k,q} \tag{6}$$

5. For each vertex $q \in Q^2_{J_{c,k}}$, let $R_{c,k,q}$ be the set of all output edges of q. For each element $r \in R_{c,k,q}$, let $S^r_{c,k,q}$ be the set of all simple paths between r and all sink vertices in $G_{J_{c,k}}$. Then, for each vertex $q \in Q^2_{J_{c,k}}$,

$$\sum_{r \in R_{c,k,q}} z_{q,r} \geq d_{c,k,q} \tag{7}$$

$$\sum_{s \in (S_{c,k,q} - S^r_{c,k,q})} m_s \geq \left|S_{c,k,q} - S^r_{c,k,q}\right| \cdot z_{q,r} \tag{8}$$

6. For each vertex $q \in Q^3_{J_{c,k}}$,

$$\sum_{s \in S_{c,k,q}} m_s \geq |S_{c,k,q}| \cdot d_{c,k,q} \tag{9}$$

7. For each simple path $s \in S_{c,k,q}$,

$$\sum_{e \in E_s} x_e \geq m_s \tag{10}$$

E_s represents the set of all edges which correspond to interconnects in simple path s.

Equation (2) guarantees that all cores are consecutively controllable for the required TPS. If TPS type required to test a core is either S_{off} or S_{on}, more than one $G_{J_{c,k}}$ must be consecutively controllable for the TPS. This is guaranteed by equation (3). In order to make $G_{J_{c,k}}$ consecutively controllable for the TPS, all vertices in $Q_{J_{c,k}}$ must be consecutively controllable for the TPS. This is guaranteed by Eq. (4). In order to make q in $Q^1_{J_{c,k}}$ consecutively controllable for the TPS, all simple paths in $S_{c,k,q}$ must be inserted MUXes and paths from TPS must be added. However, if q is a vertex that represents the TPS required to test the core, q is already consecutively controllable for the TPS. This is guaranteed by Eqs. (5) and (6). Each vertex q in $Q^2_{J_{c,k}}$ has more than two output edges which have label of JO, and all the edges propagate only the same sequence. In order to make q in $Q^2_{J_{c,k}}$ consecutively controllable for the TPS, MUXes must be inserted to all simple paths which include all the output edges except one output edge. This is guaranteed by Eqs. (7) and (8). In order to make q in $Q^3_{J_{c,k}}$ consecutively controllable for the TPS, all cycles which contain q must be broken by MUXes and paths from TPS must be added. This is guaranteed by Eq. (9). Let s be a simple path and let E_s be the set of all edges which correspond to interconnects in s, insertion of MUX to s means that MUX is inserted to more than one element in E_s. This is guaranteed by Eq. (10).

Test MUXes are inserted to the edges obtained by solving the above integer linear programming problem (Fig. 12), and all cores can be made consecutively controllable with minimum hardware overhead. However, in case TPS type required to test core c is S_{off}, it is necessary to add TPSs of type S_{off} (i.e., add vertices to V_{PI}) if the sum bit width of edges that must be inserted MUXes to make c consecutive controllable is larger than that of available TPSs (i.e., vertices in $V_{PI} - Q^{1,PI}_{J_{c,k}}$). Similarly, in case TPS type required to test core c is S_{on}, it is necessary to add TPSs of type

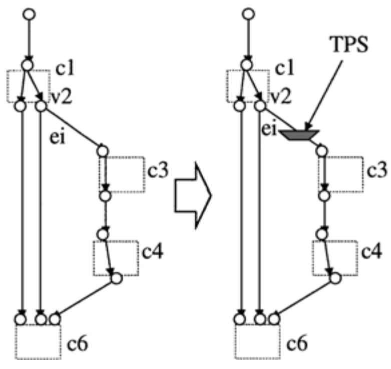


Fig. 12. Insertion of a MUX to an edge e_i for consecutive controllability.

S_{on} (i.e., add vertices to V_{source}) if the same condition as above is satisfied.

5.2.2. DFT for Consecutive Observability of Cores (Stage 2). The objective of the second stage is to modify a given SoC with minimum hardware overhead so that all cores are consecutively observable for the required TRS (i.e., each output port $v \in V_{out}$ of cores has a propagation subgraph G_P of v where G_P satisfies the condition 1 of Theorem 1). The strategy of the algorithm is that, for each core,

Step 1: remove consecutively controllable paths for the core.
Step 2: create observation initial graph.
Step 3: create observation middle graph.
Step 4: formulate as integer linear programming problem.

In Step 1, the algorithm first selects each configuration for the core in order to realize the consecutively controllable paths which are already decided in Stage 1. Then, it removes the consecutively controllable paths from the core connectivity graph G in order to guarantee that consecutively controllable paths and consecutively observable paths are disjoint. Procedures for Step 2, Step 3, and Step 4 are similar to Step 1, Step 2, and Step 3 in Stage 1, respectively. In Step 2, it creates *observation initial graph* for each output port of the core. After that, it creates *observation middle graph* in Step 3. In Step 4, the algorithm induces conditions such that the observation middle graph satisfies the Definition 8 and the output port of the core is consecutively observable for the required TRS. Finally, the algorithm formulates the DFT in this stage as *an integer linear* *programming problem*. All cores can be made consecutively observable with minimum hardware overhead by solving the integer linear programming problem.

5.2.3. DFT for Consecutive Controllability of Interconnects (Stage 3). The objective of the third stage is to modify a given SoC with minimum hardware overhead so that all interconnects are consecutively controllable for the required TPS (i.e., each interconnect $e \in E_{net}$ has a justification subgraph G_J of e where G_J satisfies the condition 2 of Theorem 1). The algorithm for this stage is similar to that of Stage 1 except for addition of test registers. When an interconnect-under-test is directly connected to the output port of a core which is not consecutively transparent (i.e., control initial graph $G_{J_{e,k}}$ for the interconnect e is empty), it is necessary to isolate the interconnect from the core in order to make the interconnect consecutively test accessible (Fig. 8(b)). Therefore, this stage can be formulated as the following integer linear programming problem exchanging the objective function (Eq. (1)) and constraint 2 (Eq. (3)) as follows.

Minimize

$$\sum_{e \in E_{net}} (x_e + x_{e,reg}) \cdot width(e) \tag{11}$$

Subject to:

2. (a) if $G_{J_{e,k}}$ is empty,

$$x_e \geq 1 \tag{12}$$

$$x_{e,reg} \geq 1 \tag{13}$$

(b) otherwise,

$$\sum_{k \in K_{J_e}} a_{e,k} \geq y_e \tag{14}$$

Here, $x_{e,reg}$ is the integer linear programming variable, and is equal to one if a test register is inserted to interconnect e, otherwise, is equal to zero.

All interconnects can be made consecutively controllable with minimum hardware overhead by solving the integer linear programming problem.

5.2.4. DFT for Consecutive Observability of Interconnects (Stage 4). The objective of the fourth stage is to modify a given SoC with minimum hardware overhead so that all interconnects are consecutively observable for the required TRS (i.e., each interconnect $e \in E_{net}$ has a propagation subgraph G_P of e where G_P satisfies the condition 2 of Theorem 1). The algorithm for this stage is similar to that of Stage 2 except for addition of test registers. The procedure for addition of test registers can be presented in a similar fashion to Stage 3. Therefore, all interconnects can be made consecutively observable with minimum hardware overhead by solving the integer linear programming problem.

After above four stages, we can modify a given SoC so that all cores and all interconnect are consecutively controllable and observable (i.e., consecutively testable).

6. Experimental Results

In this section, we present experimental results of the proposed method. We applied the method to three SoC examples shown in Fig. 13. System S_1 consists of four consecutively transparent cores and two non-consecutively transparent cores, and it contains one on-chip TPS inside of core5 and one on-chip TRS inside of core1. System S_2 consists of six consecutively transparent cores and it has the same connectivity information and on-chip TPS/TRS as system S_1. System S_3 consists of eight consecutively transparent cores and have no on-chip TPS/TRS.

We used the *lp_solve* package from Eindhoven University of Technology [1]. Assuming that all interconnects are of the same bit-width, the running time is negligible (less than 0.01 second) for each stage of all examples on a SUN Ultra 5 workstation. The results of the running SoC examples are shown in Table 1. Column "systems" denotes system name. Columns "Stage 1" and "Stage 2", "Stage 3", "Stage 4" and "Total" denote the number of DFT elements added in Stage 1, Stage 2, Stage 3, Stage 4 and all four stages, respectively. e_i shows the edge to which DFT element is added.

Table 2 shows the estimations of the area overhead. Column "interconnects" denotes the bit width of interconnects in each system. Columns "0.1" and "0.5", "1" and "10" of each system denote the area overhead when we assume that the system contains 0.1 million gates, 0.5 million gates, 1 million gates and 10 million gates, respectively. For example, "0.61" in the first row of the column "0.1" of system S_1 shows the area overhead of the DFT element (5 MUXes and 4 registers) when we assume that S_1 contains 0.1 million

Table 1. Results of the running SoC examples.

	# DFT elements				
Systems	Stage 1	Stage 2	Stage 3	Stage 4	Total
S_1	MUX: 2 (e_3, e_5)	MUX: 0	MUX: 1 (e_7) Reg: 2 (e_5, e_7)	MUX: 2 (e_3, e_8) Reg: 2 (e_3, e_8)	MUX: 5 Reg: 4
S_2	MUX: 1 (e_1)	MUX: 0	MUX: 0 Reg: 0	MUX: 0 Reg: 0	MUX: 1 Reg: 0
S_3	MUX: 2 (e_4, e_5)	MUX: 3 (e_3, e_4, e_5)	MUX: 2 (e_1, e_9) Reg: 0	MUX: 1 (e_9) Reg: 0	MUX: 8 Reg: 0

Table 2. Estimation of area overhead (%).

	System S_1 (million gates)				System S_2 (million gates)				System S_3 (million gates)			
Interconnects (bit)	0.1	0.5	1	10	0.1	0.5	1	10	0.1	0.5	1	10
32	0.61	0.12	0.06	0.01	0.10	0.02	0.01	0.00	0.77	0.15	0.08	0.01
64	1.22	0.24	0.12	0.01	0.19	0.04	0.02	0.00	1.54	0.31	0.15	0.02
128	2.44	0.48	0.24	0.02	0.38	0.08	0.04	0.00	3.08	0.62	0.31	0.03
256	4.87	0.97	0.49	0.05	0.77	0.15	0.08	0.01	6.15	1.23	0.62	0.06
512	9.73	1.94	0.97	0.10	1.54	0.31	0.15	0.02	12.29	2.46	1.23	0.12
1024	19.46	3.89	1.95	0.20	3.07	0.61	0.31	0.03	24.58	4.92	2.46	0.25

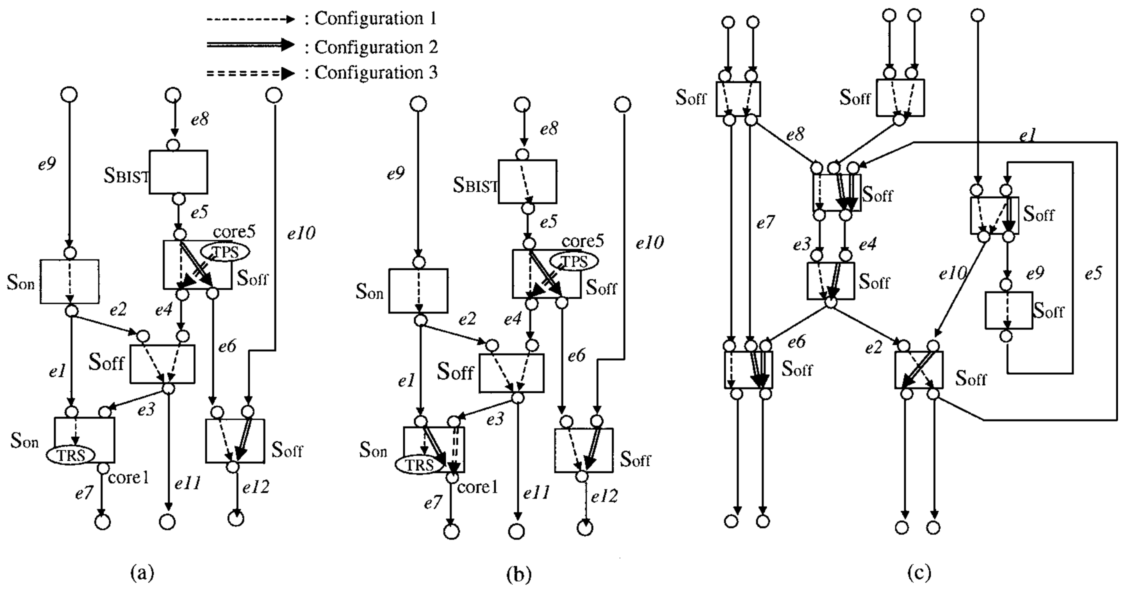


Fig. 13. SoC examples. (a) System S_1; (b) System S_2; (c) System S_3.

gates and all interconnects in S_1 are of 32 bit width.

In our proposed method, the delay overhead is negligible since at most only one multiplexer is inserted to each interconnect.

7. Conclusions

In this paper, we introduced a new testability called consecutive testability. For a consecutively testable SoC, testing can be performed as follows. Test patterns of a core are propagated to all input ports of the core from TPS, and the test responses appeared at an output port of the core are propagated to TRS consecutively at the speed of system clock. The propagation of test patterns and responses is achieved by using interconnects and consecutively transparent paths of surrounding cores. All interconnects can be tested in a similar fashion. Therefore, it is possible to apply any test sequence and observe any response sequence consecutively at the speed of system clock. We also proposed a design-for-testability method for making a given SoC consecutively testable based on integer linear programming problem. Our future work is to propose a DFT method for making cores consecutively transparent with minimum hardware overhead.

Acknowledgments

This work was sponsored in part by NEDO (New Energy and Industrial Technology Development Organization) through the contract with STARC (Semiconductor Technology Academic Research Center) and supported in part by Foundation of Nara Institute of Science and Technology under the Grant for Activity of Education and Research. Authors would like to thank Toshimitsu Masuzawa (Osaka University), Michiko Inoue and Satoshi Ohtake (Nara Institute of Science and Technology) for their valuable discussion.

References

1. M. Berkelaar, *lp_solve*, version 3.2, Eindhoven University of Technology, The Netherlands, *ftp://ftp.ics.ele.tue.nl/pub/lp_solve*.
2. S. Bhatia, T. Gheewala, and P. Varma, "A Unifying Methodology for Intellectual Property and Custom Logic Testing," in *Proc. 1996 Int. Test Conf.*, Oct. 1996, pp. 639–648.
3. K. Chakrabarty, "Design of System-on-a-Chip Test Access Architectures Using Integer Linear Programming," in *Proc. 18th VLSI Test Symp.*, May 2000, pp. 127–134.
4. K. Chakrabarty, "Design of System-on-a-Chip Test Access Architectures Under Place-and-Route and Power Constraints," in *Proc. 37th Design Automation Conf.*, June 2000, pp. 432–437.
5. K. Chakrabarty, R. Mukherjee, and A. Exnicios, "Synthesis of Transparent Circuits for Hierarchical and System-on-a-Chip Test," in *Proc. IEEE International Conference on VLSI Design*, Jan. 2001, pp. 431–436.
6. I. Ghosh, S. Dey, and N.K. Jha, "A Fast and Low Cost Testing Technique for Core-Based System-Chips," *IEEE Trans. on CAD*, vol. 19, no. 8, pp. 863–877, Aug. 2000.
7. I. Ghosh, N.K. Jha, and S. Dey, "A Low Overhead Design for Testability and Test Generation Technique for Core-Based Systems-on-a-Chip," *IEEE Trans. on CAD*, vol. 18, no. 11, pp. 1661–1676, Nov. 1999.
8. E. Marinissen, R. Arendsen, G. Bos, H. Dingemanse, M. Lousberg, and C. Wouters, "A Structured and Scalable Mechanism for Test Access to Embedded Reusable Cores," in *Proc. 1998 Int. Test Conf.*, Nov. 1998, pp. 284–293.
9. M. Nourani and C.A. Papachristou, "Structural Fault Testing of Embedded Cores Using Pipelining," *Journal of Electronic Testing: Theory and Applications*, vol. 15, pp. 129–144, 1999.
10. T. Ono, K. Wakui, H. Hikima, Y. Nakamura, and M. Yoshida, "Integrated and Automated Design-for-Testability Implementation for Cell-Based ICs," in *Proc. 6th Asian Test Symp.*, Nov. 1997, pp. 122–125.
11. S. Ravi, G. Lakshminarayana, and N.K. Jha, "Testing of Core-Based Systems-on-a-Chip," *IEEE Trans. on CAD*, vol. 20, no. 3, pp. 426–439, March 2001.
12. N.A. Touba and B. Pouya, "Testing Embedded Cores Using Partial Isolation Rings," in *Proc. 15th VLSI Test Symp.*, May 1997, pp. 10–16.
13. P. Varma and S. Bhatia, "A Structured Test Re-Use Methodology for Core-Based System Chips," in *Proc. 1996 Int. Test Conf.*, Oct. 1998, pp. 294–302.
14. L. Whetsel, "An IEEE 1149.1 Based Test Access Architecture for ICs with Embedded Cores," in *Proc. 1997 Int. Test Conf.*, Nov. 1997, pp. 69–78.
15. T. Yoneda and H. Fujiwara, "A DFT Method for Core-Based Systems-on-a-Chip Based on Consecutive Testability," in *Proc. 10th Asian Test Symp.*, Nov. 2001, pp. 193–198.
16. Y. Zorian, E.J. Marinissen, and S. Dey, "Testing Embedded-Core Based System Chips," in *Proc. 1998 Int. Test Conf.*, Oct. 1998, pp. 130–143.

Tomokazu Yoneda received the B.E. degree in information systems engineering from Osaka University, Osaka, Japan, in 1998, and M.E. degree in information science from Nara Institute of Science and Technology, Nara, Japan, in 2001. Presently he is a Ph.D. candidate in Graduate School of Information Science, Nara Institute of Science and Technology. His research interests are VLSI CAD, design for testability and SoC testing.

Hideo Fujiwara received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1969, 1971, and 1974, respectively. He was with Osaka University from 1974 to 1985 and Meiji University from 1985 to 1993, and joined Nara Institute of Science and Technology in 1993. In 1981 he was a Visiting Research Assistant Professor at the University of Waterloo, and in 1984 he was a Visiting Associate Professor at McGill University, Canada. Presently he is a Professor at the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan.

His research interests are logic design, digital systems design and test, VLSI CAD and fault tolerant computing, including high-level/logic synthesis for testability, test synthesis, design for testability, built-in self-test, test pattern generation, parallel processing, and computational complexity. He is the author of Logic Testing and Design for Testability (MIT Press, 1985). He received the IECE Young Engineer Award in 1977, IEEE Computer Society Certificate of Appreciation Award in 1991, 2000 and 2001, Okawa Prize for

Publication in 1994, IEEE Computer Society Meritorious Service Award in 1996, and IEEE Computer Society Outstanding Contribution Award in 2001. He is an advisory member of IEICE Trans. on Information and Systems and an editor of IEEE Trans. on Computers, J. Electronic Testing, J. Circuits, Systems and Computers, J. VLSI Design and others. Dr. Fujiwara is a fellow of the IEEE, a Golden Core member of the IEEE Computer Society, a fellow of the IEICE (the Institute of Electronics, Information and Communication Engineers of Japan) and a member the Information Processing Society of Japan.