A Non-Scan DFT Method for RTL Data Paths To Achieve Complete Fault Efficiency

Toshimitsu MASUZAWA†, Hiroki WADA†,
Kewal K. SALUJA‡, and Hideo FUJIWARA†

†Graduate School of Information Science
Nara Institute of Science and Technology (NAIST)
8916-5 Takayama, Ikoma, Nara 630-0101, JAPAN
e-mail:{masuzawa,hiroki-w,fujiwara}@is.aist-nara.ac.jp

‡Department of Electrical and Computer Engineering
1415 Engineering Drive
University of Wisconsin-Madison
Madison, WI 53706-1691, USA
e-mail:saluja@engr.wisc.edu

Abstract This paper proposes a non-scan DFT method for RTL data paths. It achieves 100% fault efficiency with low hardware overhead and short test application time. Moreover, the proposed DFT algorithms are very efficient.

1 Introduction

Design and test of large VLSI circuits is known to be a difficult problem. Recent design efforts have focused on automating the design process at high-level resulting into design discipline of High-Level Synthesis (HLS). A typical high-level synthesis system decomposes the design into data path and controller. The intermediate output of such a system consists of a register-transfer-level (RTL) netlist of basic components of a data path. These components, or hardware elements, include registers, multiplexors, and operational modules. These hardware elements are often designed or synthesized at gate level and their gate level description is normally available during test generation phase. Hardware elements that constitute the system are interconnected in a complex manner, and as a result the underlying structure contains loops and cycles. The testing problem in such an environment is to test the complete system making use of both the gate level and RTL description.

Classical approaches to test a complex digital system is to rely on the use of scan, in which a complete system is converted to a scan based design and the tests are derived for such a scan-base system. This approach, though effective, is costly in hardware overhead, has long test application time and does not make use of either the RTL description or the inherent data flow features of data path components such as multiplexors and adders. The basic design-for-testability (DFT) and built-in-self-test (BIST) approaches used during high-level synthesis of systems[1, 6, 7, 10] are summarized in a survey paper by Wagner and Dey[11]. These approaches have tried to address the hardware overhead problem by including DFT and BIST based cost functions in the high-level synthesis system. Systems designed using these approaches can reduce the hardware cost but do not overcome the other drawbacks and can not provide 100% fault efficiency[1].

More recently Norwood et. al.[8, 9] have proposed a DFT method, orthogonal scan, that avoids forming an explicit scan path by using data flow paths for scan type operation, thus potentially reducing the hardware cost. The test generation for a system designed with this approach can be carried out in a conventional manner by treating the combinational logic in the complete data path as a single block. As a result this method requires a substantial test generation effort. Further, this approach does not benefit from the available RTL description of the circuit and the hierarchy of the design.

The approaches proposed in Genesis[2, 4, 5] and H-SCAN[3] also make use of the data flow paths. In these approaches test multiplexors are added to achieve the desired scan function, thus the expected area overhead can still be high. Further, in the case of Genesis the test plan generation algorithm is of high complexity, thus making it inefficient for large data path designs. H-SCAN, like orthogonal scan, relies on test generation for the complete data path as a single combinational block, thus requiring substantial test generation effort and is unable to make use of RTL description and the hierarchy of the design. The DFT approach proposed by Ghosh et. al.[4, 5] incorporates

[1] Fault efficiency is the ratio of the number of faults identified as either detectable or redundant to the total number of faults

RTL knowledge into test generation, but the resulting circuit often does not meet the 100% fault efficiency goal.

In this paper we propose a DFT method for modifying a data path such that a combinational test generator can be used to derive tests for the modified data path. Further, the tests are generated for individual modules, thus making the test generation process very efficient and 100% fault efficiency can be achieved. Next we derive test plans that can be used to apply the test vectors derived by the combinational test generator for each module. The DFT scheme proposed by us is a low cost scheme and makes use of data flow paths for testing the circuit. Also the modification algorithm is very efficient, for it is of linear complexity in the number n of hardware elements in the RTL data path. We also give a test plan derivation algorithm of complexity $O(n^2)$.

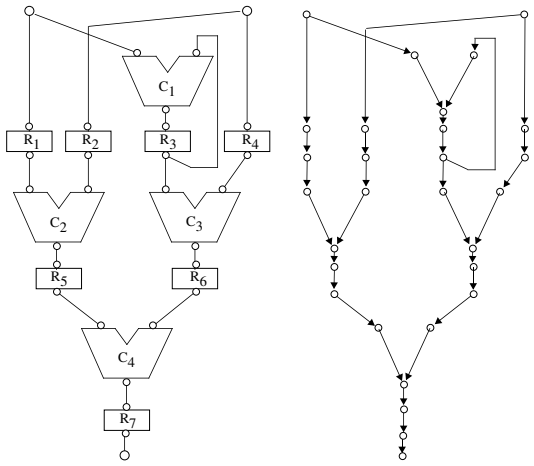
As indicated above we set the following goals for data path designs:

1. 100% fault efficiency
2. low DFT overhead
3. short test application time
4. use of combinational test generation
5. hierarchical testing

The last goal makes use of the RTL description of the data path for generation of test plans. To achieve these goals we place certain restrictions on the structure of the data path. These restrictions are listed in section 2 of this paper. We must add that it is a simple matter to generate data paths satisfying these restrictions for an arbitrary high-level description. In section 3 of this paper we give a definition of a data path for which test generation can be performed using combinational test generation. DFT techniques for modifying a data path and test plan generation for the modified data path are given in sections 4 and 5 respectively. Experimental results for test overhead, test efficiency etc. for a number of circuits are summarized in section 6 and the paper concludes with section 7.

2 Data Paths

A data path consists of *hardware elements* and *lines*. A hardware element is a primary input, a primary output, a register, a multiplexor, or an operational module. Operational modules are combinational circuits and include no register. We introduce *ports* as interface points of each hardware element in a natural fashion: values go into a hardware element through its *input ports*, and come out from a hardware element through its *output ports*. For convenience, we regard a primary input (resp. a primary output) as an output port (resp. an input port). A line connects an output port with an input port. Any number of lines can connect to an output port (i.e., fanout is allowed), but only one line can connect to an input port.



(a) a data path (b) a port digraph

Figure 1: A data path and its port digraph

We introduce a digraph called a *port digraph* to represent direct dependencies between ports. For a data path, a *port digraph* $G = (V, A)$ is defined as follows.

- Node set V consists of all ports in the data path.
- Arc set A is defined as $A = A_1 \cup A_2$. Arc set A_1 contains arcs corresponding to lines, i.e., for an output port u and an input port v, $(u, v) \in A_1$ iff a line connects u to v. Arc set A_2 contains arcs representing dependencies within hardware elements, i.e., for ports u and v, $(u, v) \in A_2$ iff u and v are an input and an output ports of the same element, respectively.

In Fig. 1, we illustrate an example of a data path and its port digraph.

We use some graph terms (e.g., path, cycle etc.) to a port digraph. We also use the graph terms to a data path, since there is a straightforward correspondence between a data path and a port digraph,

A *path* p is denoted by a sequence of ports $p = (u_1, u_2, \ldots u_k)$ such that there is an arc (u_i, u_{i+1}) for each i $(1 \leq i < k)$. We sometimes call p a u_1-u_k path and call u_1 and u_k be a *start node* and a *terminal node* respectively. The *sequential depth* of a path is the number of the registers that appear in the path.

In this paper, we make the following assumptions on a data path.

(A1) Every input port is reachable from a primary input, and every output port is reachable to a primary output.

(A2) Every path between operational modules includes a register in it: Letting u be an output port of an operational module M and v be an input port of an operational module M' (it may be $M = M'$), every u-v path includes a register in it.

(A3) Every cycle includes a register, that is, there is no cycle with the sequential depth of 0.

(A4) For every pair of reconvergence paths (i.e., paths with the same start node and the same terminal node), there is a register that appears in either of the paths but not in the both.

(A5) All input and output ports in a data path have the same bit width.

(A6) An operational module has only one or two input ports and only one output port.

(A7) If M is an operational module with one input port, the function M provides is a bijection between its input port and its output port.

(A8) If M is an operational module with two input ports x and y, M provides a bijection between x and its output port by fixing y to an adequate constant, and provides a similar bijection between y and its output port. That is, there is constants c and c' such that both $f_{y=c}(x)$ and $f_{x=c'}(y)$ are bijections, where $f_{y=c}(x)$ (resp. $f_{x=c'}(y)$) is the function M provides when its input y (resp. x) is fixed to c (resp. c').

A hardware element may have *a control port* to feed control signals in. For example, a register may have a control port for a load/hold signal, a multiplexor has a control port for selection, and an operational module with multiple functions may have a control port for selecting its function. For such an operational module, assumptions (A7) and (A8) requires the existence of the bijection for some function that the module M provides. Control signals usually come from a controller to the data path. In this paper, we assume that every control port is completely controllable. For simplicity, however, we consider explicitly control signals only when dealing with test plans. Some hardware elements may also have *status ports* to report results to the controller. In this paper, we assume that every status port is completely observable. For simplicity, however, we do not explicitly consider the status ports.

3 Strong Testability

Hierarchical test is a promising way for testing very large sequential circuits. In the hierarchical test, test for each hardware element M proceeds as follows.

1. Test vectors are generated for M (a combinational circuit) using a combinational ATPG (Automatic Test Pattern Generation) tool.
2. The test vectors are applied to M through primary inputs: some adequate values are fed to primary inputs at some adequate timing, so that the desired test vectors can be applied to M.
3. The responses of M to the test vectors are propagated to primary outputs and are observed at the primary outputs.

To propagate the test vectors and the responses, some hardware elements must be controlled adequately. For example, to propagate a value through a multiplexor, we must feed a control signal for selecting this input port at some adequate timing. A *test plan* specifies the control signals to propagate the test vectors and the responses. The test plan is denoted by a set of triplets (a, cv, t) where a is a control port, cv is a control signal value and t is time (i.e., clock count). The triplet (a, cv, t) implies value cv is fed to control port a at time t.

During test of a hardware element M, we have to apply several test vectors to M. Generally, the test plans for M may depend on the values of the test vectors. However, this may complicate test process and introduce an undesirable feedback from test vector generation at gate level to DFT process at RTL. Thus, we define *strong testability* of a data path as follows.

Definition 1 A data path is *strongly testable* iff there exists a test plan for each hardware element M that makes it possible to apply any vector to M and to observe any response of M. ■

Strongly testable data paths have the following advantages.

- Fast test vector generation: Test vector generation time is short since a combinational ATPG tool can be used. Moreover, it is shorter than that of full scan design, because the ATPG tool is applied to every hardware element separately but not to a whole circuit at once.
- Fast test plan generation: Test plan generation time is short since test plans are generated at RTL (not at gate level).
- 100% fault efficiency: 100% fault efficiency is achieved for a whole circuit, since each hardware element M is a combinational circuit of small size and strong testability guarantees complete controllability and complete observability of M.

4 DFT for strong testability

In this section, we propose a DFT algorithm for making given data paths strongly testable.

4.1 Overview

In our DFT method, we add no special path for propagating test vectors and responses. Test vectors and responses are propagated along existing paths.

Consider test application for a hardware element M with two input ports x and y. To test M, some value specified by a test vector should be fed into input port x. We propagate the value along a path p from a primary input a to x. If some operational module C appears in p, the output value of C results from some operation on its input values. But assumption (A7) guarantees that the output of C is completely controllable by its input if C has only one input port. Assumption (A8) also guarantees for C with two input ports that the output of C is completely controllable by one input if the other input is fixed to some constant. To feed such a constant, in our DFT method, a *mask element* is added to the input port. We add the mask element to every operational module C in path p so that its input port appearing in p can control its output port. Thus, using the mask elements, primary input a (the start node of p) can completely control input port x (the terminal node of p).

However, we cannot achieve the strong testability only by adding the mask elements. The mask elements guarantee controllability of a single path. To test an operational module with two input ports, some specified values must be fed in these ports simultaneously. If the paths for propagating the values are not disjoint, a *timing conflict* may occur: two different values are required to be fed in a primary input at the same time. To resolve timing conflicts, in our DFT method, some registers are augmented with *hold function*.

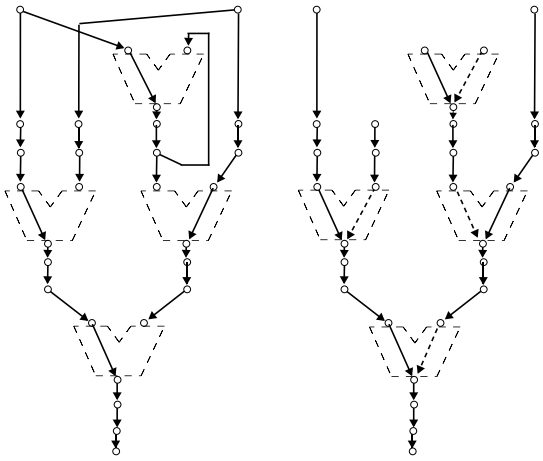
The goal of the DFT is to make a given data path strongly testable with the minimum hardware overhead. Computation time for the DFT and for test plan generation is also important. We propose a heuristic algorithm for the DFT in this section, and a test plan generation algorithm in the next section. Computation time of these algorithms is $O(n)$ and $O(n^2)$ respectively, where n is the number of hardware elements in a given data path.

The heuristic algorithm for the DFT proceeds in the following three stages.

(Stage 1) Construct a *control forest*: To determine paths to propagate test vectors, we construct a control forest. The control forest is a spanning out-forest where primary inputs are roots, and its paths are used to propagate test vectors.

(Stage 2) Construct an *observation forest*: To determine paths to propagate responses, we construct an observation forest. The observation forest is a spanning in-forest where primary outputs are roots, and its paths are used to propagate responses.

(Stage 3) Add DFT elements: To make the data path strongly testable, we add some DFT elements (mask elements and hold function) to the data path.



(a) a control forest (b) an observation forest

Figure 2: A control and an observation forests

4.2 Control forest (stage 1)

Some values specified by test vectors are propagated from a primary input to an input port of a hardware element. The path used for the propagation is called a *control path* of the port. As the control path, we choose a path with the minimum sequential depth among all the paths from primary inputs to the port. If there exist several paths to the port with the minimum sequential depth, we choose a path among them so that the chosen paths for all ports in the data path form an out-forest. We call the out-forest a *control forest*. In Fig. 2 (a), we show a control forest of the data path in Fig. 1.

For a given data path, its control forest is constructed as follows.

1. Construct a weighted digraph $G_1 = (V, A, w_1)$ from the port digraph $G = (V, A)$, where w_1 is a weight function that assigns the following weight to each arc.
   - $w_1(u, v) = 1$ if u and v are the input and the output ports of a register, and
   - $w_1(u, v) = 0$ otherwise.
2. Construct a shortest spanning out-forest T_1 of G_1 where all primary inputs are roots of the forest. Out-forest T_1 is the control forest. For the third stage, we find the sequential depth $depth(u)$ and the start node (a primary input) $ctrlPI(u)$ of the control path of u for each port u.

Consider any hardware element M with two input ports. In the control forest, one of its input ports has an outgoing arc, and the other is a leaf (with no outgoing arc). We call the input port with an outgoing arc the *first input port* of M, and call the other the *second input port* of M.

We show how to propagate a value along a control path. Let p be a control path from a primary input a to a port u, and let $M_1, M_2, \ldots M_k$ be all the hardware

elements that appear in p in this order. To get a desired value at u, we control these elements and feed an adequate value to a as follows.

(a) Case that M_i is a register: control M_i to load a value.

(b) Case that M_i is a multiplexor: control M_i to select its first input port (i.e., the input port on p).

(c) Case that M_i is an operational module with one input port: no special control is necessary for M_i. Let f_i be the function that M provides.

(d) Case that M_i is an operational module with two input ports: let x and y be the first and the second input ports of M_i. (Note that x is on p.) From the DFT algorithm described later, a mask element is added to y. We control M_i to activate the mask element so that M_i can provide a bijection f_i between x and its output port.

(e) Primary input a: Let d be the sequential depth of p, and $M_{c_1}, M_{c_2}, \ldots M_{c_m}$ be all the operational modules that appear in p in this order. We can get any value val at u (terminal node of p) after d clocks, if we feed the value $f_{c_1}^{-1}(f_{c_2}^{-1}(\ldots(f_{c_m}^{-1}(val))\ldots))$ to primary input a.

4.3 Observation forest (stage 2)

Responses of a hardware element are propagated to a primary output. The path used for the propagation is called an *observation path* of the port. In the observation path, we utilize control paths as much as possible, since any value can be propagated along the control paths. But, in most cases, there exists no control path from the output port of a hardware element to a primary output. In such a case, we construct an observation path by connecting the minimum number of control paths. If there exist several control paths each consisting of the minimum number of control paths, we choose a path among them so that the chosen paths for all ports in the data path form an in-forest. We call the in-forest an *observation forest*. We call the arcs that appear on the observation forest but not in the control forest *junction arcs*. In Fig. 2 (b), we show an observation forest of the data path in Fig. 1 and the junction arcs (broken arcs in the figure).

For a given data path, its observation forest is constructed as follows.

1. Construct a weighted digraph $G_2 = (V, A, w_2)$ from the port digraph $G = (V, A)$ and the control forest T_1, where w_2 is the following weight function.
   - $w_2(u, v) = 0$ if (u, v) is an arc of T_1.
   - $w_2(u, v) = 1$ otherwise.

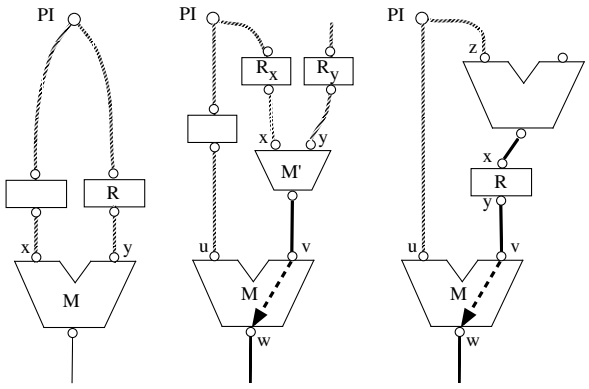


Figure 3: Addition of hold function

2. Construct a shortest spanning in-forest T_2 of G_2 where all primary outputs are roots of the forest. In-forest T_2 is the observation forest.

We show how to propagate a value along an observation path. Let p be an observation path from a port u to a primary output a, and let $M_1, M_2, \ldots M_k$ be all the hardware elements that appear in p in this order. To observe a value of u, we control these elements. Registers, multiplexors, and operational modules with one input port are controlled in a similar way to a control path.

(d) Case that M_i is an operational module with two input ports: let x and y be the two input ports and assume that x is on p. Let z be the output port of M_i.

- If x is the first input port, then the DFT algorithm described later adds a mask element to y. We control M_i to activate the mask element so that M_i can provide a bijection f_i between x and z.
- If x is the second input port, then y has no mask element. We feed some constant to y using the control path of y so that M_i can provide a bijection f_i between x and z.

(e) Primary output a: Let d be the sequential depth of p, and $M_{c_1}, M_{c_2}, \ldots M_{c_m}$ be all the operational modules that appear in p in this order. If value val appears at u (start node of p), we can observe $f_{c_m}(f_{c_{m-1}}(\ldots(f_{c_1}(val))\ldots))$ at primary output a after d clocks.

4.4 DFT elements (stage 3)

The DFT elements are added as follows so that any value can be propagated along the control paths and the observation paths.

Mask elements: For each operational module M with two input ports, a mask element is added to its second input port y so that M provides a bijection

between its first input port x and its output port z. The mask element feeds a constant c to y such that $z = f_{y=c}(x)$ is a bijection.

Hold function: Registers are augmented with hold function in the following three cases H1, H2 and H3. In case H1, we consider the possibility of timing conflicts in propagation of test vectors.

H1: Let M be any hardware element with two input ports, and x and y be the first and the second input ports of M, respectively. If M satisfies $depth(x) = depth(y)$ and $ctrlPI(x) = ctrlPI(y)$, augment register R with hold function where R is the nearest register to y in the control path of y (Fig. 3 (a)).

In cases H2 and H3, we consider the possibility of timing conflicts in propagation of responses. As described in the previous subsection, some constant should be fed to the first input port u, along the control path, when a response passes a junction arc (v, w), where u, v and w are ports of an operational module. This may cause a timing conflict only if $depth(u) = depth(v)$. Thus, in the following, we consider a forest T_2' obtained from the observation forest T_2 by removing all junction arcs (v, w) such that $depth(v) \neq depth(w)$. For each junction arc (v, w) in T_2' corresponding to an operational module M, add hold function as follows.

H2: Let M' be any multiplexor, and x and y be its input ports. If M' satisfies the following two conditions, then for registers R_x and R_y are augmented with hold function where R_x (resp. R_y) is the nearest register to x (resp. y) in the control path of x (resp. y) (Fig. 3 (b)).

(a) Port v is reachable from x in T_2' by a path of sequential depth 0, and
(b) $ctrlPI(u) = ctrlPI(x)$.

H3: If register R satisfies the following conditions, then augment R with hold function (Fig. 3 (c)). In the following conditions, x (resp. y) denotes the input port (resp. the output port) of R.

(a) Port v is reachable from y in T_2' by a path of sequential depth 0, and
(b) there is a port z reachable to x in T_2' such that $ctrlPI(u) = ctrlPI(z)$.

Figure 4 shows an DFT result for the data path in Fig. 1.

4.5 Time complexity

Since the arc weights in G_1 and G_2 are 0 or 1, shortest spanning forest construction in the first and the second stages can be done in $O(n)$ time where n is the number of hardware elements in the data path. Notice that the numbers of nodes and arcs in the port digraph are $O(n)$ since we consider only hardware elements with one or two input ports. Thus, the first and the second stages can be executed in $O(n)$ time.

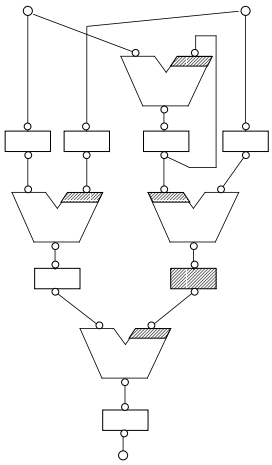


Figure 4: An DFT result for the data path in Fig. 1

Addition of mask elements and addition of hold function by H1 can be done for the whole data path in $O(n)$ time by traversing the control forest. In addition of hold function by H2, the conditions (a) and (b) can be checked for the whole data path in $O(n)$ time by traversing the observation forest. The registers to be augmented with hold functions can be determined in $O(n)$ time by traversing the control forest. Similarly, addition of hold function by H3 can be done in $O(n)$ time if condition (b) can be checked in $O(n)$ time. This can be done as follows. By traversing T_2' in a depth-first fashion, we find the post ordering number $post(u)$ and find a set $desc(u)$ of descendants of u (i.e., ports reachable to u) for each port u. Notice that a single post ordering number can express $desc(u)$, since descendants of u appear consecutively and immediately before u in the post ordering. Thus, $post(u)$ and $desc(u)$ can be found in $O(n)$ time for all ports. We prepare an array A of size m (where m is the number of primary inputs), and traverse T_2' in the post ordering. Assume the initial value of each component of A is 0. When we visit each port u in the traverse, we execute $A[a] := post(u)$ where $a = ctrlPI(u)$. When we visit a port x in the traverse, the following property holds: there is a port z reachable to x in T_2' such that $ctrlPI(z) = a$ iff $A[a]$ is a descendant of x. (Correctness of the property is intuitively understandable from the fact that descendants of u appear consecutively and immediately before u in the post ordering.) Thus, when we visit an input port of a register, condition (b) of H3 can be checked by checking only one component of A. Notice that, from assumption 2, there is only one computational module that is reachable in T_2' from R by a path of sequential depth 0.

Thus, the whole time complexity of the DFT algorithm is $O(n)$.

5 Test plan generation

In this section, we propose a test plan generation algorithm for a data path modified by the DFT algorithm in the previous section.

5.1 Primary and preliminary tests

To test a hardware element M, test vectors for M are propagated to the input ports along the control paths of the ports. If M has two input ports and is included in a cycle in the data path, then the control path of its input port, say x, may go through M itself. For example, in Fig. 2 (a), the control path of the second input port of C_1 goes through C_1 itself. In such a case, a fault of M may affect the value propagated through M and prevent the desired values from reaching x. Thus, we first observe the value propagated through M using the observation path of the output port of M. If the observed value is not the expected one, a fault is detected. We call this test a *preliminary test*. Notice that we execute the preliminary test for each test vector of M, since affect of a fault depends on the vector value. If no fault is detected in the preliminary test, the test vector is applied to the two input ports of M and the response of M is observed. We call this test a *primary test*.

We have to find test plans for preliminary tests as well as for primary tests. In the following subsections, however, we consider only test plan generation for primary tests. Test plans for preliminary tests can be generated similarly.

5.2 Timing conflict

To test a hardware element M, test vectors are applied to M using the control paths. However, if the control paths of the two input ports of M have the same start node (a primary input) and have the same sequential depth, then a conflict occurs at the primary input: these two control paths require to feed different values to the primary input at the same timing.

Such timing conflicts also occur in propagation of responses of M. If a junction arc appears in the observation path and its corresponding element is an operational module, some adequate constant must be fed to the first input port using the control path. We call the control path a *support path* of M. A support path may cause a timing conflict with the control paths and the other support paths of M.

To resolve the timing conflicts, we can use hold function of registers and make difference on the timing.

5.3 Test plan generation algorithm

Consider a primary test of an arbitrary hardware element M. This subsection presents an algorithm that generates a test plan for M when the control paths, the observation path and the support paths for M are given. To generate the test plan, the algorithm detects and resolves timing conflicts.

A test plan specifies control signals for propagating a test vector and its response. Recall that a test plan consists of triplets and each triplet specifies a control signal for one element at one clock. We can make an example where $\Omega(n)$ of hardware elements must be controlled during $\Omega(n)$ clocks (where n is the number of hardware elements in the data path). It follows that the size of the test plan is $\Omega(n^2)$ and it requires $\Omega(n^2)$ time only to output the plan. (Thus, it requires $\Omega(n^3)$ time to generate test plans for *all* hardware elements.)

The algorithm proposed in this subsection runs in $O(n)$ time and generates a test plan in a different format. (Thus, it requires $O(n^2)$ time to generate test plans for *all* hardware elements.) It first determines a *default control signal* for each hardware element, and generates triplets only when it requires a control signal other than the default.

The algorithm consists of three stages. In the first stage, it determines a default control signal. It generates triplets for propagating vectors in the second stage, and triplets for propagating responses in the third stage.

(Stage 1) In testing M, we use only the control, the observation and the support paths for M. Most arcs in these paths are arcs in the control forest. Only the exceptions are junction arcs in the observation path. Thus, it is natural to consider, as the defaults, the control signals that activate the control forest:

- The default for a register is to load a value.
- The default for a multiplexor is to select the first input port (i.e., the input port with an outgoing arc in the control forest).
- The default for an operational module with two input ports is to activate its mask element.

(Stage 2)

(a) Case that M has only one input port: Let u be the input port, $a = ctrlPI(u)$ and $d = depth(u)$.

Generate a triplet
(a, "TV", 1)
to specify that a test vector is fed to a primary input a at clock 1[2].

In the third stage, we use three variables PI, $time$ and p_depth to keep paths that may cause a timing conflict with a support path. The variable PI stores the start nodes (primary inputs) of the paths, $time$ stores the latest time at which values are fed to the primary inputs, and p_depth

[2] The actual value fed to a is not a raw test vector. It should be adequately modified with considering operational modules in the control path. But for simplicity, we use similar expression in what follows.

stores the sequential depth of the paths. We keep information of only the shortest control paths because of the following reason. Consider a support path for propagating a constant to an input port x of a hardware element. The support path is the control path of x, that is, a shortest path to x. Thus, the support path causes a timing conflict only with a shortest path.

Set the variables as follows.
$PI := \{a\}$
$time := 1$
$p_depth := d$

(b) Case that M has two input ports: Let u_1 and u_2 be the first and the second input ports of M respectively. Let $a_i = ctrlPI(u_i)$ and $d_i = depth(u_i)$ for each i $(i = 1, 2)$. Notice that $d_1 \leq d_2$ holds since u_1 is the first input port.

A timing conflict occurs iff $a_1 = a_2$ and $d_1 = d_2$.

If no timing conflict occurs, then generate triplets
$(a_1$, "TV_1", $d_2 - d_1 + 1)$ and
$(a_2$, "TV_2", $1)$,
where "TV_1" and "TV_2" represent respective test values to u_1 and u_2. Set the variables as follows.

$PI := \{a_1, a_2\}$ if $d_1 = d_2$
$\{a_1\}$ otherwise $(d_1 < d_2)$
$time := d_2 - d_1 + 1$
$p_depth := d_1$

If the timing conflict occurs, we resolve it as follows. Let R be the register nearest to u_2 in the control path of u_2. To resolve the timing conflict, the test value to u_2 in a test vector of M is first propagated and is held at R during one clock. From the DFT algorithm (case H1), R has hold function. Moreover, from assumption (A4), R does not appear in the control path of u_1. Thus, the test value for u_2 held at R does not interfere with the propagation for u_1. That is, generate triplets
$(a_1$, "TV_2", $1)$,
$(a_1$, "TV_1", $2)$, and
$(R$, "H", $d_1 + 1)$.
The last triplet specifies that R holds its value at clock $d_1 + 1$.

Moreover, set the variables as follows.
$PI := \{a_1\}$
$time := 2$
$p_depth := d_1$

(Stage 3) Let p be the observation path of M. The observation path p can be expressed by $p = p_1, p_2, \ldots p_k$ where p_i $(1 \leq i \leq k)$ is a maximal partial path that contains no junction arc. In other words, we partition p into paths $p_1, p_2, \ldots p_k$ by removing the junction arcs. Thus, the end node of p_i $(1 \leq i < k)$ is an input port of a hardware element, say M_i, that is a multiplexor or an operational module. If M_i is an operational module, p_i meets a support path at M_i. Let s_i be the support path.

The third stage consists of $k-1$ iterations. In the i^{th} iteration $(1 \leq i \leq k-1)$, a triplet for hardware element M_i is generated and the timing conflict caused by s_i is resolved if necessary. In what follows, we describe the i^{th} iteration. Let $d = p_depth + d_i$ where d_i is the sequential depth of p_i.

(a) Case that M_i is a multiplexor: Generate a triplet
$(M_i$, "x", $time + d)$
where "x" is the input port of M_i that p_i connects to. The triplet specifies that M_i selects its input port "x" at clock $time + d$.

Moreover, update variable p_depth as follows.
$p_depth := p_depth + d_i$
(Variables PI and $time$ are not updated.)

(b) Case that M_i is an operational module: Let b_i be the start node (a primary input) and d' be the sequential depth of s_i. Notice that $d' \leq d$ holds since the support path is a shortest path to M_i.

A timing conflict occurs iff $d' = d$ and $b_i \in PI$ hold.

If no timing conflict occurs, then generate triplets
$(M_i$, "normal", $time + d)$ and
$(b_i$, "const", $time + d - d')$.
The first triplet specifies that M_i operates normally (i.e., its mask element is not activated) at clock $time + d$. The second one specifies that the constant for a bijection of M_i is fed to a primary input b_i at clock $time + d - d'$.

Moreover, set the variables as follows.

$PI := PI \cup \{b_i\}$ if $d' = d$
$\{b_i\}$ otherwise $(d' < d)$
$time := time + d - d'$
$p_depth := d'$

If the timing conflict occurs, we resolve it by holding the value along p_i during one clock at the register nearest to M_i. Precisely, the following is executed.

- Case that there is a register in p_i: Let R be the register nearest to M_i in p_i. From the DFT algorithm (case H3), R is augmented with hold function.

  Generate triplets
  $(R$, "H", $time + d)$ and
  $(b_i$, "$const$", $time + d - d' + 1)$.

  Moreover, update the variables as follows.
  $PI := \{b_i\}$
  $time := time + d - d' + 1$
  $p_depth := d'$

- Case that there is no register in p_i: If $i \geq 2$ or M (the hardware element under test) is an operational module, there is a register in p_i from assumption (A2). Thus, if there is no register in p_i, then $i = 1$ and M is a multiplexor. In this case, we use the nearest register in each of the control paths of M to hold the values. Let R_1 and R_2 be these registers. From the DFT algorithm (case H2), these registers have hold function.

  Thus, generate triplets
  (R_1, "H", $time + d$),
  (R_2, "H", $time + d$) and
  (b_i, "$const$", $time + d - d' + 1$).

  Moreover, update the variables as follows.
  $PI := \{b_i\}$
  $time := time + d - d' + 1$
  $p_depth := d'$

5.4 Time complexity

We show a test plan for a single hardware element is generated in $O(n)$ time where n is the number of hardware elements. The first and the second stages can be clearly executed in $O(n)$ time. In the third stage, we have only to estimate time required to search the nearest registers and to manage variable PI since the other part can be clearly executed in $O(n)$ time. The nearest registers can be found by traversing the control paths and the observation path. Since the searched parts in distinct iterations are disjoint, the traverse requires $O(n)$ time in the whole algorithm. Concerning PI, three kinds of operations, *insert*, *member* and *make_empty*, are used $O(n)$ times. In order to execute all operations in $O(n)$ time, we implement PI using both a bit vector and a linked list. The key is that the linked list is utilized to make the bit vector empty without accessing components with *false* values. Thus, it takes $O(n)$ time to generate a test plan for each hardware element.

6 Experimental results

In this section, we present experimental results of our DFT method. We applied our method to three example data paths: GCD (a greatest common divisor), 4th IIR (a 4th order IIR cascade filter), and Paulin (a popularly used example). Table 1 shows the characteristics of these data paths, where *#PIs* and *#POs* denote the numbers of primary inputs and primary outputs respectively, and *#registers*, *#MUXs* and *#op. mod.* respectively denote the numbers of registers, multiplexors and operational modules.

Tables 2, 3 and 4 show the experimental results. To evaluate the effectiveness of our method, our method is compared with (traditional) full scan designs. In the experiments, gate-level implementation of each of the three data paths is generated using a logic synthesis tool AutoLogicII (Mentor Graphics Co.), and test vectors are generated using a combinational ATPG tool TestGen (Sunrise Test System Inc.) on a SUN SPARC Station-20 (SuperSPARC 75MHz).

Table 1: Circuit characteristics

	GCD	4th IIR	Paulin
#PIs	2	1	2
#POs	1	1	2
#registers	3	12	7
#MUXs	3	3	8
#op. mod.	5	5	7

Table 2: Test generation time (sec.)

bit-	GCD		4th IIR		Paulin	
width	*ours*	*F.S.*	*ours*	*F.S.*	*ours*	*F.S.*
8	1.1	0.5	0.8	0.7	1.2	1.4
16	1.6	1.7	1.3	2.2	2.8	4.5
32	5.8	7.5	11.5	33.7	11.3	18.2

For all the circuits, we achieved 100% fault efficiency in both our method and the full scan method. Table 2 shows test generation time. Column *bitwidth* denotes the bitwidth of each line: we made experiments for three cases. Columns *ours* and *F.S.* show test generation time for designs of our method and for full scan designs respectively. The test generation time for our method includes running time of our DFT algorithm (in Section 4) in addition to the total test generation time for all hardware elements. The time for the DFT was 0.2 seconds for each of three data paths. Notice that the running time for the DFT algorithm does not depend on bitwidth since the algorithm works on RTL data paths. From the results, we can recognize the advantage of hierarchical test. While ATPG in full scan designs is executed to the whole data paths at once, ATPG in our method is executed to each hardware element separately. Our method gains the greater advantage as the size of data paths becomes larger.

Table 3 shows hardware overhead of the DFT methods. Column *orig.* shows the numbers of gates in circuits generated from the original data paths. Column *our method* and *F.S.* show the numbers of gates in circuits generated from the data paths after modification by the DFT methods. In all examples, our DFT method has lower hardware overhead (*ovhd.*) than the full scan method. We want to mention the reason why hardware overhead of our method is relatively high in 4th IIR. Data path 4th IIR includes three operational modules each of which has a single input port and executes multiplication with some constant. Since the multiplications are not bijections, we modified the operational elements to satisfy assumption (A7).[3] The

[3] We made modification independently of the constant values. We can attain less hardware overhead if we consider the constant values.

Table 3: Hardware overheads

bit-width	*orig.*	*our method*		*full scan*	
	#gates	*#gates*	*ovhd.* (%)	*#gates*	*ovhd.* (%)
GCD					
8	259	267	3.1	283	9.3
16	529	545	3.0	677	28.0
32	873	905	3.7	1161	33.0
4th IIR					
8	350	510	45.7	636	81.7
16	1192	1512	26.8	1768	48.3
32	4316	4641	7.5	5468	26.7
Paulin					
8	1152	1184	2.8	1240	7.6
16	3848	3912	1.7	4042	5.0
32	13848	13976	0.9	14200	2.5

Table 4: Test application time (clocks)

bit-width	GCD		4th IIR		Paulin	
	ours	F.S.	ours	F.S.	ours	F.S.
8	254	1299	852	2715	1137	2792
16	470	5732	1392	8105	1503	8587
32	760	22794	2936	71224	2512	24524

hardware overhead includes the overhead for this modification.

Table 4 shows test application time for the data paths. We estimate the time by the number of clocks required to apply all the generated test vectors and to observe the responses. In our DFT method, the test application time is measured by

$$\sum_{\text{all hardware elements } M_i} (\#tv_i \times tp_i)$$

where $\#tv_i$ is the number of test vectors generated for M_i and tp_i is the number of clocks required for a primary and a preliminary (if necessary) test plans of M_i. In the full scan method, it is measured by

$$\#tv \times (\#FFs + 1) + \#FFs$$

where $\#tv$ is the the number of generated test vectors and $\#FFs$ is the number of flip-flops. Notice that $\#FFs$ represents the length of a scan chain in full scan designs. The experimental results show that test application time is drastically improved in our DFT method.

7 Conclusions

In this paper, we proposed a new DFT method for RTL data paths to achieve 100% fault efficiency. The DFT method is based on hierarchical test and usage of a combinational ATPG tool. First, we defined strong testability of data paths that guarantees applicability of the above approach. Next, we presented a heuristic DFT algorithm that modifies data paths to strongly testable ones with low hardware overhead. We also presented an algorithm that generates test plans for the modified data paths. These algorithms are very efficient: the DFT algorithm runs in $O(n)$ time, and the test plan generation algorithm generates test plans for each hardware element in $O(n)$ time, where n is the number of hardware elements in the data path. Experimental results show that our DFT method achieves 100% fault efficiency with low hardware overhead, and improves test application time drastically compared with the traditional full scan method. One of our future works is to relax the restrictions placed on the structure of the data paths.

Acknowledgment The authors would like to thank Dr. Tomoo Inoue and Dr. Michiko Inoue of Nara Institute of Science and Technology for their helpful discussion. This work was supported in part by Semiconductor Technology Academic Research Center (STARC) under the Research Project and in part by the Ministry of Education, Science, Sports and Culture, Japan under the Grant-in-Aid for Scientific Research B(2) (No.09480054).

References

[1] L. Avra. Allocation and assignment in high-level synthesis for self-testable data paths. In *Proc. of the IEEE Int. Test Conf.*, pages 463–472, 1991.

[2] S. Bhatia and N. K. Jha. Genesis: A behavioral synthesis system for hierarchical testability. In *Proc. the European Design and Test Conference*, pages 272 – 276, 1994.

[3] S. Bhattacharya and S. Dey. H-SCAN: A high-level alternative to full-scan testing with reduced area and test application time. In *Proc. the IEEE VLSI Test Sympo.*, pages 74–80, 1996.

[4] I. Ghosh, A. Raghunathan, and N. K. Jha. Design for hierarchical testability of RTL circuits obtained by behavioral synthesis. In *Proc. IEEE Int. Conf. on Computer Design*, pages 173 – 179, 1995.

[5] I. Ghosh, A. Raghunathan, and N. K. Jha. A design for testability technique for RTL circuits using control/data flow extraction. In *Proc. the ACM/IEEE Int. Conf. on Computer-Aided-Design*, 1996.

[6] T.-C. Lee, W. H. Wolf, and N. K. Jha. Behavioral synthesis for easy testability in data path scheduling. In *Proc. the ACM/IEEE Int. Conf. on Computer-Aided-Design*, pages 616–619, 1992.

[7] A. Mujumdar, R. Jain, and K. Saluja. Incorporating testability considerations in high-level synthesis. *Journal of Electronic Testing: Theory and Applications*, 5(1):43–55, February 1994.

[8] R. B. Norwood and E. J. McCluskey. Orthogonal scan: Low overhead scan for data paths. In *Proc. of the IEEE Int. Test Conf.*, pages 659 – 668, 1996.

[9] R. B. Norwood and E. J. McCluskey. High-level synthesis for orthogonal scan. In *Proc. the IEEE VLSI Test Sympo.*, pages 370 – 375, 1997.

[10] C. A. Papachristou, S. Chiu, and H. Harmanami. A data path synthesis method for self-testable designs. In *Proc. the ACM/IEEE Design Automation Conf.*, pages 378–384, 1991.

[11] K. D. Wagner and S. Dey. High-level synthesis for testability: A survey and perspective. In *Proc. the ACM/IEEE Design Automation Conf.*, pages 131–138, 1996.