Interconnect Delay Testing of Designs on Programmable Logic Devices

Mehdi Baradaran Tahoori
Dept of ECE
Northeastern University
Boston, MA 02115
mtahoori@ece.neu.edu

Subhasish Mitra
Intel Corporation
Sacramento, CA
subhasish.mitra@intel.com

Abstract

Very thorough interconnect delay testing technique for designs implemented on programmable logic devices, such as FPGAs, is presented (application-dependent test). The presented technique achieves 1) 100% robust path delay coverage on all the paths in the design, 2) 100% transition fault coverage, and 3) 100% TARO coverage, transition to all reachable primary outputs. The required number of test configurations is two or four depending on the structure of the design. An algorithmic approach to generate the test vectors and configurations is presented.

1. Introduction

Field Programmable Gate Arrays (FPGAs) are increasingly used for several applications. The programmability of FPGAs helps in achieving a short design cycle and low development costs, as well as a reduced time-to-market. FPGAs are widely used in various applications such as networking, digital signal processing, high performance computing, rapid prototyping and hardware emulation.

The architecture of SRAM-based FPGAs consists of two-dimensional arrays of logic blocks and programmable interconnection network, surrounded by programmable input/output blocks on the periphery. In SRAM-based FPGAs different designs can be mapped into the same device by loading different configuration bitstreams.

A circuit has a *delay fault* if there is a timing failure that makes the circuit fail to work at the designed speed but to be functional at a slower (or faster) speed. Transition and path delay fault models are commonly used for modeling and test generation for delay defects.

In FPGAs, most of the device area, more than 80%, is dedicated to the interconnection network as programmable switches and wiring segments in more than eight metal layers [Xilinx 03][Toutounchi 02]. As a result, FPGAs are vulnerable to defects in interconnects, such as resistive bridges and opens which lead to delay faults. Increasing the clock frequency of designs mapped on FPGAs increases the sensitivity to delay faults.

There is a trend in the FPGA industry for application-specific flow [Xilinx EasyPath]. By testing the resources of an FPGA with respect to a specific design to be implemented on it, some defective chips, which are good only for that particular design but do not have the programmability of a typical FPGA, can be sold to customers. This strategy can be used for high-volume fixed designs.

In this paper, testing of delay faults in the interconnects of the designs mapped into FPGAs is addressed. The presented technique guarantees detection of all path delay faults irrespective of presence of delay faults on other paths (robust path-delay test). Moreover, the fault effect is propagated to all reachable primary outputs (TARO test).

A linear time algorithm, in terms of the number of sequential paths in the design, is presented to decompose the design and generate test configurations.

Unlike some techniques presented in the previous work which are comparison-based and hence, cannot detect timing defects due to global process shifts and common-mode failures [Abramovici 03][Chmelar 03] [Krasniewski 01][Harris 01], the presented solution in this paper does not suffer from these limitations.

The rest of this paper is organized as follows. In Sec.2, the review, including delay fault models, the FPGA architecture, and previous work, is presented. In Sec. 3, the properties of the AND and OR networks in detecting delay faults are presented. The proposed test vectors and configurations are presented in Sec. 4. In Sec. 5, the algorithm is described. In Sec. 6, the problem of applying vector pairs at appropriate clock cycles is addressed. In Sec. 7, results for some benchmark designs are presented. In Sec. 8, the problems of false paths and testing only critical paths are addressed. Also, the application of the presented technique in application-independent (manufacturing) test is also discussed. Finally, Sec. 9 concludes the paper.

2. Background

2.1. Delay Fault Models

In the *path delay fault* model it is assumed that the propagation delay of at least one sensitizable path (from a

0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

primary input or a bistable output to a primary output or a bistable input) of the circuit exceeds the specified cycle time. However, the number of faults (paths) can grow exponentially with the number of gates. Hence, in practice only a certain fraction of longest paths (*critical* paths) and paths with propagation delays within 5-10% of critical paths are tested. However, this approach may not be very effective in detecting defective chips with timing failures, as illustrated by the Murphy test chip experiment results [Ma 95, Tseng 01].

Path delay test is categorized into *robust* and *non-robust* tests. A robust path delay test guarantees to produce an incorrect value at the destination if the delay of the path under test exceeds the clock interval, irrespective of the delay distribution in the circuit [Bushnell 00]. Note that robust path delay test generation is more complicated than non-robust test generation and also not possible for some paths in the circuit.

As an example, consider a NAND gate with the inputs a and b, and the output z, such that a and z are on the path under test (b is called *side-fan-in*). $a(1,0)$ and $b(1,1)$ is a robust test for rising transition on z. Both $\{a(0,1), b(1,1)\}$ and $\{a(0,1), b(0,1)\}$ are robust for falling transition on z, as shown in Fig. 1. However, as shown in [Chen 97], $\{a(0,1), b(0,1)\}$ (Fig. 1(a)) is higher quality robust test which captures worst case conditions based on CMOS implementation of logic networks. In this case the output makes a rising transition only if the rising transitions on both input paths are propagated. The advantage of $\{a(0,1), b(0,1)\}$ transition test over $\{a(0,1), b(1,1)\}$ is that the latter only considers timing defects on the longer path but the former detect any timing failure on any path which prevents the output transition within the clock period (timing fault).

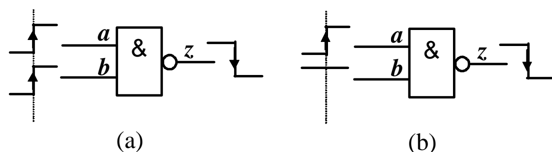

Figure 1 Robust falling transition tests on the output of a NAND gate

In transition fault model it is assumed that a localized timing failure is large enough such that the delay of all paths through some gate to observable outputs exceed the clock interval. For two kinds of transitions, rising or falling, there are two types of transition faults: *slow-to-rise* and *slow-to-fall*. There are two transition faults, slow-to-rise and slow-to-fall, associated with each gate input or output of a logic network.

Recently, the concept of TARO tests has been introduced [Tseng 01]. TARO stands for Transition to All Reachable Outputs. In a TARO test each transition fault is detected using input combinations such that the fault effect is propagated to all outputs. One problem with the transition fault model is that it does not pay attention to the paths along which the fault effects are propagated to the primary outputs. The concept of TARO test partially compensates for the lack of selection of propagation paths in the transition fault model.

Delay testing is not only useful of identifying fault-free (good) chips from failing (bad) ones, it is widely used for speed binning, identifying the maximum clock frequency at which the device functions. This is used for FPGAs and microprocessors. Hence, a delay testing scheme which precisely characterizes maximum speed of the device is very profitable. A delay test scheme with 100% TARO coverage is a good candidate for speed binning since activates and observes all paths to primary outputs.

2.2. FPGA Model

The FPGA architecture consists of an array of *programmable logic blocks* (PLB) and programmable interconnection network, surrounded by programmable input/output blocks on the periphery.

The interconnection network consists of wiring channels in the vertical and horizontal directions which span the entire FPGA. The selective connectivity among wiring channels, logic resources, and IO blocks is provided by programmable switches and/or multiplexers. In SRAM-based FPGAs, the state of these switches and multiplexers are stored in SRAM memory cells. Hence these resources are reprogrammable.

There are two basic FPGA interconnection architectures. In the *segmented routing* scheme, the interconnection network consists of a two-dimensional array of identical switch matrices, and an abundance of line segments, with a variety of length, size and speed [Xilinx 03]. Switch matrices, which consist of programmable switches, provide selective and configurable connectivity among the line segments. In *multiplexer-based* architectures, the programmable switches are replaced by programmable multiplexers that provide a more deterministic routing structure [Altera 03].

Interconnect resources can be categorized into *inter-PLB* and *intra-PLB* resources. The interconnect inside the PLB is called intra-PLB interconnect whereas the routing resources connecting PLBs are called inter-PLB interconnect.

FPGA logic blocks typically consist of 4-input logic generators (*look-up tables* (LUTs)), programmable sequential elements and additional logic gates for special

arithmetic operations. The storage element can be bypassed in order to implement combinational functions.

The programmable sequential (storage) element can be configured as a latch or a flip-flop. Each has preset and reset signals which can be configured to be synchronous or asynchronous. Reset forces a storage element into the initialization state specified for it in the configuration, while preset forces it into the opposite state.

2.3. Previous Work

In current FPGAs, almost 80% of the area is devoted to interconnect resources, hence many papers have addressed testing and diagnosis of these resources. Application-independent testing of routing resources in FPGAs has been addressed in [Michinishi 96][Renovell 98][Stroud 98][Sun 00] [Tahoori 03]. Diagnosis of interconnect faults in FPGAs has been discussed in [Harris 00][Huang 96][Tahoori 02b][Yu 98]. Fault grading techniques for computing of fault coverage of FPGA interconnect test configuration are presented in [Tahoori 02a]. Application-dependent testing of FPGAs has been addressed in [Das 99][Quddus 99] [Krasniewski 03, 01, 99] [Renovell 03, 01] [Harris 01].

Delay fault testing for FPGAs can be categorized into application-independent and application-dependent delay testing.

Application-dependent delay testing for FPGAs has been addressed in [Krasniewski 01] [Harris 01]. The method presented in [Krasniewski 01] configures unused CLBs as leaner feedback shift registers (LFSRs) for pattern generation and signature analysis. However, delay fault coverage of random patterns is very low. The technique presented in [Harris 01] requires some modifications to the original configuration during placement and routing stages to improve delay fault testability of critical paths in the design using a BIST approach. However, this introduces overheads in terms of mapping efforts (modified placement and routing algorithms) and resource usage (extra logic and routing resources for BIST). Moreover, testing for only critical paths is not very effective in detecting timing defects [Ma 95] [Tseng 01].

Previous work on application-independent delay testing has been presented in [Chmelar 03][Abramovici 03]. Both of these techniques are based on implementing identical routing paths into the FPGA, which are supposed to have identical propagation delays in a defect-free device. Testing is performed by comparing relative delays of those paths: a mismatch in relative delays is considered as a delay fault. However, these techniques cannot identify delay faults induced by global and local process variations which affect all the resources in some portion of the chip (common-mode failure).

In this paper a new technique for robust path delay testing of reprogrammable logic devices is presented by just modifying the configuration of logic blocks. Since the configurations of logic blocks are modified, inter-PLB routing resources are fully tested. This technique detects delay faults along all the paths in the design and guarantees propagation of the fault effect to all reachable primary outputs.

3. Properties of AND (OR) Networks

Consider the logic network shown in Fig. 2. This network only consists of AND functions. A transition test vector consists of 0-to-1 (rising) transitions on the all primary inputs is a robust slow-to-rise transition test for all the paths in the network. Since 1 is the non-controlling value for AND function, the transition at the output of any gate will not happen until the rising transition happens at all the inputs of that gate. As a result, a slow-to-rise transition will be observed on all reachable primary output if there is a slow-to-rise transition fault on any paths or nodes of the design. Hence, this test is not only an all-path robust transition fault test, it is also a TARO test.

Similarly, for a network of OR gates, 1-to-0 transition on all primary inputs is a robust TARO all-path slow-to-fall transition test.

Lemma 1. *In a combinational network of AND (OR) function, 0-to-1 (1-to-0) transition at all primary inputs is a robust all-path delay test. Moreover, all slow-to-rise (slow-to-fall) transition faults are tested and TARO-ed.*

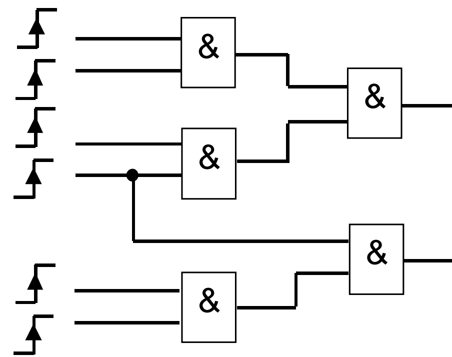

Figure 2 A network of AND function

For similar structure for sequential circuits, consider the circuit shown in Fig. 3. This circuit is a *balanced* sequential network, i.e. the sequential distances of all inputs of each gate from primary inputs are the same. In this circuit, all combinational gates implement the OR functions. Also, the following two conditions are satisfied

for all sequential elements (flip-flops) of the circuit. First, all flip-flops are preset to 1, the controlling value for the OR function. Second, there is no sequential loop in this circuit. By applying the falling transitions on all primary inputs, all slow-to-fall transition faults will be detected.

Lemma 2. *In a balanced sequential network of AND (OR) function, 0-to-1 (1-to-0) transition at all primary inputs is a robust all-path delay test if 1) there is no sequential loop in the logic network, and 2) the initial values of all bistable (latch or flip-flop) elements are 0 (1). Moreover, all slow-to-rise (slow-to-fall) transition faults are tested and TARO-ed.*

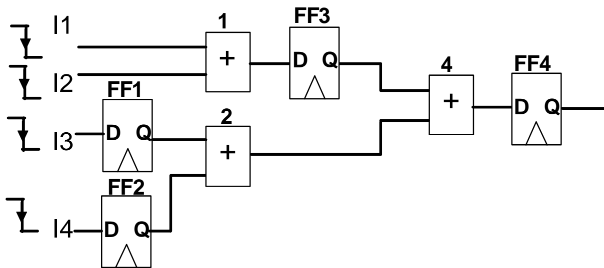

Figure 3 A balanced sequential network of OR functions

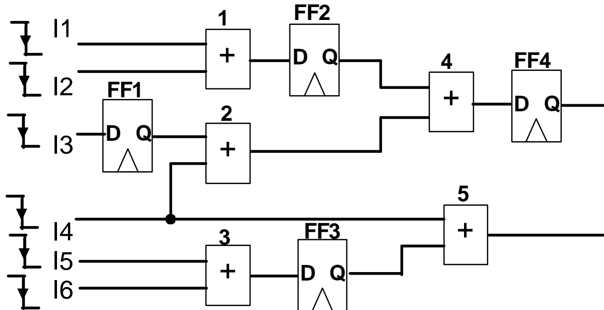

Figure 4 An unbalanced sequential network of OR functions

Since all inputs of each AND (OR) block are changing from a controlling value to a non-controlling value, this results in a robust test, i.e. the output make a transition if and only if all the inputs make their transitions.

In an unbalanced sequential circuit, the conditions of Lemma 2 only guarantee that each node will eventually makes an appropriate transition in some clock cycle. In other words, these do not guarantee that all inputs of each AND (OR) gate in an unbalanced circuit make transitions in the same clock cycle. Consider the unbalanced loop-free circuit shown in Fig. 4. In this circuit, the first input of OR gate 2 makes a falling transition in the second cycle, while the second input makes a falling transition in the first cycle. Since the second input path has one extra cycle to become 0, delay faults on this path cannot be tested. Similar situation happens for the inputs of OR gate 5. In this example, if all primary inputs except I4 make falling transition in the first clock cycle whereas I4 makes a falling transition in the second clock cycles, then for each both inputs of each OR gate make transitions in the same cycle.

Note that in a network of AND (OR) functions if there is a flip-flop whose initial value is 1 (0), all paths originated from the output of that flip-flop cannot be sensitized, since all the nets on those paths are initially set to 1 (0). For example in Fig. 4, if the preset value of FF2 is 0 instead of 1, the path from FF2 to FF4 cannot be sensitized for falling transition.

If there is a sequential loop in the circuit, no transition can be generated for all the nets in the loop even if all the other conditions of Lemma 2 are met. As an example consider the network shown in Fig. 5 which consists of OR functions and flip-flops with a sequential loop. If the preset values of all flip-flops are 1, no transition can happen on any nets other than primary inputs. As mentioned above, if the flip-flops are initiated to 0, some paths cannot be sensitized and tested.

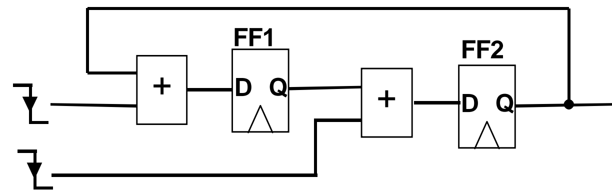

Figure 5 A circuit with sequential loop

4. Dual Test Configurations

Let's assume that the design does not contain any sequential loop. The general case will be considered in Sec. 5.

Based on the Lemma 1 and Lemma 2, we can modify the design mapped into the FPGA to implement the mentioned test structures. As a result, there will be two dual test configurations for a given mapped design. In the first configuration, the *AND-configuration*, all LUTs are configured to implement the AND functions. All the flip-flops used in the design are reprogrammed such that the preset (initial) values of all flip-flops are set to 0. The test vector pair is a 0-to-1 transition on all primary inputs. This test vector pair and test configuration detects all slow-to-rise transition faults, as well as path delay faults in the design.

The second configuration, the *OR-configuration*, is the dual of the first one. In the OR-configuration, all LUTs used in the design are configured to implement the OR function. Also, the preset values of all flip-flops are programmed to 1. The test vector pair is a 1-to-0

transition on all primary inputs. This test vector pair detects all slow-to-fall transition faults and path delay faults on all the paths in the design.

The test vectors for the AND (OR) test configuration consist of all-0 (all-1) followed by all-1 (all-0) at primary inputs. The number of test clock cycles is equal to maximum number of flip-flops on any paths from a primary input to a primary output (i.e. the *maximum sequential depth*). For unbalanced sequential designs, the clock cycles at which each primary input makes transition(s) must be chosen such that the inputs of each AND (OR) gate make transitions in the same cycle. This issue will be addressed in Sec. 6 in details.

These configurations can be obtained from the configuration of the original design by just modifying the contents of logic blocks. Note that in these test configurations, the configuration of routing resources used in the original design remains unchanged.

Note that the user-defined logic can be implemented in additional logic resources in PLBs besides LUTs and flip-flops. The examples of such logic resources are carry generation and propagation logic, cascade chains, etc. Since the proposed test configurations target inter-PLB interconnects, those used additional logic in PLBs are bypassed in the test configurations and used PLBs are configured as LUTs and flip-flops (provided that the flip-flop is used in the original user configuration).

5. Partitioning Algorithm

As mentioned in Sec. 3, in order to achieve the desired robust path delay test, there shouldn't be any sequential loop in the configuration. As a result, the original configuration must be partitioned into a set of loop-free (acyclic) networks.

We construct a *sequential graph*, $G(V,E)$, from the original design. In this graph, each node represents a bi-stable element in the design. An edge between two nodes in the graph denotes that there is a combinational path between the corresponding bi-stables in the circuit.

The sequential graph of a circuit can be constructed as follows. The original design is represented by a netlist which is a circuit graph. In this directed graph, each cell, e.g. gate or flip-flop, is represented by graph node. The interconnections among cells are represented by graph edges. Each node n_i representing a sequential cell (flip-flop) is split into two nodes, n_{i1}, n_{i2}. All incoming edges n_i are connected to n_{i1} and all outgoing edges from n_i are outgoing from n_{i2}. The outgoing degree of n_{i1} and incoming degree of n_{i2} are set to zero (i.e., n_{i1} represent the input part of n_i and n_{i2} represents the output part of n_i, and there is no edge between n_{i1} and n_{i2}).

By performing a *depth first search* (DFS) traversal of this modified graph [Cormen 90] starting from each n_{i1}, the sequential graph will be constructed: for each n_{j2} visited by a DFS originated from n_{i1}, there is an edge from n_i to n_j in the corresponding sequential graph.

The set of edges, E, of this graph is needed to be partitioned into $E_1,\ldots, E_k$, such that $G(V,E_i)$ is acyclic for $i = 1,..,k$. each partition correspond to a pair of test configurations.

We prove that this partitioning can be performed by only two partitions. This is done by using *DFS*. This traversal identifies a *spanning tree* of the graph. A tree which covers all the nodes of the graph using a subset of graph edges. Based on this spanning tree, all the edges are categorized into three groups: *forward* edges, *cross* edges, and *backward* edges. Forward edges correspond to the edges of the spanning tree, or edges from parents to children. Backward edges are those from children to parents with respect to the spanning tree. All the remaining edges are cross edges. Backward edges form cycles in the graph.

The first partition consists of all forward and cross edges, while the second partition consists of backward edges.

Lemma 3. *If a partition, P1, of a graph consists of all forward and cross edges and the other partition, P2, consists of all backward edges, with respect to a DFS traversal of that graph, both partitions are acyclic.*

Proof. Since P1 does not contain any backward edges, i.e. there is no edge from a child node to its ancestor to form a cycle, it is acyclic.

P2 contains only backward edges. For any backward edge (u,v) in P2, v is visited before u in the DFS traversal [Cormen 90]. If P2 contains a cycle, consider cycle C: $u_1,u_2,\ldots,u_n,u_1$. Based on the above statement, for any edge (u_k,u_{k+1}) in C, u_{k+1} must be visited before u_k. Hence, u_n must be visited before u_1. On the other hand, since $(u_n,u_1) \in$ C, so u_1 must have been visited before u_n, which is a contraction. Hence, P2 is also acyclic.

Based on Lemma 3, the partitioning can be completed by just performing a DFS traversal of the graph. The outline of the partitioning algorithm is sketched in Fig. 6.

1. *Convert the netlist graph* $G^n(V^n,E^n)$ *into a sequential graph* $G(V,E)$
2. *Perform DFS traversal of G*
3. $E1 \leftarrow \{forward\ edges\}$
4. $E2 \leftarrow \{cross\ edges\}$
5. $E3 \leftarrow \{backward\ edges\}$
6. $P1 \leftarrow G(V, E1 \cup E2)$
7. $P2 \leftarrow G(V, E3)$

Figure 6 Pseudo code of the partitioning algorithm

The time complexity of this algorithm is $O(|E^n|+|E|)$, since the complexity of converting the netlist graph to the sequential graph is $O(|E^n|)$, and the complexity of acyclic partitioning is $O(|E|)$. In the overall, since $|E^n| > |E|$, the overall complexity of this algorithm is $O(|E^n|)$.

Note that each edge e_i in the sequential graph corresponds to a set of nets, N_i, (combinational paths) in the netlist graph (original design). N_i can be recorded for each e_i during construction of the sequential graph using DFS traversal of the netlist graph, by storing parent lists of the nodes visited in DFS. We define N_1 as follows:

$$N_1 = \cup N_i,\ \forall e_i \in E_1.$$

N_2 and N_3 are defined respectively. Although E1, E2, and E3 are mutually disjoint, N_1, N_2, and N_3 are not necessarily disjoint. Therefore, the set of nets in P1 is $\{N_1 \cup N_2\}$. Similarly, the set of nets in P2 is N_3.

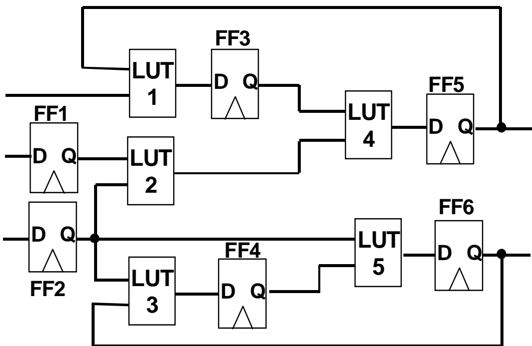

Figure 7 A circuit with sequential loops

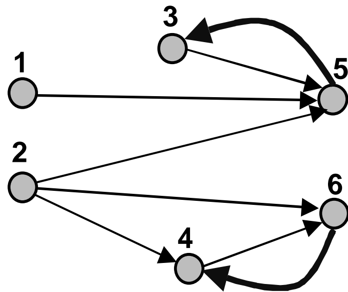

Figure 8 Sequential graph of the circuit shown in Fig. 7

An example of a sequential design is shown in Fig 7. Figure 8 shows the corresponding sequential graph. Backward edges are shown in bold in this graph. Two partitions of this sequential graph are shown in Fig. 9. The resulting two partitions of the original design are shown in Fig. 10.

To summarize, if the design contains any sequential loops, it requires four test configurations (one AND and one OR configuration per each partition) to detect all transition faults as well as all path delay faults. Otherwise, if the sequential graph is acyclic, it requires only two test configurations.

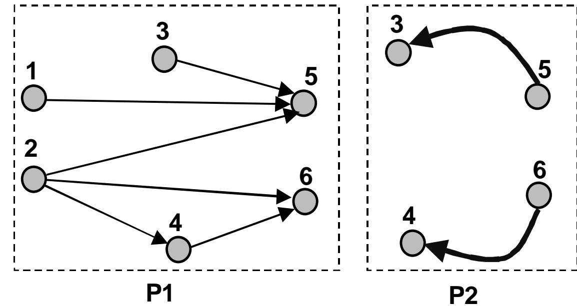

Figure 9 Two acyclic partitions of graph of Fig. 8

6. Test Vector Scheduling

As mentioned in Sec. 4, in order to test all combinational paths in a general unbalanced sequential circuit, the clock cycles at which each primary input makes transition(s) must be chosen such that the inputs of each AND (OR) gate make transitions in the same cycle.

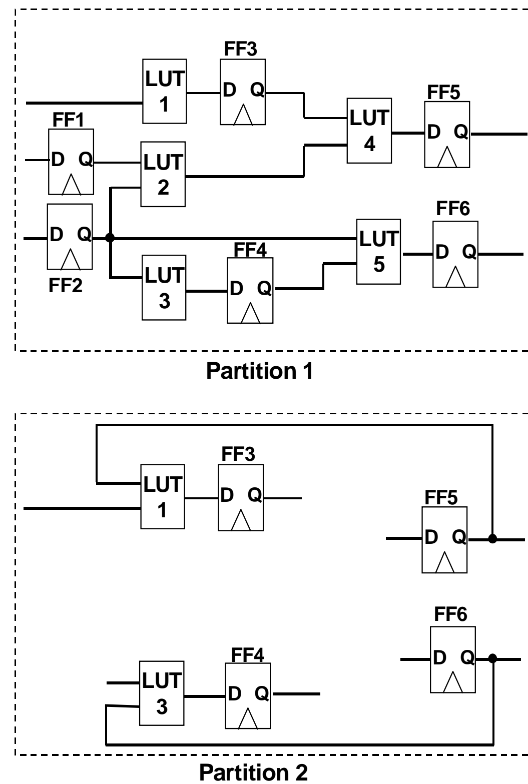

Figure 10 Two acyclic partitions of the circuit in Fig. 7

For each LUT in the design, we compute the sequential distance, *d*, of the *dependent* primary inputs (PI) for the inputs of each LUT. A PI is a dependent PI for an LUT input if there is a path from the PI to that LUT input. We can schedule the relative transition clock cycles, *t*, of those dependent primary inputs such that all inputs of the LUT make transition in the same cycle. For example in Fig. 4, the top input of OR5 is dependent on primary input I4 with sequential distance 0, and the bottom input is dependent on primary inputs I5 and I6 both with sequential distance 1. So, if I4 make a falling transition one cycle later than the falling transitions on I5 and I6, both inputs of OR5 make falling transitions at the same clock cycle. We need to satisfy this condition for all LUTs in the design. Table 1 shows the sequential distances of dependent primary inputs for each LUT in Fig. 4 and the corresponding conditions on relative transition cycles. In this example, if I1, I2, I3, I5, and I6 make transitions in the same cycle and I4 makes a transition one clock cycle later, all conditions will be satisfied.

Note that in the general case, primary inputs can have multiple transitions to satisfy the conditions. The only restriction is that two consecutive (falling) transitions on the same primary input should be at least two clock cycles apart (since we have to generate a rising transition between two falling transitions).

Table 1 Transition cycles for example of Fig. 4

LUT	d(dependent PI)	Transition cycles
OR 1	*d*(I1) = 0, *d*(I2) = 0	*t*(I1) = *t*(I2)
OR 2	*d*(I3) = 1, *d*(I4) = 0	*t*(I4) = *t*(I3) + 1
OR 3	*d*(I5) = 0, *d*(I6) = 0	*t*(I5) = *t*(I6)
OR 4	*d*(I1) = 1, *d*(I2) = 1 *d*(I3) = 1, *d*(I4) = 0	*t*(I1) = *t*(I2) *t*(I4) = *t*(I3) + 1
OR 5	*d*(I4) = 0,*d*(I5) = 1,*d*(I6) =1	*t*(I4)=*t*(I5,I6)+1

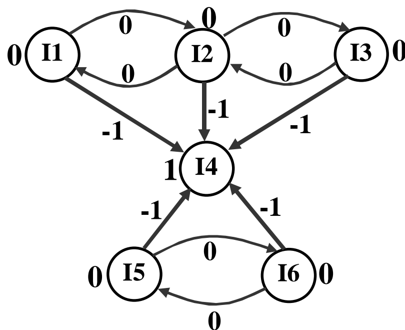

Figure 11 TCDG for the example of Fig. 4

We can build *transition cycle dependency graph* (TCDG) for each circuit. In this weighted directed graph, the primary inputs of the circuit are represented by graph nodes. Edges represent constraint on relative transition cycles for PIs. The edge *(u,v)* with negative weight *w(u,w)* indicates that the transition cycle for the PI *u* must be *-w* cycles *before v*. The corresponding TCDG for this example is shown in Fig. 11.

Constraint 1. The constraints corresponding to each LUT are represented by a subgraph of TCDG. The problem of transition scheduling for PIs can be represented by assigning a set of weights, *W*, to each node of TCDG such that for any path $P = u_0,u_1,\ldots,u_n$ in each subgraph corresponding to an LUT,

$$\exists w_0 \in W(u_0), w_n \in W(u_n) \mid w_0 - w_n = \sum_{i=1}^{n} w(u_{i-1},u_i)$$

The set of weights for each node correspond to transition cycles for the corresponding PI. These weights are also shown in Fig. 11. In this simple example, the conditions can be satisfied by only one weight per each node.

Let's consider a more complicated situation. Assume that the inputs of LUT1 are dependent on PIs *A* and *B* such that *d(A)* =1 and *d(B)* = 2. The inputs of LUT2 are also dependent on *A* and *B* but *d(A)* =2 and *d(B)* = 1. In the corresponding TCDG, there two edges of weight -1 from *A* to *B* and *B* to *A*. These edges form a cycle. In this case, *W(A) = {0,2}* and *W(B) = {1}* is a solution to this problem. Now consider that for another LUT, the inputs are dependent on *A* and *B* with *d(A) = d(B)* = 0. One possible weight assignment could be *W(A) = {0,2}* and *W(B) = {0,1}*. Since it is impossible to generate two transitions in the same direction in two consecutive clock cycles, *W(B) = {0,1}* is unacceptable. To solve this problem, we can assign mutually prime transition periods to *A* and *B*. For example, if *A* makes (falling) transitions every two cycles, and *B* makes (falling) transitions every three cycles, both starting at the same clock cycle, the problem will be solved.

In the general case, if the transition periods of *n* primary inputs are mutually prime (e.g. 2, 3, 5, 7, 11, …), all possible relative ordering of transition clock cycles of these inputs can be generated. One approach to solve this scheduling problem is to assign these prime clock periods for the primary inputs since all possible relative ordering between any two PIs will be generated and any constraint would be satisfied. For example for four PIs, if their transition periods are 2, 3, 5, and 7, respectively, any possible transition ordering will be generated in at most 2×3×5×7=210 clock cycles. Using this approach for all PIs results in a huge number of test clock cycles. However, this needed to be performed only for dependent PIs which is a smaller subset of all PIs. Another approach is to assign weights to TCDG based on Constraint 1 and use these prime cycles for only a subset

of PIs for which transitions in consecutive cycles are required.

The algorithm to satisfy condition 1 in a TCDG is as follows. This is based on a depth-first search of this graph. We first assign weight 0 to the root and assign one weight per each node as we discover the nodes using *tree* edges: for each tree edge (u,v), $w_v = w_u - w(u,v)$. When we encounter a forward or cross edge (x,y), we add a new weight to each node in the corresponding sub-tree from y. Note that for such a forward or cross edge, the entire sub-tree of y is already visited and this edge creates new paths from the root to any previously created path in the sub-tree of y. The only exception is a backward edge (p,q) where p is in the sub-tree of y but q is an ancestor of x. In this case we don't add any new weight for q, since this back edge doesn't add any new path from the root (it only adds a cycle). For the same reason, we do not add new weights for any backward edge. Since the DFS tree is constructed bottom-up (from leaves to the root), all paths are constructed bottom-up and the above procedure doesn't miss any new path from the root to the other nodes.

Since Constraint 1 must be satisfied for all possible pairs of nodes, the above procedure must be executed for all nodes in the graph as DFS root. The outline of this algorithm is shown in Fig. 12.

Since the number of all possible paths in a graph could be exponential to the size of the graph, the worst-case running time of this algorithm is exponential. However, TCDG is partitioned into some subgraphs each representing the constraint for each LUT and this algorithm is executed for each subgraph. Since the number of PIs for a given design is far smaller than the circuit size (at most few hundred), and each subgraph is even smaller (less than a hundred nodes), this approach is quite tractable.

1. **for** each $u_0 \in V$ **do**
2. $W(u) \leftarrow W(u_0) + \{0\}$
3. perform DFS(u_0)
4. **for** each tree edge $(u,v) \in E$ **do**
5. $W(v) \leftarrow W(v) + \{w_u - w(u,v)\}$
6. **for** each forward or cross edge $(x,y) \in E$ **do**
7. perform DFS on y for nodes in subtree of x
8. **for** any visited edge (p,q) such that (p,q) is not a backward edge **do**
9. $path(q) \leftarrow$ weight of path $y \Rightarrow q$ in DFS
10. $W(q) \leftarrow W(q) + \{ w_x - w(x,y) - path(q)\}$

Figure 12 The algorithm for assigning weights

7. Results

The partitioning algorithm is implemented and executed on some FPGA benchmark designs, which are mapped (placed) designs to a generic switch-based FPGA architecture [Alexander 96]. The results, obtain on a SUN (UltraSPARC-III 333MHz) with 128MB of memory, are shown in Table. 2. The number of PLBs and nets in each benchmark circuits are shown in second and third columns, respectively. The fault coverage for robust path delay test, transition faults, and TARO is 100% for all these benchmark circuits. As can be seen in this table, the execution time (configuration generation time) is very small for all the benchmark designs, less than 7 seconds in the worst case.

Table 2 Configuration generation time using the partitioning algorithm

Circuit	PLBs	Nets	Fault Coverage			Exec. Time (sec)
			Robust Path Delay	Transition	TARO	
Term1	90	88	100%	100%	100%	0.10
9symm	110	79	100%	100%	100%	0.14
Appex	120	115	100%	100%	100%	0.17
Busc	156	151	100%	100%	100%	0.29
Exm2	168	205	100%	100%	100%	0.35
Alu2	195	153	100%	100%	100%	0.42
2large	196	186	100%	100%	100%	0.45
Vda	272	225	100%	100%	100%	0.84
Dma	288	213	100%	100%	100%	0.93
Alu4	323	255	100%	100%	100%	1.27
K2	440	404	100%	100%	100%	2.18
Bnre	462	352	100%	100%	100%	2.52
Dfsm	506	420	100%	100%	100%	2.99
Z03	702	608	100%	100%	100%	6.08

8. Discussion

8.1. False Paths and Critical-Paths-Only Test

In the proposed test configurations, the original LUT functions in the user configuration are replaced with AND (OR) functions. As a result, all structural paths in the user configuration are sensitized and their propagation delays are tested to check if they do not exceed the accepted value. However, some of these paths cannot be sensitized when LUTs contain their original user-defined functions (*functional false paths*). This may result in test invalidation due to excessive delay of some false paths (compared to the clock period) which is not visible in the functional mode and hence, systematically identifying some good circuits as defective ones.

However, in the presented technique it is possible to exclude some paths from the test (*path deactivation*). A particular path can be deactivated if in one of the LUTs

along the path (for example the last LUT in the path), the LUT is programmed as a function of all inputs other than the input on the path. An example is shown in Fig. 13. To exclude a particular path from the test (shown in bold in Fig. 13.a), the LUT along the path is reprogrammed to be the AND function of its first three inputs (as shown in Fig. 13.b). In this case, the path from that LUT input to the LUT output is blocked and hence, the entire path is deactivated.

Using this approach, the specific paths, which correspond to false paths in the original design, can be excluded from the test.

Note that short false paths (those whose delay is much less that the clock period) will most likely don't cause any problem at the tester even if they are activated in the test configurations. However, long false paths may cause systematic fails at the tester since they are activated in the test configurations. Hence, if there are consistent fails at some particular output(s) at the tester, we can perform static timing analysis (STA) to identify and trace all long paths (critical path or those within 10% of the delay of critical paths) reaching those failing outputs. Then, we can selectively deactivate these paths and redo testing. If the systematic fails are not observed anymore, this means that the deactivated path is a false path and can be excluded from the test.

It is also possible to include only some specific paths in the test, by excluded the remaining paths from the test. For example, only critical paths can be included in the test, if for any reasons it is desirable to test only the critical paths or any specific set of paths.

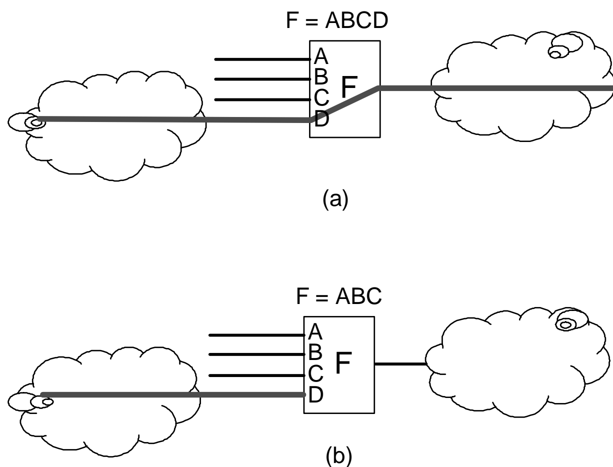

Figure 13 Path deactivation

8.2. Applications to Manufacturing Test

Although the presented technique is originally developed for application-dependent interconnect delay testing, it can be applied to application-independent (manufacturing) interconnect delay testing, as well.

In application-independent interconnect testing, the routing resources are configured as *wires under test* (WUTs) which are concatenation of programmable switched and wire segments [Renovell 98][Stroud 98][Sun 00][Tahoori 03a]. WUTs can pass through logic blocks in which LUTs implement *transparent logic* (identity function).

We can use LUT configuration and test vectors as described in Sec. 4 to test WUTs for delay faults, as well. In this case, LUTs implement AND (OR) logic function instead of transparent logic. Note that using this approach, more WUTs can be tested per test configuration compared to conventional interconnect test configurations, resulting in fewer number of test configurations. This is because only one LUT input is used when LUT is configured as transparent logic (one WUT per LUT) while all LUT inputs can be used when AND or OR functions are implemented.

9. Conclusion

Delay fault testing becomes more important and challenging as the aggressive scaling in feature size progresses and less timing margins are available in designs. Accurate speed binning adds another dimension of complexity to this problem.

In this paper, a new delay fault testing technique for the designs mapped into FPGAs is presented. This technique guarantees 1) 100% robust path delay coverage for all the paths in the design 2) 100% transition fault coverage, and 3) 100% TARO test coverage. Only two or four test configurations are required to achieve the above properties, depending on the existence of sequential loops in the design. A linear time algorithm (with respect to the number of sequential arcs in the design) is presented to automatically generate the test configurations. We have shown how to schedule transition cycles at primary inputs such that all combinational paths at each sequential level make appropriate transitions at the same cycles.

References

[Abramovici 03] M. Abramovici, C. Stroud, "BIST-Based Delay-Fault Testing in FPGAs," Journal of Electronic Testing 19(5): pp. 549-558, 2003.

[Abramovici 02] M. Abramovici, C. Stroud, "BIST based delay fault testing in FPGAs," Proc. 8th IEEE Int'l On-Line Testing Workshop, pp. 131–134, 2002.

[Alexander 96] M. J. Alexander, G. Robins, "New Performance-Driven FPGA Routing Algorithms," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, pp. 1505-1517, 1996.

[Altera 03] http://www.altera.com, 2003.
[Chen 97] L. Chen; S. K. Gupta, M.A. Breuer, "High Quality Robust Tests for Path Delay Faults", proc. VLSI Test Symp., pp. 88-93, 1997.
[Chmelar 03] E. Chmelar, "FPGA Interconnect Delay fault Testing," Proc. Int'l Test Conf., 2003.
[Cormen 01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein "Introduction to Algorithms," McGraw-Hill, 2001.
[Das 99] D. Das, N. A. Touba, "A Low Cost Approach for Detecting, Locating, and Avoiding Interconnect Faults in FPGA-Based Reconfigurable Systems," Proc. Int'l Conf. On VLSI Design, 1999.
[Girard 03] P. Girard, O. Heron, S. Pravossoudovitch, M. Renovell, "Defect Analysis for Delay-Fault BIST in FPGAs", Proc. 9th IEEE Int'l. On-Line Testing Symp., pp. 124-128, 2003.
[Harris 01] I. Harris, P. Menon, R. Tessier, "BIST-based Delay Path Testing in FPGA Architectures," Proc. IEEE Int'l. Test Conf., pp. 932-938, 2001.
[Harris 00] I. Harris, R. Tessier, "Diagnosis of Interconnect Faults in Cluster-Based FPGA Architectures", Proc. of Design Automation Conference, pp. 49-54, 2000.
[Huang 96] W. Huang, X. Cheng, F. Lombardi, "On the Diagnosis of Programmable Interconnect Systems: Theory and Application", Proc. IEEE VLSI Test Symposium, pp. 204-209, 1996.
[Krasniewski 03] A. Krasniewski, "Evaluation of Testability of Path Delay Faults for User-Configured Programmable Devices", in Field Programmable Logic and Applications, Lecture Notes in Computer Science, 2778, pp. 828-838, Springer Verlag, 2003.
[Krasniewski 01] A. Krasniewski, "Testing FPGA delay-faults in the System Environment is very Different from Ordinary delay-fault Testing," Proc. IEEE Int'l On-Line Test Workshop, pp. 37-40, 2001.
[Krasniewski 99] A. Krasniewski, "Application-Dependent Testing of FPGA Delay Faults," Proc. 25th EUROMICRO Conf., vol. 1, pp. 260-267, 1999.
[Ma 95] S. C. Ma, P. Franco, E.J. McCluskey, "An Experimental Chip to Evaluate Test Techniques Experiment Results," Proc. 1995 Int. Test Conf., pp. 663-672, 1995.
[Michinishi 96] H. Michinishi, T. Yokohira, T. Okamoto, "A Test Methodology for Interconnect Structures of LUT-Based FPGAs," Proc. Asian Test Symp., pp.68-74, 1996.
[Quddus 99] W. Quddus, A. Jas, N. A. Touba, "Configuration Self-Test in FPGA-Based Reconfigurable Systems," Proc. ISCAS'99, pp. 97-100, 1999.
[Renovell 03] M. Renovell, "Some Aspects of the Test Generation Problem for an Application-Oriented Test of SRAM-Based FPGAs", Journal of Circuits, Systems, and Computers, vol. 12, no. 2, pp. 143-158, 2003.
[Renovell 01] M. Renovell, P. Faure, J.M. Portal, J. Figueras, Y. Zorian, "IS-FPGA: A New Symmetric FPGA Architecture with Implicit SCAN," Proc. IEEE Int'l Test Conf., pp. 924-931, 2001.
[Renovell 98] M.Renovell, J.M.Portal, J Figuras, Y. Zorian, "Testing the Interconnect of RAM-Based FPGAs", IEEE Design and Test of Computers, pp. 45-50, 1998.
[Stroud 98] C. Stroud, S. Wijesuriya, C. Hamilton, M. Abramovici, "Built-in self-test of FPGA interconnect," Proc. Int'l Test Conf., pp.404-411, 1998.
[Sun 00] X.Sun, J.Xu, B.Chan, P. Trouborst, "Novel Technique for Built-In Self-Test of FPGA Interconnects", Proc. Int'l Test Conf., pp. 795-803, 2000.
[Tahoori 03] M.B. Tahoori, S. Mitra, "Automatic Test Configuration Generation for FPGA Interconnect Testing", Proc. IEEE VLSI Test Symp., 2003.
[Tahoori 02a] M.B. Tahoori, S. Mitra, S. Toutounchi, E.J. McCluskey, "Fault Grading FPGA Interconnect Test Configuration", Proc. Int'l Test Conf., 2002.
[Tahoori 02b] M.B.Tahoori, "Diagnosis of Open Defects in FPGA Interconnects", Proc. of IEEE Int'l Conf. on Field-Programmable Technology, pp. 328-331, 2002.
[Tseng 01] C. W. Tseng, E.J. McCluskey, "Multiple-Output Transition Fault ATPG," Proc Int'l Test Conf., 2001.
[Toutounchi 02] S. Toutounchi, A. Lai, "FPGA Test and Coverage," Proc. Int'l Test Conf., pp. 599-607, 2002.
[Yu 98] Y. Yu, J. Xu, W. Huang, F. Lombardi, "A Diagnosis Method for Interconnects in SRAM based FPGAs", Proc. Asian Test Symp., pp. 278-282, 1998.
[Xilinx 03] Programmable Logic Data Book, Xilnx Inc, 2003.
[Xilinx EasyPath] Xilinx EasyPath Solution, http://www.xilinx.com, 2003.