Exploiting Off-Line Hierarchical Test Paths in Module Diagnosis and On-Line Test

Yiorgos Makris & Alex Orailoğlu

Reliable Systems Synthesis Lab
Computer Science & Engineering Department
University of California San Diego
9500 Gilman Drive MC-0114
La Jolla, CA 92122, USA
E-mail: {makris, alex}@cs.ucsd.edu

Abstract

We propose a RT-Level hierarchical test generation methodology based on the concept of transparent hierarchical test paths, which results in significant test generation speedup over traditional gate-level ATPG, while preserving equivalent fault coverage and vector count. Additionally, we demonstrate how hierarchical test paths may assist in design diagnosis and debug through an algorithm that disambiguates among several possibly faulty modules in a design. Furthermore, we devise an on-line test methodology using invariance inherent in hierarchical test paths and in the algorithm implemented by controller/datapath pairs. This scheme results in fault security in excess of 90% while keeping the hardware overhead below 40%. Utilization of hierarchical test paths not only for off-line test but also for design diagnosis and on-line test helps in reducing the overhead associated with each of these tasks, by amortizing the cost of the required hardware.

1. Introduction

The size and complexity of modern electronic circuits impose significant challenges to several tasks such as off-line test generation, design diagnosis and on-line test. Significant efforts are therefore expended towards simplifying the aforementioned tasks. However, such efforts incur an additional cost both in developing efficient test and diagnosis methodologies and in area and performance overhead in the actual design. While it is inevitable that additional cost needs to be invested into such methodologies, it is highly desirable to amortize this cost among several tasks, in order to minimize the overhead. Towards this direction, we describe how a common off-line hierarchical test methodology may be used to simplify the design diagnosis problem and to support an on-line test scheme. More specifically, we demonstrate how the concept of transparent hierarchical test paths, commonly used for off-line test, can be utilized for identifying faulty modules in a hierarchical design and for devising an invariance-based on-line test methodology.

In section 2, we discuss the concept of transparent hierarchical test paths, based on which a hierarchical test generation methodology is proposed, achieving equivalent fault coverage to gate-level ATPG in significantly less time as shown through experimental results. In section 3, we discuss the role of hierarchical test paths in assisting with the problem of design diagnosis. In section 4, we utilize the inherent invariance of transparent hierarchical test paths, in order to devise a low-cost, highly fault secure on-line test methodology, as supported through experimental results. Conclusions about the applicability of hierarchical test paths are drawn in section 5.

2. Hierarchical test paths and off-line test

sIn hierarchical test methodologies [1,2,5,6,11], highly efficient test is locally generated for each module. However, the success of these approaches relies on the ability to apply the locally generated test to each module through the surrounding logic. Reachability paths through the upstream and downstream modules are utilized for justifying vectors and propagating responses to the module under test, as depicted in figure (1). Such paths may be either inherent in the design specification or explicitly incorporated in the design implementation through DFT hardware. The transparency behavior of the surrounding modules is utilized on these paths to establish bijective functions between the primary inputs (outputs) of the design and the inputs (outputs) of the module under test. Through these bijective functions, test vectors and responses can be justified to the inputs and propagated from the outputs of the module under test. The concept of *transparency channels* for capturing bijective behavior is introduced in this section and their applicability in RT-Level hierarchical test generation is discussed.

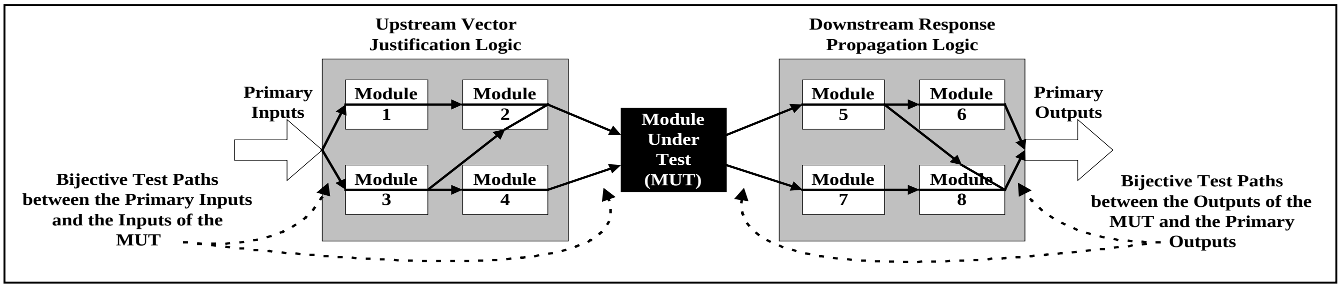
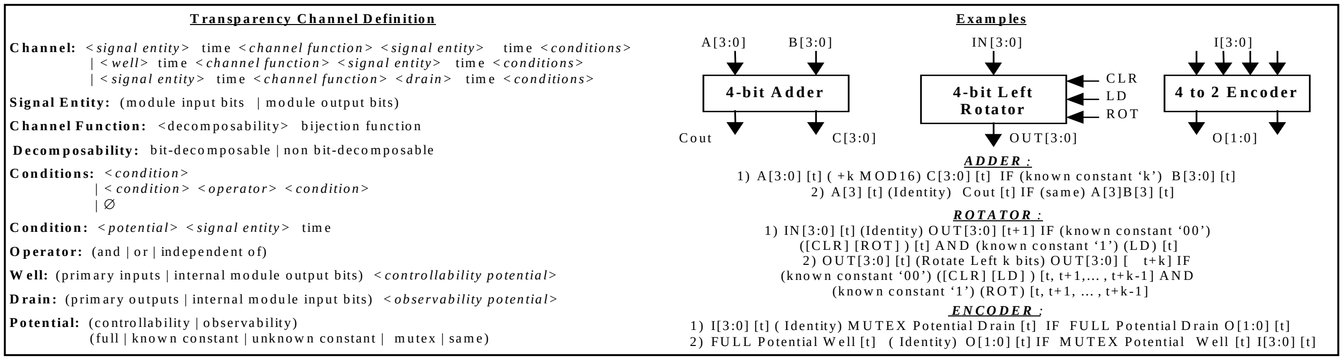

Upstream Vector Justification Logic
Downstream Response Propagation Logic
Primary Inputs
Module 1
Module 2
Module 3
Module 4
Module Under Test (MUT)
Module 5
Module 6
Module 7
Module 8
Primary Outputs
Bijective Test Paths between the Primary Inputs and the Inputs of the MUT
Bijective Test Paths between the Outputs of the MUT and the Primary Outputs

Figure (1): *Hierarchical Test Paths*



Transparency Channel Definition

Channel: <*signal entity*> time <*channel function*> <*signal entity*> time <*conditions*>
| <*well*> time <*channel function*> <*signal entity*> time <*conditions*>
| <*signal entity*> time <*channel function*> <*drain*> time <*conditions*>

Signal Entity: (module input bits | module output bits)

Channel Function: <decomposability> bijection function

Decomposability: bit-decomposable | non bit-decomposable

Conditions: <*condition*>
| <*condition*> <*operator*> <*condition*>
| ∅

Condition: <*potential*> <*signal entity*> time

Operator: (and | or | independent of)

Well: (primary inputs | internal module output bits) <*controllability potential*>

Drain: (primary outputs | internal module input bits) <*observability potential*>

Potential: (controllability | observability)
(full | known constant | unknown constant | mutex | same)

Examples

ADDER :
1) A[3:0] [t] (+k MOD16) C[3:0] [t] IF (known constant 'k') B[3:0] [t]
2) A[3] [t] (Identity) Cout [t] IF (same) A[3]B[3] [t]

ROTATOR :
1) IN[3:0] [t] (Identity) OUT[3:0] [t+1] IF (known constant '00')
([CLR] [ROT]) [t] AND (known constant '1') (LD) [t]
2) OUT[3:0] [t] (Rotate Left k bits) OUT[3:0] [t+k] IF
(known constant '00') ([CLR] [LD]) [t, t+1,... , t+k-1] AND
(known constant '1') (ROT) [t, t+1, ... , t+k-1]

ENCODER :
1) I[3:0] [t] (Identity) MUTEX Potential Drain [t] IF FULL Potential Drain O[1:0] [t]
2) FULL Potential Well [t] (Identity) O[1:0] [t] IF MUTEX Potential Well [t] I[3:0] [t]

Figure (2): *Transparency Channel Definition and Examples*

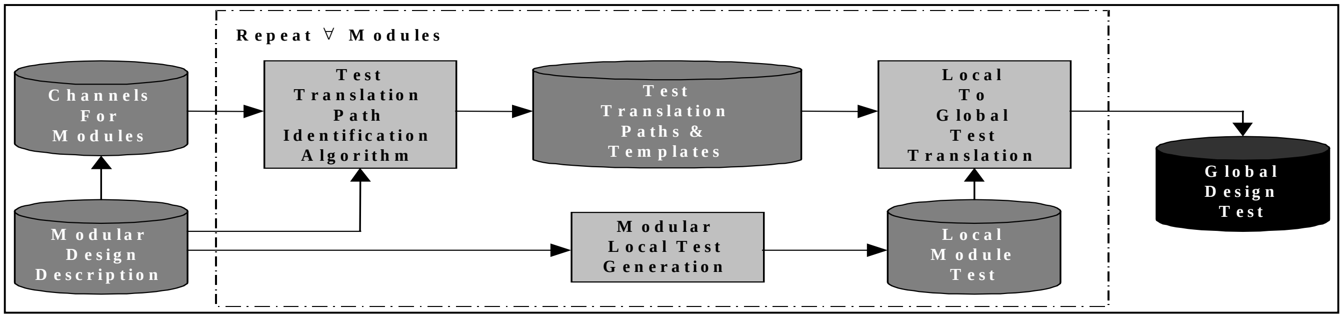

Repeat ∀ Modules
Channels For Modules
Test Translation Path Identification Algorithm
Test Translation Paths & Templates
Local To Global Test Translation
Global Design Test
Modular Design Description
Modular Local Test Generation
Local Module Test

Figure (3): *RT-Level Hierarchical Test Generation*

2.1. RTL hierarchical test generation

Transparency channels capture modular transparency in terms of bijection functions between input and output *signal entities*. Channel functions may be *bit-decomposable* (e.g. left rotation) or not (e.g. addition MOD k). Channels are instantiated upon compliance of *conditions* that require a specific *potential* on signal entities. The potential may be either controllability or observability of the signal entity to a set of values. Signal entities may be defined either on the full word bitwidth or on sub-word bitwidths. A succinct definition of the transparency channels is given in figure (2) along with a few examples of simple RTL modules and channels.

2.2. RTL hierarchical test generation

Hierarchical test paths facilitate a powerful test generation methodology resulting in significant test generation time reduction and highly efficient test for modular designs. In the proposed methodology hierarchical test paths comprise bijective behavior of the modules on the path, captured in terms of *transparency channels* [4]. Local test is generated for each module and subsequently translated through the hierarchical test paths into global design test. The proposed methodology is shown in figure (3). This scheme is independent of the actual method employed for local test generation for each module. Since globally translated vectors are functionally justified at-speed to the module under test, any fault model can be accommodated. In addition, channel-based translation paths are identified regardless of the locally generated test vectors. As a result, local tests can be modified and enhanced in order to provide higher fault coverage, without invalidating the translation paths.

A recursive design traversal algorithm introduced in [5] is applied for each module in the design, employing transparency channels in order to identify test justification and propagation paths. The algorithm traverses the design, backtracking as necessary, in order to satisfy the

Benchmark Circuit	Gate-Level ATPG	Hierarchical Test Generation
MTC100	3.383 sec	0.44 sec
MUL	13.58 sec	4.63 sec
MAC	21.30 sec	8.87 sec

Table (1): Test Generation Time Comparison

Benchmark Circuit	Gate-Level ATPG	Hierarchical Test Generation	Random Test Generation
MTC100	1034 faults	1038 faults	748 faults
MUL	760 faults	790 faults	569 faults
MAC	27896 faults	27245 faults	22454 faults

Table (2): Fault Coverage Comparison

Benchmark Circuit	Gate-Level ATPG	Hierarchical Test Generation
MTC100	146 vectors	178 vectors
MUL	191 vectors	198 vectors
MAC	627 vectors	703 vectors

Table (3): Vector Count Comparison

test requirements for each module. While traversing an upstream/downstream module, available channels are probed as to their suitability for providing the required potential. Channels may be combined into wider channels or broken into smaller ones. Reconvergence and feedback loops are considered in order to prioritize the probing of channels, accelerating algorithm convergence. The transparency channels and conditions on the test translation paths are reported and subsequently combined into test translation templates for each module. An extensive description of the test requirements, the recursive path identification algorithm and its applicability on testability analysis may be found in [5]. Channels on the identified vector justification and response propagation paths are combined into test translation templates. Using these templates, the actual test translation can be rapidly performed.

2.3. Experimental results

In order to evaluate the proposed hierarchical RT-level test generation methodology, we compare it to full circuit gate level ATPG and to random test generation. The experimental setup is applied on three benchmark designs. Full circuit gate-level ATPG [8] is applied and the test generation time, fault coverage and vector count is obtained. Then, the proposed scheme is applied and the test generation time, fault coverage and vector count are also noted. Finally, a number of random patterns [3] equal to the number of patterns produced by the proposed method are fault simulated [9] and the corresponding fault coverage is obtained. As shown in Table (1), in the first circuit the reduction was almost of an order of magnitude, while for the other two circuits it was approximately 65%. Table (2) shows that fault coverage slightly increased in the first two circuits, while in the third there was a 2.5% drop. Random vectors achieved significantly lower coverage in all cases. In terms of vector count, a slight increase in the order of 10-15% was observed, as shown in Table (3), due to the divide-&-conquer scheme. According to these results, transparency channels facilitate a powerful RT-level hierarchical test generation.

3. Module diagnosis

The module reachability capacity of hierarchical test paths is currently extensively exploited for the purpose of hierarchical test application. Based on such paths, hierarchical test methods reveal the existence of faults in the design. However, the capabilities of the hierarchical test paths are not fully exploited. Additional information, relevant to fault diagnosis and design debugging [10], is inherently attainable through these paths. Thorough utilization of each path may not only provide information for the MUT, but also assist in identifying possibly faulty modules and exonerating unambiguously non-faulty modules. We first investigate the debug-related information that can be obtained from hierarchical test paths. We then introduce an algorithm that utilizes this information in order to identify the minimal set of possibly faulty modules under the single faulty module assumption. Finally, we give a necessary and sufficient condition relating the modules on the hierarchical paths so that we can always identify the faulty module, under any combination of test path outcomes.

3.1 Debug information

We first examine which modules are *fully testable* through a path. *Full testability* implies that a complete test set, capable of 100% fault coverage, can be applied to a module. Each hierarchical test path has the ability to provide the complete test set and evaluate all the responses of the *MUT*. Additional modules may also be *fully testable* through this particular test path, if the *MUT* exhibits appropriate bijection behavior. Such modules need to have all their inputs and outputs on the bijection path used for *fully testing* the *MUT*. For each *fully testable* module on the path, we can apply the complete set of test vectors and attain the following information, depending on the test application outcome:

i) If no faulty response is reported then we certainly know that the module for which the complete test set was applied is not faulty and can be exonerated. However, no additional conclusions can be drawn about other modules in the design.

ii) If a faulty response is obtained, any module on the path can be the faulty one. However, under the single faulty module assumption, if a fault has already been reported while applying the test set of a previous module, then the faulty module has to be in the intersection of the cones of logic used for testing the current module and the previous module. All other modules can be exonerated. In addition, we may be able to exonerate some modules if the observation path splits and one of the sub-paths always produces correct responses. Under the single faulty module assumption, the fault has to be in the cone of logic driving the observation sub-path that reports the fault. Any module on the path outside this cone of logic can be exonerated. Modules on the common portion of the path cannot be exonerated, since a fault in them may affect only one sub-path.

3.2 Module diagnosis algorithm

In this section we utilize the debug information provided by hierarchical test paths in order to devise a faulty module diagnosis algorithm, which we further demonstrate by an example. The input to the faulty module diagnosis algorithm is a set of hierarchical test paths available in a design. Each path has associated with it a list of modules that are *fully testable* through this path and the test vectors for each of these modules. The algorithm utilizes these paths in order to apply the test vectors to each *fully testable* module and combines the attained information in order to provide a minimal list of possibly faulty modules. Initially the candidate list comprises all design modules. Each time the complete test set of a module is applied through a path, modules are removed from the list according to the disambiguation criteria of the previous section. The algorithm is provided below in pseudo-code form:

```
Candidate_List = {All Design Modules};
For each Path
  {For each Fully Testable Module on the Path
    {Apply Complete Set of Test Vectors to Module;
      If no fault is reported
        {Reduce Candidate_List according to case (i);}
      else
        {Candidate_List=Candidate_List $\cap$ All Modules on Path;
          Reduce Candidate_List according to case (ii);}}}
```

Depending on the participating modules, paths may not always be able to identify the faulty module. The following section provides a rule for checking if a given set of paths can always diagnose the faulty module.

3.3 Disambiguation rule

If an arbitrary module *M* is faulty then any path on which *M* is *fully testable* will certainly report a fault. The modules on the intersection of these paths are the possibly faulty modules. Additionally, any path not using *M* at all will not report a fault, exonerating all modules that are *fully testable* through these paths. Based on this information, the following relation between the modules on the hierarchical test path is a necessary and sufficient condition for always reducing the faulty module *Candidate_List* to a single module.

Disambiguation Rule: If *M* is a module in the design, let *PT(M)* be the set of paths that can *fully test* module *M*, and let *PNC(M)* be the set of paths that do not contain *M*. Let also *AM(P)* be the set of all modules on a path *P* and let *TM(P)* be the set of all modules that a path *P* can *fully test*. We can diagnose the faulty module if and only if:

$$\forall M : \left\{ \bigcap_{x \in PT(M)} AM(x) \right\} - \left\{ \bigcup_{x \in PNC(M)} TM(x) \right\} = \{M\}$$

4. On-line test

Transparency behavior, whether inherent in the design or explicitly incorporated through DFT, provides a simple and rapid mechanism for traversing a hierarchical design in order to access and test a module through reachability paths. In this section we examine the applicability of modular transparency for on-line test [7].

4.1 Modular transparency and on-line test

As discussed in section 2, transparency channels express modular transparency behavior in terms of bijective functions, appropriate for justifying vectors and propagating responses to and from the module under test. Such behavior is activated under certain conditions imposed on the inputs of the module in transparency mode. The transparency functions most commonly employed, such as *Identity* and *Inversion*, establish a simple relationship between the inputs and the outputs of the module in transparency mode. Checking that this simple relationship holds, whenever the respective activation conditions are met during normal functionality, provides an on-line test scheme capable of detecting any fault that disturbs the transparency behavior of the module, as demonstrated in figure (4). In addition, whenever the transparency function has an *if-and-only-if* relation to the conditions, we can perform a dual-rail logic check. We not only verify the transparency existence whenever the activation conditions hold, but also verify the lack of transparency whenever the activation conditions do not hold, thus increasing the number of faults that can be detected through this mechanism.

4.2 Transparency-based path invariance

Checking each transparency function of every module in the design would result in very expensive hardware overhead, making such a scheme impractical. In order to reduce this cost, the key idea is to combine several of these modular transparency functions on a transparent path, such that only one checking mechanism for the complete path suffices for all the constituent transparency functions. Such a path may span across several modules of the design and across several clock cycles.

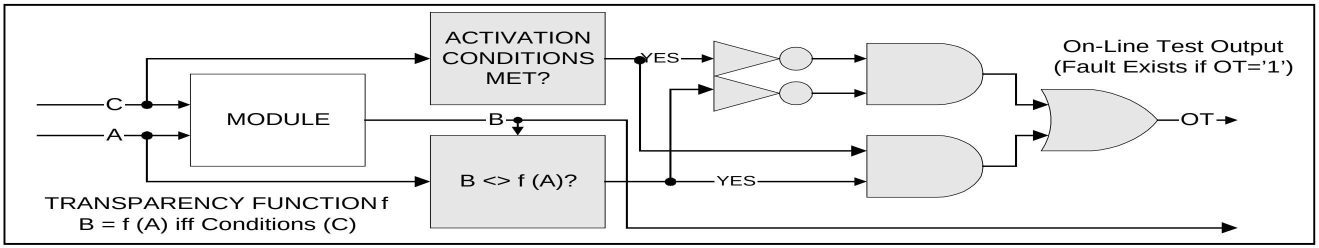

ACTIVATION CONDITIONS MET?
YES
On-Line Test Output (Fault Exists if OT='1')
C
A
MODULE
B
OT
B <> f (A)?
YES
TRANSPARENCY FUNCTION f
B = f (A) iff Conditions (C)

Figure (4): *On-Line Test via Modular Transparency Functions*

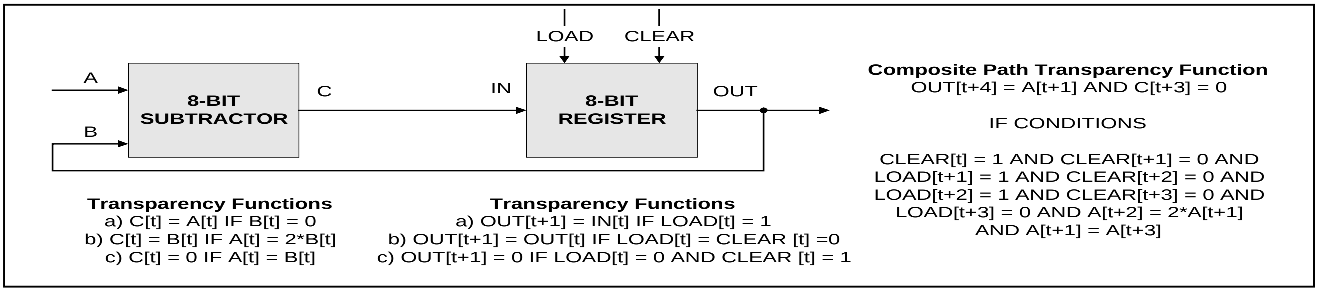

LOAD
CLEAR
Composite Path Transparency Function
OUT[t+4] = A[t+1] AND C[t+3] = 0
IF CONDITIONS
CLEAR[t] = 1 AND CLEAR[t+1] = 0 AND
LOAD[t+1] = 1 AND CLEAR[t+2] = 0 AND
LOAD[t+2] = 1 AND CLEAR[t+3] = 0 AND
LOAD[t+3] = 0 AND A[t+2] = 2*A[t+1]
AND A[t+1] = A[t+3]
A
8-BIT SUBTRACTOR
B
C
IN
8-BIT REGISTER
OUT
Transparency Functions
a) C[t] = A[t] IF B[t] = 0
b) C[t] = B[t] IF A[t] = 2*B[t]
c) C[t] = 0 IF A[t] = B[t]
Transparency Functions
a) OUT[t+1] = IN[t] IF LOAD[t] = 1
b) OUT[t+1] = OUT[t] IF LOAD[t] = CLEAR [t] =0
c) OUT[t+1] = 0 IF LOAD[t] = 0 AND CLEAR [t] = 1

Figure (5): *Transparency Path Composition Example*

As an example of transparent path composition, consider the circuit shown in figure (5). Three transparency functions are provided for each of the two modules in the design, requiring six distinct checking mechanisms. However, a path spanning 5 clock cycles can be composed, comprising all of the six transparency functions. Although the activation conditions of the modular transparency functions still need to be checked for, the composite path has a simple-to-check transparency function, requiring only one checking mechanism instead of six. As a result, path composition reduces significantly the hardware required, while preserving the attained fault coverage.

4.3 Algorithmic path invariance

While modular transparency functions and the composite invariant paths are capable of detecting many faults, the resulting fault coverage may not be adequate in itself to ensure high fault security in the design. Additional sources of invariance, capable of checking for faults that can not be detected through transparency functions, are therefore required. Invariant behavior of the implemented algorithm, captured through the interaction between the datapath and the controller, is such a source. Algorithmic invariance can be identified directly from the controller-datapath interaction, thus providing additional invariant paths in the design. Algorithmic invariance essentially captures the restrictions imposed by the controller on the datapath components. Any fault causing a deviation from this restricted behavioral domain of the controller-datapath pair will be detected.

The most interesting cases of algorithmic invariance capture non-transparency behavior, therefore covering faults that cannot be detected through transparency functions. Especially in the controller, where transparency rarely exists, algorithmic invariance helps significantly in achieving high fault security. Additionally, algorithmic invariance has a clear advantage in terms of activation frequency; algorithmically invariant paths are always active and therefore the on-line test scheme is continuously checking for errors. Even if such invariance is valid only in parts of an algorithm, the activation frequency is still expected to be very high, reflecting the activation frequency of the specific part of the algorithm. In short, identification of simple-to-check algorithmic path invariance provides a very efficient on-line test mechanism that complements the transparency-based path invariance methodology in terms of coverage and latency.

4.4 Experimental results

To evaluate the proposed methodology we examine the fault security and fault coverage achieved by the identified path invariance, as well as the area overhead imposed by the invariance checking hardware on three benchmarks. These are difficult-to-test sequential circuits, implemented as controller-datapath pairs, on which ATPG has a hard time reaching high fault coverage. We first apply gate-level ATPG using HITEC [8], in order to obtain the *deterministic off-line test fault coverage* as a reference point. Subsequently, for the main part of the experimental validation, we utilize fault simulation of random input values for each design. For the GCD circuit, 1000 random pairs of numbers are generated and the corresponding GCDs are calculated. These vectors are fault simulated on the design using HOPE [3] and the *random off-line test fault coverage* is obtained, along with the set of covered faults. The design is then augmented with path invariance checking hardware and an on-line test output pin. Only the on-line test output is considered a primary output in the modified design. The same random vectors are then fault simulated on the

	Total Faults	Deterministic Off-Line Test Fault Coverage	Random Off-Line Test Fault Coverage	Random On-Line Test Fault Security	Random On-Line Test Fault Coverage	Area Before	Area After	Area Overhead
GCD	973	863/973 (88.69%)	822/973 (84.48%)	744/822 (90.51%)	781/973 (80.26%)	480 Gates 27 F-Fs	646 Gates 39 F-Fs	36.39%
MINMAX	1185	1117/1185 (94.26%)	1054/1185 (88.94%)	952/1054 (90.32%)	996/1185 (84.05%)	534 Gates 56 F-Fs	781 Gates 64 F-Fs	36.80%
MULT	834	798/834 (95.68%)	769/834 (92.20%)	693/769 (90.11%)	719/834 (86.21%)	356 Gates 30 F-Fs	471 Gates 48 F-Fs	39.28%

Table (4): *Experimental Results*

modified design, targeting only the faults covered in off-line random vector fault simulation. The fault coverage achieved in this experiment indicates the percentage of faults that can be detected both on the original and on the modified design, thus providing the *random on-line test fault security*. In addition, the random vectors are fault simulated targeting all faults in the modified design, in order to obtain the *random on-line test fault coverage*. The results are summarized in Table (4), where the area overhead of the proposed scheme is also shown. The area overhead is calculated assuming an implementation with only 2-input gates and 4 gates for each flip-flop.

The obtained results demonstrate that the proposed on-line test methodology achieves fault security exceeding 90%, while area overhead is kept below 40%. The proposed scheme also provides on-line random test fault coverage that is only 6% worse than random off-line fault coverage on difficult-to-test sequential benchmark designs. It is important to note that a large portion of the remaining faults is due to primary I/O faults that no invariance-based, on-line test methodology can capture.

5. Conclusions

We discuss an efficient off line hierarchical test methodology based on the concept of transparent hierarchical test paths. The proposed RTL test generation methodology results in significant test generation speedup while preserving fault coverage and vector count equivalent to traditional gate-level ATPG. Furthermore, we demonstrate the applicability of hierarchical test paths for design diagnosis. We discuss the debug-related information that can be obtained through hierarchical test paths and we describe an algorithm that utilizes this information to identify the minimal set of possibly faulty modules. Finally, we utilize the invariance present in transparent hierarchical test paths and we propose an on-line test methodology achieving high fault security with very low hardware overhead. In short, we demonstrate that a single feature, namely the transparent hierarchical test paths, may be utilized across several test-related tasks, amortizing and reducing the cost incurred by each individual task.

References

[1] M. S. Abadir, M. A. Breuer, “A Knowledge-Based System for Designing Testable VLSI Chips”, *IEEE Design and Test of Computers*, vol. 2, no. 4, pp. 56-68, 1985.

[2] S. Freeman, “Test Generation for Data-Path Logic: The F-Path Method”, *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 421-427, 1988.

[3] H. K. Lee, D. S. Ha, “HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048-1058, 1996.

[4] Y. Makris, A. Orailoğlu, “DFT Guidance Through RTL Test Justification and Propagation Analysis”, *International Test Conference*, pp. 668-677, 1998.

[5] Y. Makris, A. Orailoğlu, “RTL Test Justification and Propagation Analysis for Modular Designs”, *Journal of Electronic Testing: Theory & Applications*, Kluwer Academic Publishers, vol. 13, no. 2, pp. 105-120, 1998.

[6] B. T. Murray, J. P. Hayes, “Hierarchical Test Generation Using Precomputed Tests for Modules”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 6, pp. 594-603, 1990.

[7] M. Nicolaidis, Y. Zorian, “On-Line Testing for VLSI-A Compendium of Approaches”, *Journal of Electronic Testing: Theory & Applications*, Kluwer Academic Publishers, vol. 12, no. 1-2, pp. 7-20, 1998.

[8] T. Niermann, J. H. Patel, “HITEC: A Test Generation Package for Sequential Circuits”, *European Conference on Design Automation*, pp. 214-218, 1992.

[9] T. Niermann, W. T. Cheng, J. H. Patel, “PROOFS: A Fast, Memory Efficient Sequential Circuit Fault Simulator”, *Design Automation Conference*, pp. 535-540, 1990.

[10] *Practical Aspects of IC Diagnosis and Failure Analysis: A Walk through the Process,* IEEE Computer Society Press, *International Test Conference*, Lecture Series II, 1996.

[11] P. Vishakantaiah, J. Abraham and M. S. Abadir, “ATKET: Automatic Test Knowledge Extraction From VHDL”, *Design Automation Conference*, pp. 273-278, 1992.