A Module Diagnosis and Design-for-Debug Methodology Based on Hierarchical Test Paths

Yiorgos Makris & Alex Orailoğlu
Reliable Systems Synthesis Lab
Computer Science & Engineering Department
University of California, San Diego
{makris, alex}@cs.ucsd.edu

Abstract

Fault identification capabilities are becoming increasingly important in modern designs, not only in support of design debugging methodologies, but also for the purpose of process characterization and yield enhancement. At the same time, hierarchical test approaches are becoming the prevalent means for addressing the size and complexity of large designs and for accommodating the varying individual test needs of each design module. In this paper, we discuss a module diagnosis and design-for-debug methodology through hierarchical test paths. Based on debug information inherently attainable from hierarchical test paths, we outline a diagnosis algorithm that identifies the minimal set of faulty module candidates, under the single faulty module model. We further provide a disambiguation rule to ensure unfailing identification of the single faulty module. Low-cost, design-for-debug techniques are subsequently proposed for establishing the disambiguation rule and for providing a module diagnosis capability.

1. Introduction

In an effort to address size and complexity considerations, hierarchical approaches [6, 9, 10, 13, 14] have become the dominant strategy for testing modern designs. Within such hierarchical strategies, highly efficient test is locally generated for each module, tuned to the individual module test requirements. However, the success of these approaches relies on the ability to apply the locally generated test to each module through the surrounding logic. For this purpose, 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, establishing bijective functions between the primary inputs (outputs) of the design and the inputs (outputs) of the module under test. Through these bijective functions [1, 2, 8], test vectors and responses can be justified to the inputs and propagated from the outputs of the module under test respectively.

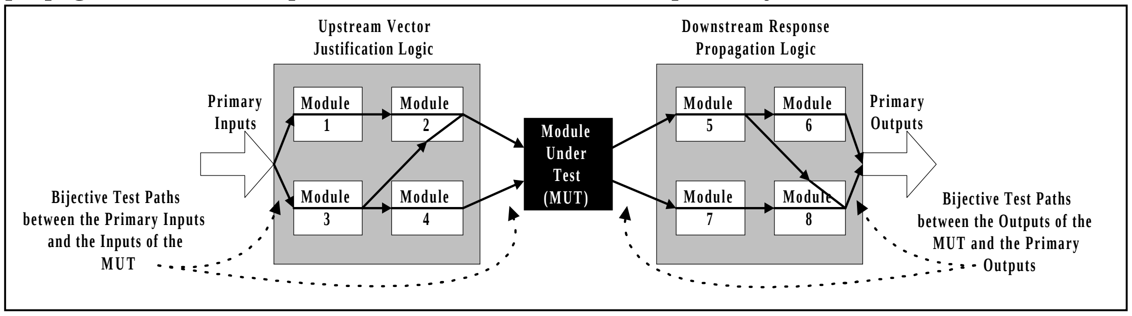
Upstream Vector Justification Logic
Downstream Response Propagation Logic
Primary Inputs
Module 1
Module 2
Module 3
Module 4
Module Under Test (MUT)
Module 5
Module 6
Module 7
Module 8
Primary Outputs
Bijective Test Paths between the Primary Inputs and the Inputs of the MUT
Bijective Test Paths between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

Figure (1): Hierarchical test generation and application through bijective test paths

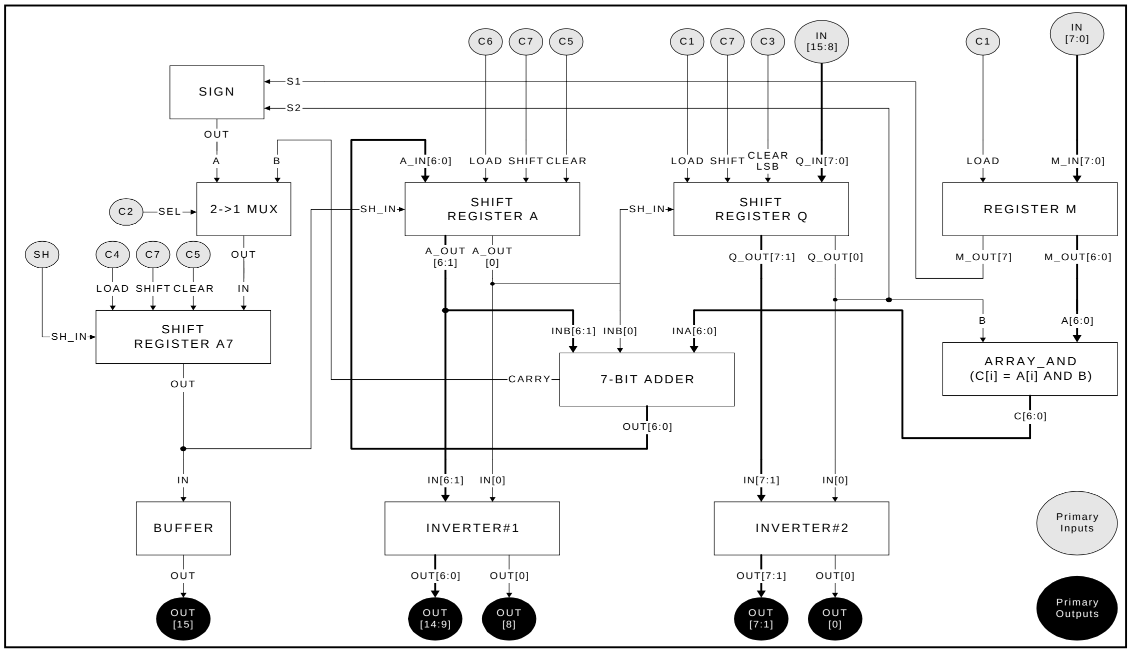

Figure (2): An 8-bit shift-&-add multiplier

Current hierarchical test approaches detect the existence of faults in the design and do not aim at identifying faults. For reasons such as design debugging, manufacturing process characterization and yield enhancement, fault diagnosis capabilities [5, 11, 12] are becoming increasingly desirable. Interestingly, despite being able to reach any module for test application purposes, the structure of hierarchical test paths is not always suitable for providing sufficient information for identifying the faulty module. As an example, consider the 8-bit shift-&-add multiplier [4] depicted in figure (2). Although bijective paths for testing each module exist in the design, if a faulty test response is reported through a path, we cannot diagnose whether the fault is in the module under test or in the surrounding modules used in the path. Set theory can be applied in order to combine the information attained from each path; however, in order for such a process to be effective, it needs to address efficiently feedback loops and variable bitwidth signals that are present in the test paths. Moreover, design-for-debug paths need to be introduced when the provided test paths are not able to disambiguate the faulty module.

In section 2, we define the hierarchical test path notion used for establishing bijective functions to and from the module under test. A hierarchical test path example is provided and the debug-related information that can be attained from the outcome of each path is extensively discussed. In section 3, we introduce an algorithm that utilizes this information in order to identify the minimal set of possibly faulty modules under the single faulty module assumption. In section 4, we prove 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. Finally, in section 5, we propose a low-cost, design-for-debug hardware methodology through which the aforementioned condition can be imposed on the design.

2. Hierarchical test paths

In this section, we provide a formal definition and an example demonstrating the hierarchical test path concept. We then examine the debug information that can be attained through hierarchical test paths and discuss their capacity for diagnosing faulty design modules.

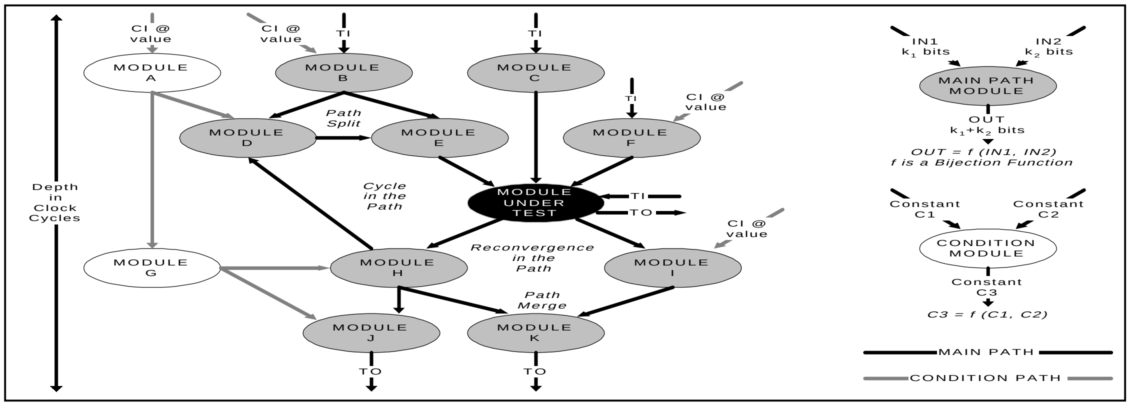

Figure (3): Hierarchical test path definition

2.1. Path definition

Figure (3) shows the hierarchical test path concept for a MUT, which is defined as a 6-tuple:

PATH={*MUT (m, n), Depth, TI (m), TO (n), CI (k), PC (Depth)*}, where

- ***MUT (m, n):*** The *MUT* is the module that is targeted for test using the hierarchical test path. The *MUT* has *m* inputs that need to be justified from primary inputs and *n* outputs that need to be propagated to primary outputs during test application.
- ***Depth:*** The time depth of the path in terms of clock cycles (*Depth* ≥ 1).
- ***TI (m):*** The *m* test inputs. These are primary inputs from which there is a bijection function, through the hierarchical test path, to the *m* inputs of the *MUT*. Each *TI* is defined as a 2-tuple *{PI, t}*, where *PI* is the primary input participating in the bijection function and *t* is the time at which the primary input participates in the bijection function.
- ***TO (n):*** The *n* test outputs. These are primary outputs to which there is a bijection function, through the hierarchical test path, from the *n* outputs of the *MUT*. Each *TO* is defined as a 2-tuple *{PO, t}*, where *PO* is the primary output participating in the bijection function and *t* is the time at which the primary output participates in the bijection function.
- ***CI (k):*** The *k* condition inputs. These are primary inputs that are kept to specific values in order to establish the path. Each *CI* is defined as a 3-tuple *{PI, Value, t}*, where *PI* is the primary input to be held constant, *Value* is the constant to which the *PI* is held and *t* is the time at which the *PI* is held to the *Value*.
- ***PC (Depth):*** The path connectivity model. The *PC* for each clock cycle is a 2-tuple *{Bijections, Conditions}*, where *Bijections* capture the main test path connectivity and *Conditions* capture the condition connectivity necessary for establishing the path. Each *Bijection* is defined as a list of one or more 3-tuples *{Module Inputs, Bijection Function, Module Outputs}*, where the *Module Inputs* are bijected through the *Bijection Function* to the *Module Outputs*. Similarly, each *Condition* is defined as a list of one or more 4-tuples *{Module Inputs, Input Values, Module Outputs, Output Values}*, where the *Module Outputs* attain the *Output Values* when the *Module Inputs* are held to the *Input Values*.

Hierarchical test paths provide a mechanism for accessing and testing a module through the surrounding logic. Through the bijective functions of the path, any test vector can be justified and any test response can be unambiguously propagated to and from the *MUT*. Both combinational and sequential modules are allowed on a path. Reconvergence and cycles are supported and variable bitwidth signals are allowed. Finally, a module may multiply participate across the justification, propagation and condition portions of the path.

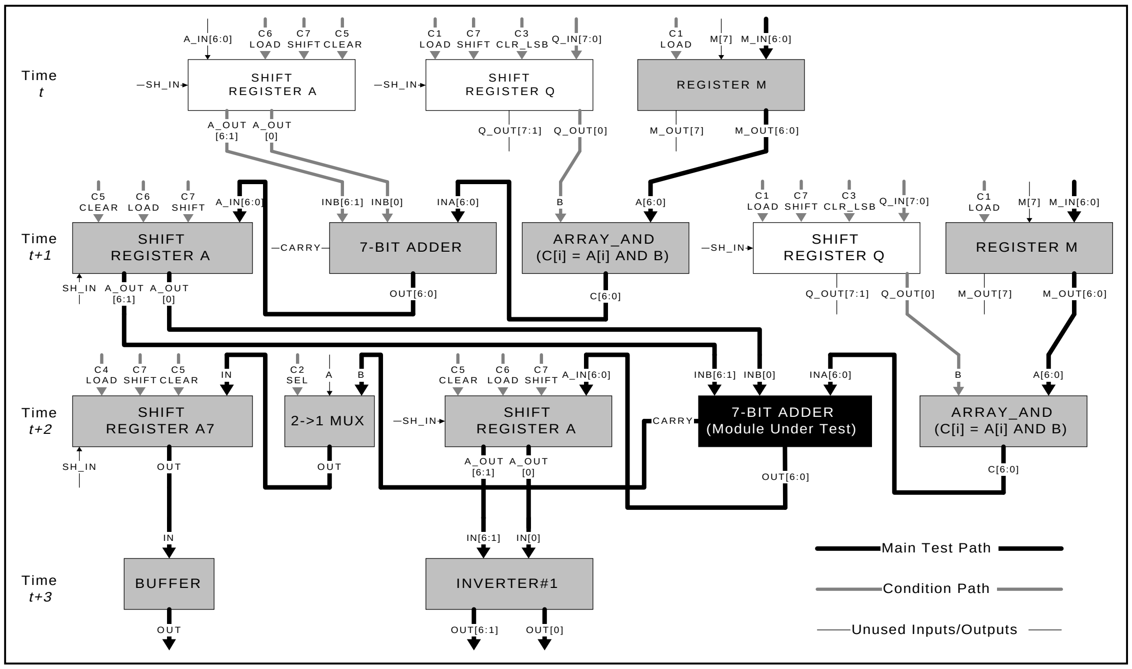

Figure (4): Hierarchical test path for the ADDER module of the example circuit

2.2. Path example

The hierarchical test path for testing the ADDER module in the example circuit of figure (2) is demonstrated in figure (4). In this example the path definition is as follows:

- The *MUT* is the 7-bit ADDER and has *m*=14 inputs and *n*=8 outputs.
- The *Depth* of the path is 4 clock cycles.
- The *m*=14 *TI* (Test Inputs) are the seven least significant inputs of the register M at time *t* and at time *t+1*, that are bijected through the path to the *m*=14 inputs of the ADDER.
- The *n*=8 *TO* (Test Outputs) are the seven outputs of the INVERTER#1 and the output of the BUFFER at time *t+3*, to which the *n*=8 outputs of the ADDER are bijected.
- The path requires *k*=17 *CI* (Condition Inputs) that establish the path:
  - At time *t*: C1=1, C3=0, C5=1, C6=0, C7=0, Q_IN[0]=1
  - At time *t+1*: C1=1, C3=0, C5=0, C6=1, C7=0, Q_IN[0]=1
  - At time *t+2*: C2=1, C4=1, C5=0, C6=1, C7=0
- The *PC* (Path Connectivity) for each clock cycle is depicted in figure (4). For example, at time *t*, there is one *Bijection* and three *Conditions.* The *Bijection* is a 3-tuple (M_IN[6:0], Identity, M_OUT[6:0]), defined on register M, that propagates the inputs M_IN[6:0] identically to the M_OUT[6:0] signal entity. The first *Condition* is the 4-tuple (C5 C6 C7, 100, A_OUT[6:0], 0000000) defined on register A for clearing it. The second *Condition* is the 4-tuple (C1 C3 Q_IN[0], 101, Q_OUT[0], 1) defined on register Q for setting its LSB. The third *Condition* is the 4-tuple (C1, 1, M_OUT[6:0], M_IN[6:0]) defined on register M for loading the test inputs. The *PC* is similarly defined for the remaining clock cycles.

2.3 Debugging information

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, 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 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 example, the path of figure (4), targeting the ADDER module, can also be used for *fully testing* the BUFFER and the INVERTER#1. All inputs and outputs of these modules are on the bijection path and furthermore the ADDER exhibits bijection behavior if one of its inputs is kept constant. Therefore, this path can be used for *fully testing* the ADDER, the BUFFER and the INVERTER#1. 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.

   Furthermore, as a special case, 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. Notice that modules on the common portion of the path cannot be exonerated, since a fault in them may possibly affect only one of the sub-paths.

For example, if we send the test vector set for the ADDER through the path of figure (4) and all responses are correct, then the ADDER is not faulty. In case a fault is reported however, any module on the path (M, Q, A, ARRAY_AND, ADDER, MUX, A7, INVERTER#1, BUFFER) can be the reason. If additionally, the test vectors for INVERTER#1 report a fault, we can exonerate the MUX, A7 and BUFFER, since they are not in the intersection of the cone of logic used for *fully testing* the ADDER and the INVERTER#1.

As an example of the special case, let us assume again that the test vectors for the ADDER report faults but the faulty responses appear only at the output of the INVERTER#1, and never at the output of the BUFFER. In this case, we can safely exonerate the MUX, A7 and BUFFER modules since they are not in the cone of logic driving the faulty outputs.

3. Faulty module diagnosis

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 ∩ All Modules on Path;
           Reduce Candidate_List according to case (ii);}}}
```

Application of the RTL hierarchical testability analysis described in [7] on each module of the circuit of figure (2), revealed the following 11 paths, with the modules that each path can *fully test* in bold & italicized face:

ADDER: Path #1 {A, M, Q, ARRAY_AND, ***ADDER***, ***BUFFER***, A7, MUX, ***INVERTER#1***}
M*:** Path #2 {A, ***M, Q, SIGN, ARRAY_AND, MUX, ADDER, A7, ***BUFFER***, ***INVERTER#1***}
Q: Path #3 {A, M, ***Q***, ARRAY_AND, ADDER, ***INVERTER#2***}
A: Path #4 {***A***, M, Q, ARRAY_AND, ADDER, A7, ***INVERTER#1***}
SIGN: Path #5 {M, Q, ***SIGN***, MUX, ***BUFFER***, A7}
MUX: Path #6 {A, M, Q, ARRAY_AND, ADDER, ***BUFFER***, A7, ***MUX***, SIGN}
A7: Path #7 {M, Q, SIGN, MUX, ***BUFFER***, ***A7***}
BUFFER: Path #8 {M, Q, SIGN, MUX, ***BUFFER***, A7}
INVERTER#1: Path #9 {A, M, Q, ARRAY_AND, ADDER, ***INVERTER#1***}
ARRAY_AND: Path #10 {A, M, Q, ***ARRAY_AND***, ADDER, ***INVERTER#1***}
INVERTER#2: Path #11 {Q, ***INVERTER#2***}

Let us assume that test application through these paths gave the following results:

Path #1: ADDER (***FAULT*** BUT BUFFER OUTPUT OK), INVERTER#1 (OK), BUFFER (OK)
Path #2: M (***FAULT*** BUT BUFFER OUTPUT OK), INVERTER#1 (OK), BUFFER (OK)
Path #3: Q (***FAULT***), INVERTER#2 (OK)
Path #4: A (OK), INVERTER#1 (OK)
Path #5: SIGN (OK), BUFFER (OK)
Path #6: MUX (***FAULT***), BUFFER (***FAULT***)
Path #7: A7 (OK), BUFFER (OK)
Path #8: BUFFER (OK)
Path #9: INVERTER#1 (OK)
Path #10: ARRAY_AND (***FAULT***), INVERTER#1 (***FAULT***)
Path #11: INVERTER#2 (OK)

Based on the outcome of the tests and the connectivity of each hierarchical test path, the faulty module diagnosis algorithm reduces the *Candidate_List* to the 3-element set {ADDER, ARRAY_AND, M}. However, the provided paths have no way of further disambiguating among these 3 modules. The reason is that the paths are constructed in such a way that they cannot fully disambiguate all combinations of test path outcomes. The following section provides a rule for checking if a given set of paths can always diagnose the faulty module.

4. Disambiguation rule

Assuming a single faulty module model, the following observations are used in devising a disambiguation rule for the hierarchical paths. If an arbitrary module *M* is faulty then:

- Paths through which *M* is *fully testable* will report a fault. After examining all these paths, in the worst case the faulty module *Candidate_List* will contain their intersection.
- Paths not using *M* at all will not report a fault. Therefore, we will always be able to exonerate any module that is *fully testable* through these paths.
- Paths on which M is used but is not *fully testable* may or may not report a fault. Since in the worst case a fault will be reported, we cannot rely on such paths for exonerating modules.

Theorem: 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 will always be able to diagnose the faulty module if and only if:

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

Proof: Let us first assume that there is a module *M* for which the above equation does not hold. We will show that there is at least one possible outcome of the paths, based on which we cannot fully disambiguate the faulty module. If the above equation does not hold for module *M*, the right-hand side yields a set with at least two elements, one of which is *M*. Let us assume that module *N*≠*M* is one of the elements of this set. In this case, the two parts of the left-hand side of the equation show that all the paths capable of *fully testing M* contain *N* and that there is no path capable of *fully testing N* without using *M*. Consequently, if all paths capable of *fully testing M* report a fault but no other path does, either *M* or *N* can be faulty.

Let us now examine the case where the equation holds for every module in the design and that *M* is the faulty module. In this case, all paths reporting a fault will contain *M*. Also, no paths that exclude *M* will report a fault and the modules *fully testable* through such paths will all be exonerated. We claim that when we subtract the modules exonerated through the non-faulty paths from the intersection of the modules on the faulty paths we will always get the one-element set {*M*} and we prove this claim by contradiction. Let us assume that there is a second element in this set, module *N*. This means that *N* exists on all faulty paths and that there is no good path capable of *fully testing N*. We also know that *M* exists on all faulty paths and that there is no good path capable of *fully testing M*. From these two pieces of information it can be concluded that modules *M* and *N* are either always together on a path, or if they are not together on a path, neither one can be *fully tested* through this path. Consequently, if we apply the equation for either module *M* or module *N*, the left-hand side will yield at least the two-element set {*M*, *N*}. But this contradicts the initial assumption that the equation holds for all design modules. Therefore, for any faulty module M the algorithm of section 3 always provides the one-element set {*M*} for any combination of test path outcomes.

5. Design-for-debug

In this section, we propose four design-for-debug techniques that introduce additional paths to the design to impose the disambiguation rule. The first technique is called *path augmentation* and utilizes test path fan-out to unused primary outputs in order to augment the path. As shown in figure (5), we search for path modules that have fan-out to non-path primary outputs, through non-path modules. In this case, we can reduce the ambiguity set *S (A)* of each module $A \in P2$, that is not part of the cone of logic *P1* driving the fan-out module. All modules in P1 are eliminated from the ambiguity set S (A). This technique requires only a number of new conditions but no additional hardware and therefore is low-cost and non-intrusive.

The second technique is called *condition checking* and utilizes simple comparators in order to check the condition modules. The technique requires one comparator for each path condition as shown in figure (6). We connect the output of each condition module to one of the comparator inputs and tie the second input to the expected condition value. The outputs of the comparators are connected to a "parallel in / serial out" register. Correctness information for each condition module is captured in the register and subsequently shifted out through a primary output. The Load/Shift input of the register is controlled through a primary input. These are the only two pins that the scheme requires. The basic principle of *condition checking* is similar to *path augmentation*. Additional observability information is obtained through a fan-out node, only this time observability is achieved through hardware and is applicable only for condition modules. Ambiguity sets are reduced according to the same principles used in *path augmentation*. Although *condition checking* requires hardware modifications, it is non-intrusive since no fault in the added hardware will affect the functionality of the circuit.

The third technique, called *path probing*, explicitly introduces observability at the outputs of the modules in the ambiguity sets. The technique is depicted in figure (7), through an example. The *TO* (Test Outputs) of the path are multiplexed with fan-out signals coming from each module in the ambiguity set. The control signals are provided to the multiplexer through a "serial in / parallel out" register that is loaded from an external pin. This is the only pin that the technique requires. Through the register we can probe the hierarchical test path at the outputs of each module on the ambiguity set. Therefore, if the ambiguity set contains *n* modules, we can diagnose the faulty module in O(log(n)) test applications. *Path probing* is very efficient but also intrusive. The multiplexer introduces additional delay and any fault in it affects the circuit functionality. O(n) fan-out connections are required, where *n* is the size of the ambiguity set, but this cost can be amortized among the modules of all ambiguity sets.

The fourth technique is called *path reconstruction* and is demonstrated in figure (8). This technique combines portions of existing paths to provide additional paths. Using multiplexers and a simple control scheme similar to the one used in *condition checking*, we reconstruct the paths, including or excluding modules as necessary for imposing the rule. The challenge now is to to identify the minimum number and the appropriate locations to place the multiplexers. In [3], an efficient fault isolation mechanism that embeds error correction capabilities into test tracks during synthesis is described. Based on the *Hamming* and *Mirror-Hamming* codes [3], a O(log(n)) number of tracks are constructed so that any faulty module will always be diagnosed. We utilize the same approach for addressing this challenge and guiding path reconstruction.

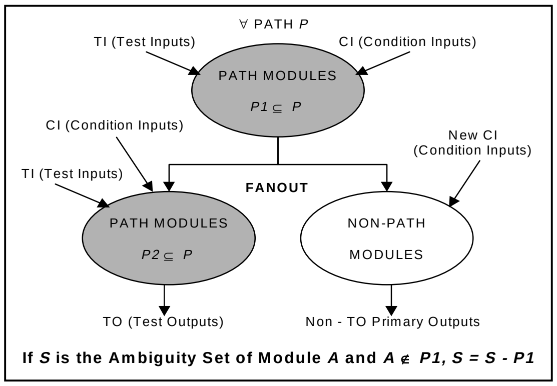

Figure (5): Path augmentation

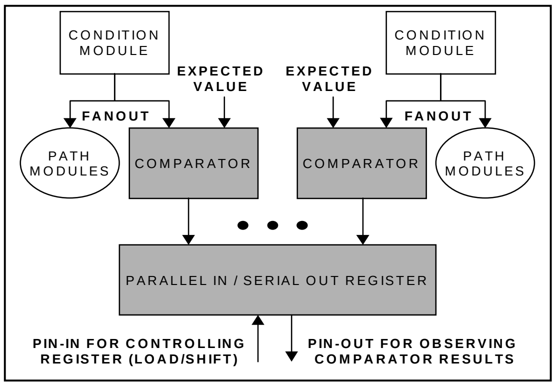
CONDITION MODULE
EXPECTED VALUE
EXPECTED VALUE
CONDITION MODULE
FANOUT
FANOUT
PATH MODULES
COMPARATOR
COMPARATOR
PATH MODULES
PARALLEL IN / SERIAL OUT REGISTER
PIN-IN FOR CONTROLLING REGISTER (LOAD/SHIFT)
PIN-OUT FOR OBSERVING COMPARATOR RESULTS

Figure (6): Condition checking

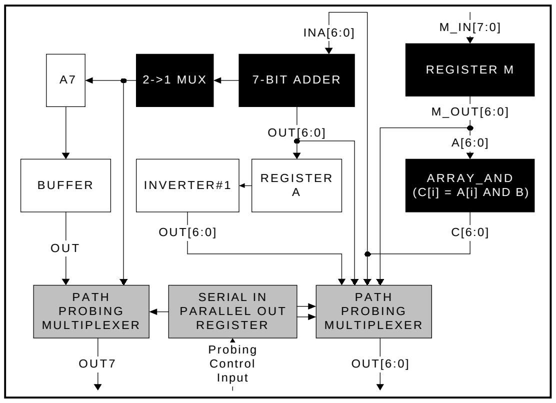

Figure (7): Path probing

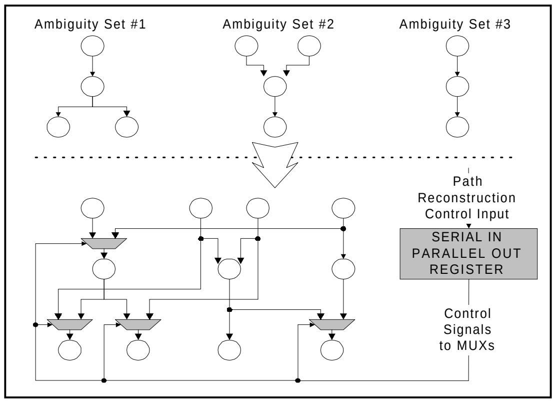

Figure (8): Path reconstruction

6. Conclusion

We introduced a faulty module diagnosis and design-for-debug methodology for hierarchical designs, based on bijective test paths. We defined the hierarchical test path notion and extensively discussed the debug-related information that such paths may provide in order to assist faulty module diagnosis. We devised an algorithm for identifying faulty module candidates in a design, according to the test application outcome of the hierarchical test paths. Subsequently, we proved a necessary and sufficient condition, relating the modules on the test paths, so that the faulty module can always be disambiguated. Finally, we described a low-cost, design-for-debug hardware insertion methodology for imposing the disambiguation rule on the hierarchical test paths. The aforementioned elements compose a unified framework for efficient modular test application and faulty module diagnosis in hierarchical designs.

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] S. N. Hamilton, A. Orailoğlu, "Microarchitectural Synthesis of ICs with Embedded Concurrent Fault Isolation", Proceedings of the 27th Fault Tolerant Computing Symposium, pp. 329-338, 1997.
[4] J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, 3rd Edition, 1998.
[5] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, J.-Y.J. Lu, "Fault-Simulation Based Design Error Diagnosis for Sequential Circuits", Proceedings of the 35th ACM/IEEE Design Automation Conference, pp. 632-637, 1998.
[6] J. Lee, J. H. Patel, "Hierarchical Test Generation under Architectural Level Functional Constraints", *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1144-1151, 1997.
[7] 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.
[8] Y. Makris, A. Orailoğlu, "DFT Guidance Through RTL Test Justification and Propagation Analysis", Proceedings of the International Test Conference, pp. 668-677, 1998.
[9] Y. Makris, J. Collins, A. Orailoğlu, P. Vishakantaiah, "TRANSPARENT: A System for RTL Testability Analysis, DFT Guidance and Hierarchical Test Generation", Proceedings of the Custom Integrated Circuits Conference, pp. 159-162, 1999.
[10] B. T. Murray, J. P. Hayes, "Hierarchical Test Generation Using Precomputed Tests for Modules", *IEEE Transactions on CAD of Integrated Circuits and Systems,* vol. 9, no. 6, pp. 594-603, 1990.
[11] M. Potkonjak, S. Dey, K. Wakabayashi, "Design-for-Debugging of Application Specific Designs", Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 295-301, 1995.
[12] *Practical Aspects of IC Diagnosis and Failure Analysis: A Walk through the Process,* IEEE Computer Society Press, Proceedings of the International Test Conference, Lecture Series II, 1996.
[13] R. S. Tupuri, J. A. Abraham, "A Novel Test Generation Method for Processors using Commercial ATPG", Proceedings of the International Test Conference, pp. 743-752, 1997.
[14] P. Vishakantaiah, J. A. Abraham, M. S. Abadir, "Automatic Test Knowledge Extraction From VHDL (ATKET)", Proceedings of the 29th ACM/IEEE Design Automation Conference, pp. 273-278, 1992.