Efficient Transparency Extraction and Utilization in Hierarchical Test*

Yiorgos Makris
Electrical Engineering Department
Yale University
New Haven, CT 06520

Vishal Patel, Alex Orailoğlu
Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA 92093

Abstract

We introduce a methodology for identifying transparency behavior appropriate for hierarchical test, based on the theoretical principles of transparency composition. Unlike high level approaches that identify limited, coarse transparency behavior, the proposed methodology is capable of extracting a wide class of fine grained transparency functions for arbitrary sub-word bit clusters. The functions in this class can furthermore be rapidly extracted on the fly and efficiently utilized for hierarchical test translation, thus alleviating the exponential extraction time and storage space requirements of exhaustive approaches. The twin benefits of rapid, automated extraction coupled with the expansion of utilizable transparency scope deliver reduced DFT while enabling cost-effective hierarchical test of high quality.

1. Introduction

Ever since in response to the rapidly growing complexity of circuits, high level test approaches made an appearance, they have been confronted with criticisms as to their effectiveness, accuracy and practicality. While raising the level of abstraction renders a rich trade-off space between complexity and efficiency of the pertinent methodologies, the ability of high-level test approaches to exploit this space has been constantly disputed. Deep skepticism typically surrounds high level testability analysis, DFT, and test generation.

In an effort to address this skepticism, hierarchical test [5,7,8,10] has been proposed as a *hybrid* scheme, wherein low-level methods are utilized locally for each design module, while high-level approaches are employed to provide accessibility to the module boundary. Efficiency and complexity issues are thus both addressed through this divide and conquer scheme. Nevertheless, hierarchical test is considered expensive, not because of test generation time but rather due to heightened DFT requirements for module accessibility. A significant portion of such DFT may be unnecessary and is attributed to the limitations of current high level methods in identifying accessibility behavior inherently available in the design.

The current concept of transparency behavior, through which module access is typically established, constitutes a key factor of the limited success of hierarchical test. While transparency has been thoroughly studied and defined theoretically [1,2,6,9,11], only a limited subset is utilized in practice [4,5,7,10]. This discrepancy amplifies the voices of concern regarding the frugality of the resultant DFT when hierarchical test approaches are applied [4,7,12]. In order to address these concerns, we propose in this paper an efficient transparency extraction and utilization method that bridges the gap between the theoretical definition of transparency and its practical applications.

2. Motivation

Hierarchical test relies on transparency paths in order to access and test each module. Mathematically, *surjective* functions are required for justifying test vectors to the module under test, while *injective* functions are necessary in order to propagate test responses from the outputs of the module under test. Surjective and injective paths are referred to in the literature as *S-Paths* and *F-Paths* respectively [2], while *bijective* paths satisfying both properties are referred to as *T-Paths* [1]. Several variations of the basic surjective, injective, and bijective functions, including *Ambiguity Sets* [9], *Transparency Modes* [11], and *Transparency Channels* [6], have also been used.

Although in theory any surjective, injective or bijective function is appropriate for hierarchical test, only a limited class of simple functions is utilized in practice [4,5,7,10]. The most widely used function is the *Identity* function, referred to as *I-Path* [1], which along with the *Inversion* function is cognitively identified for RTL modules. The functionality of arithmetic modules is also widely used, although in practice it also degrades to the identity function (addition by '0', multiplication by '1', etc.). In short, a large class of inherent transparency is not being exploited, resulting in excessive DFT to establish accessibility. But what causes this discrepancy between the theoretical definition of transparency and what is used in practice?

Practical usage of a transparency function is determined by two factors, namely identification complexity and utilization efficiency. Gate level transparency extraction is computationally NP-hard [3], limiting exhaustive algorithms to small circuits. Therefore, transparency is typically identified from high-level HDL. This cognitive approach limits the scope to coarse signal entities and

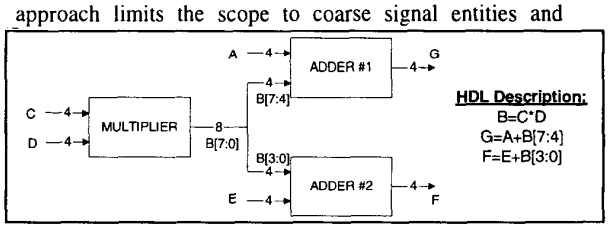

Figure (1): *Example Circuit*

* This work is supported in part through a research grant from Intel Corporation and the University of California MICRO program.

operators of the HDL, identifying only a few, word-level transparency functions. As an example, consider ADDER #1 in the circuit of figure (1). A simple cognitive approach would identify in the HDL two bijective identity functions, *G=A if B[7:4]='0'* and *G=B[7:4] if A='0'*. A more elaborate approach would possibly extend the class to the 32 functions *G=A+k if B[7:4]='k', k=0...15* and *G=B[7:4]+k if A='k', k=0...15*. However, there exist more 4-bit bijections through the adder that are not utilized. For example, any 4 bits obtained by choosing either the A or the B input for each bit position, biject to the 4 output bits for every constant value on the remaining 4 inputs, adding up to 256 bijections. While any one of these bijections is equally appropriate for traversing through the adder, cognitive approaches limit their scope to only 32 of them.

In addition, cognitive approaches are limited to word-level transparency, which is neither always inherently available nor always required. For example, in the module MULTIPLIER of figure (1) there exists no inherent bijection function that can justify all possible values at B[7:0]. Yet there exist several 4-bit bijections through the multiplier, which are adequate for justifying vectors to the rest of the circuit, such as *B[3:0]=D if C='1'*, which justifies vectors to ADDER#2. Furthermore, there exist several bijections of width less than 4 bits. In short, depending on the design connectivity, fine grained transparency functions may constitute an important portion of test paths, yet are omitted by cognitive approaches.

The usability of a transparency function in hierarchical test is also affected by the efficiency of its utilization in test translation. Since each local vector needs to be translated into global design test, a compact representation of the transparency functions is necessary. Otherwise, large translation tables will be required, resulting in excessive storage and search time for performing test translation. For example, in the circuit of figure (1), the 32 cognitively identified bijections may be stored compactly as arithmetic operations through which test translation can be performed. Storing arbitrary 4-bit bijections, however, requires tables of 16 4-bit entries. For a large number of wide bijections, table storage and translation time prove cumbersome.

While utilizing all possible transparencies is practically infeasible, the class of word level functions currently used imposes significant limitations on hierarchical test. In this paper, we propose a methodology that examines for the first time the inter-cell connectivity of the module in order to extract a wide class of transparency functions that can be efficiently utilized for test translation. In section 3, we identify a set of sufficient conditions for transparency composition, based on which a transparency extraction algorithm is devised in section 4. Utilization of these transparency functions for test translation is discussed in section 5 and experimental data is provided in section 6.

3. Transparency Composition

We propose a transparency composition method based solely on function classes and not the actual functions. We distinguish functions into four classes according to their inherent transparency behavior and we examine transparency composition through combinations of classes. Consider a function implemented by a cell, as shown in figure (2)(a). The cell has a set of function inputs *FI* and an additional set of condition inputs *CI* that activate the cell function. Additionally, the cell has a set of function outputs, *FO*, and an additional set of collateral outputs, *CO*, whose value may be either constant or variable for the values of *FI* and *CI*. Such a function could be defined for example on a full adder cell, with *FI={A}, CI={B, Cin}, FO={Z}*, and *CO={Cout}*. Functions implemented by such cells are not necessarily independent of each other, as the possible interconnection structures of figure (2)(b) reveal. Furthermore, this dependence, which we refer to as *implication*, can be bi-directional. We categorize the function of figure (2)(a) into one of four classes, based on whether it is bijective and on how the implication through the *CIs* affects the bijection. In the following definitions, S_k and S_l are the sets of all possible values of k-bit and l-bit signals, respectively.

Type #1: The function is bijective and the shape of the bijection is independent of the implication through *CIs*.

$$\bigcup_{\forall x \in S_k} f(x) = S_k$$

Type #2: The function is bijective for each constant value on the *CIs*, but the bijection depends on the constant.

$$\forall y \in S_l : \bigcup_{\forall x \in S_k} f(x, y) = S_k$$

Type #3: The function is bijective for some but not all constant values on the *CIs*; the bijection may depend on the constant. There exists at least one constant value for which the function is not bijective.

$$\exists y \in S_l : \bigcup_{\forall x \in S_k} f(x, y) = S_k \wedge \exists y \in S_l : \bigcup_{\forall x \in S_k} f(x, y) \subset S_k$$

Type #4: No constant value on the CIs makes the function bijective.

$$\forall y \in S_l : \bigcup_{\forall x \in S_k} f(x, y) \subset S_k$$

Examples of the four function types are given in figure (3). The question now is what conclusions can be drawn about functions composed for each combination of the above types. The answer is synopsized in Table (1), where we can see that the composition of two Type #1 functions, or a Type #1 and a Type #2 function always produces a

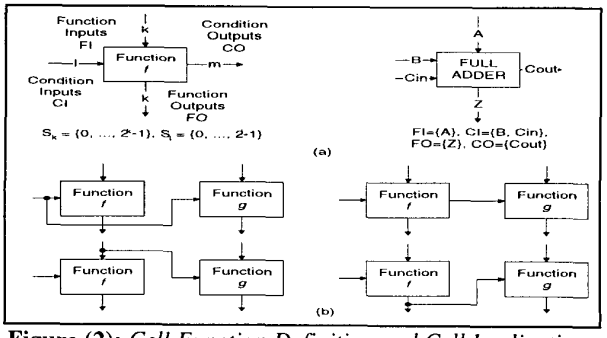

Figure (2): *Cell Function Definition and Cell Implications*

	Type #1	Type #2	Type #3	Type #4
Type #1	*(B) Always Bijective*	*(B) Always Bijective*	*(M) Maybe Bijective*	*(N) Never Bijective*
Type #2	*(B) Always Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*
Type #3	*(M) Maybe Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*
Type #4	*(N) Never Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*	*(M) Maybe Bijective*

Table (1): *Bijection Composition Possibilities*

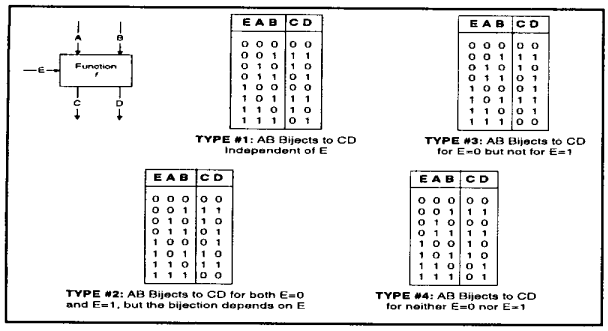

Figure (3): *Examples of Function Types*

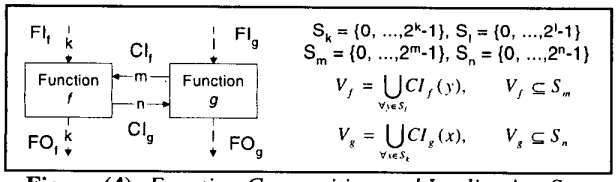

Figure (4): *Function Composition and Implication Sets*

bijection; interestingly, the combination of a Type #1 and a Type #4 function can be shown never to produce a bijection, while the combination of two Type #4 functions may turn out bijective! Theorems (1) through (3) prove some of the aforementioned observations. All other combinations may or may not produce a bijection. This is exactly where the exponential nature of the problem lies, since the only way to identify such bijections is to exhaustively examine the composed function. In order to expand the set of utilizable bijections, we devise a set of conditions, which can be easily verified without examining the composed bijection and under which the composed function is guaranteed to be bijective. These conditions are provided through the following theorems (4) and (5) and corollaries (1) and (2). Consider the two functions, f and g, of figure (4), where V_f is the set of implicated values from g to f and V_g is the set of implicated values from f to g.

Theorem (1): Composition of two functions of Type #1 always produces a bijection.

Proof: If f and g are both of Type #1, then $\bigcup_{\forall x \in S_k} f(x) = S_k$ and $\bigcup_{\forall y \in S_l} g(y) = S_l$. The cross product results in $\bigcup_{\forall x,y \in S_k \times S_l} f(x) \times g(y) = S_k \times S_l$, which implies $\bigcup_{\forall x,y \in S_k \times S_l} f \times g(x,y) = S_k \times S_l$, thus proving composition to a bijective function.

Theorem (2): Composition of a Type #1 and a Type #2 function always produces a bijection.

Proof: If f is of Type #1 and g is of Type #2, then $\bigcup_{\forall x \in S_k} f(x) = S_k$ (1) and $\forall x \in S_n : \bigcup_{\forall y \in S_l} g(x,y) = S_l$. But $V_g = \bigcup_{\forall x \in S_k} CI_g(x)$, $V_g \subseteq S_n$, which implies $\forall x \in S_k : \bigcup_{\forall y \in S_l} g(x,y) = S_l$ (2). Combining (1) and (2) results in $\bigcup_{\forall x,y \in S_k \times S_l} f(x) \times g(x,y) = S_k \times S_l$, which implies $\bigcup_{\forall x,y \in S_k \times S_l} f \times g(x,y) = S_k \times S_l$, thus proving composition to a bijective function.

Theorem (3): Composition of a Type #1 and a Type #4 function always produces a non-bijective function.

Proof: If f is of Type #1, and g is of Type #4, then $\bigcup_{\forall x \in S_k} f(x) = S_k$ (3) and $\forall x \in S_n : \bigcup_{\forall y \in S_l} g(x,y) \subset S_l$. But $V_g = \bigcup_{\forall x \in S_k} CI_g(x)$, $V_g \subseteq S_n$, which means that $\forall x \in S_k : \bigcup_{\forall y \in S_l} g(x,y) \subset S_l$ (4). Combining (3) and (4) results in $\bigcup_{\forall x,y \in S_k \times S_l} f(x) \times g(x,y) \subset S_k \times S_l$, which implies $\bigcup_{\forall x,y \in S_k \times S_l} f \times g(x,y) \subset S_k \times S_l$, thus proving that the composition results in a non-bijective function.

Theorem (4): Composition of a Type #1 function f and a Type #3 function g produces a bijection if g is bijective $\forall z \in V_g$, where V_g is the set of values implicated from f to g.

Proof: If f is a function of Type #1 and g is a function of Type #3, then $\bigcup_{\forall x \in S_k} f(x) = S_k$ (5) and $\exists z \in S_n : \bigcup_{\forall y \in S_l} g(z,y) = S_l \wedge \exists z \in S_n : \bigcup_{\forall y \in S_l} g(z,y) \subset S_l$ (6)

Although $V_g \subseteq S_n$, given $z \in V_g$, no inference can be drawn regarding the possible equality of $\bigcup_{\forall y \in S_l} g(z,y)$ to S_l. However, if according to the hypothesis g is bijective $\forall z \in V_g$, then $\forall z \in V_g : \bigcup_{\forall y \in S_l} g(z,y) = S_l$. But $V_g = \bigcup_{\forall x \in S_k} CI_g(x)$, so $\forall x \in S_k : \bigcup_{\forall y \in S_l} g(x,y) = S_l$ (7)

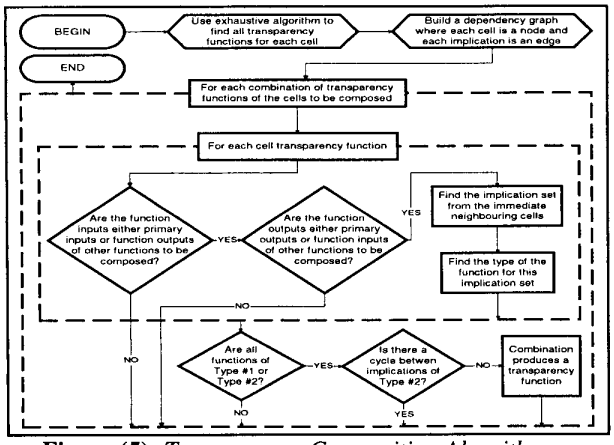

Figure (5): *Transparency Composition Algorithm*

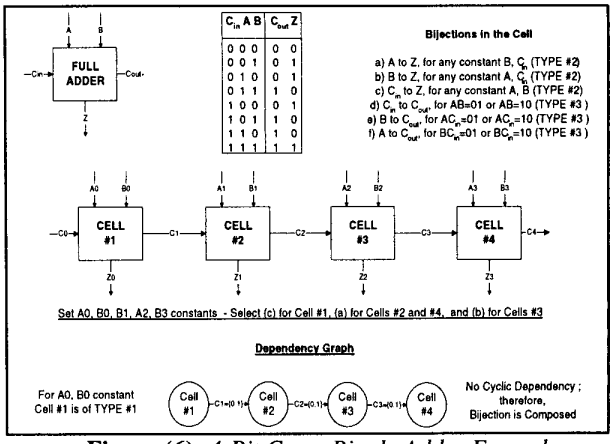

Figure (6): *4-Bit Carry Ripple Adder Example*

The combination of equations (5) and (7) implies that $\bigcup_{\forall x,y \in S_k \times S_l} f(x) \times g(x,y) = S_k \times S_l$, so $\bigcup_{\forall x,y \in S_k \times S_l} f \times g(x,y) = S_k \times S_l$, and therefore the composition is bijective. This condition is only sufficient but not necessary, because the implication set V_g may be further pruned if more than two functions are composed. Finding the exact implication set requires costly exhaustive examination of the composed function.

Theorem (5): Composition of a set of two or more functions of Type #1 or Type #2 is bijective if the implication between them is acyclic.

Proof: Since Type #1 functions are independent they cannot be part of a cyclic implication. If the implication between Type #2 functions is acyclic, there is at least one Type #2 function f that has implications only from Type #1 functions $g_1, \ldots, g_k$. Assume that the width of f is l and the widths of $g_1, \ldots, g_k$ are $m_1, \ldots, m_k$, respectively. Since f is of Type #2,

$$\forall y_1, \ldots, y_k \in S_{m_1} \times \ldots \times S_{m_k} : \bigcup_{\forall x \in S_l} f(x, y_1, \ldots, y_k) = S_l \quad (8)$$

Since, $g_1, \ldots, g_k$ are of Type #1, we know that $\forall i \in 1 \ldots k$, $\bigcup_{\forall y \in S_{m_i}} g_i(y) = S_{m_i}$ (9). Combination of (8) and (9) implies

$$\bigcup_{\forall x, y_1, \ldots, y_k \in S_k \times S_{m_1} \times \ldots \times S_{m_k}} f \times g_1 \times \ldots \times g_k(x, y_1, \ldots, y_k) = S_k \times S_{m_1} \times \ldots \times S_{m_k}$$

Therefore, the Type #2 function f and the Type #1 functions $g_1, \ldots, g_k$ compose into a Type #1 function $f \times g_1 \times \ldots \times g_k$. As a result, the implications from the Type #2 function, f, to other Type #2 functions now reduce to implications from the composed Type #1 function to Type #2 functions. In addition, since the implication between Type #2 functions is acyclic, there is again at least one Type #2 function that has implications only from Type #1 functions. Therefore, the above process is repeated until all functions are composed into a Type #1 function, resulting in a bijection. This condition is again only sufficient but not necessary.

Corollary (1): Composition of a Type #2 function f and a Type #3 function g produces a bijection if g is bijective $\forall z \in V_g$, where V_g is the set of values implicated from f to g and the implication between the functions is acyclic.

Corollary (2): Composition of two Type #3 functions f and g produces a bijection if g is bijective $\forall z \in V_g$, where V_g is the set of values implicated to g, f is bijective $\forall w \in V_f$, where V_f is the set of values implicated to f, and the implication between the functions is acyclic.

Theorems (1) through (5), along with corollaries (1) and (2), provide a powerful mechanism for reasoning on transparency composition, based on which an efficient extraction algorithm for a wide class of transparency functions is devised in the following section.

4. Transparency Extraction Methodology

Composition of Type #1, Type #2, and Type #3 functions can be guaranteed to produce a bijection if the associated conditions are satisfied. Under these conditions, Type #3 functions reduce to either Type #1 or Type #2 for the implicated set of values, and the implication between Type #2 functions is acyclic. Checking the conditions is simple, facilitating an efficient transparency extraction algorithm shown in figure (5). As mentioned earlier, the conditions are only sufficient but not necessary; therefore, not all possible bijections are identified. Type #4 functions are also omitted, since they are inherently non-bijective for any constant implication from the surrounding functions; bijection could only be established through exhaustive analysis of the composed function in this case. Yet the algorithm is capable of extracting a wide class of bit-cluster level transparency functions that are not cognitively obvious, as the following examples demonstrate.

The first example circuit, shown in figure (6), is a simple 4-bit carry-ripple adder on which a bijection that is cognitively not obvious is identified. The basic cell and its inherent bijections are shown, as well as the 4-bit adder circuit and the dependency graph. Assume that we are interested in finding a bijection to propagate inputs {A3, B2, A1, C0} through the adder. This points to the composition of bijection (a) for cell #4, bijection (b) for cell #3, bijection (a) for cell #2, and bijection (c) for cell #1. The function inputs of all four bijections are primary inputs of the circuit and the function outputs are all primary outputs. For constant {A0, B0, B1, A2, B3}, bijection (c) of cell #1 reduces to a Type #1 function, since there is no

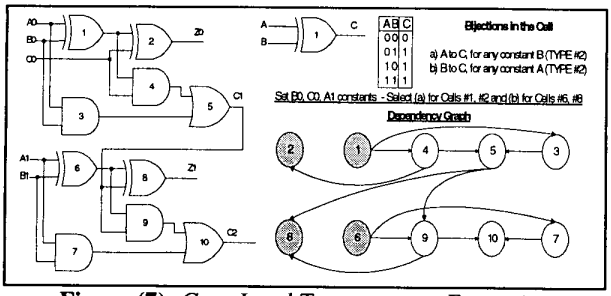

Figure (7): *Gate-Level Transparency Extraction*

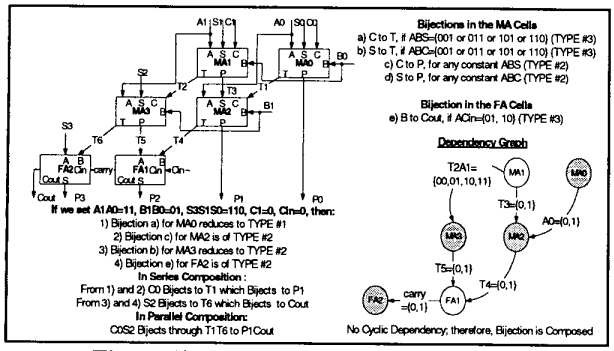

Figure (8): *Multiply-Add Circuit Example*

variable implication to the cell. The implication from the immediate neighbors to cells #2 through #4 means that the corresponding bijections remain of Type #2. Since all the functions to be composed are of Type #1 or Type #2 and there is no cyclic implication in the dependency graph, the composition is a bijection.

Even though the proposed methodology has been illustrated by utilizing cells, it does not fundamentally rely on them as a cell may be construed to map to a single gate. An example is shown in figure (7), where we demonstrate the extraction of a bijection on a 2-bit carry ripple adder. Under constant {B0, C0, A1}, the bit-cluster {B1, A0} bijects to the bit-cluster {Z1, Z0}. The XOR gate has two Type #2 bijections, and selecting bijection (a) for cell #1 and cell #2, and bijection (b) for cell #6 and cell #8, leads to the dependency graph shown. Cell #1, cell #2, and cell #6 are of Type #1, while cell #8 is of Type #2 due to the variable implication from cell #5. Since the graph is acyclic, a bijective function is composed. Composition is performed both in parallel and in series, as for example for bijection (a) of cell #1 and bijection (a) of cell #2.

The size of the cells, however, has a substantial impact on the results. Small cells result in very fast exhaustive identification of their transparencies but the algorithm has to operate on a large number of primitives. Furthermore, since the algorithm is not exhaustive, as the number of primitives increases, fewer transparency functions are identified. At the other extreme, a few very large cells require a costly exhaustive transparency identification for each cell, but the number of primitives becomes very small. The basic cell, comprising a relatively small number of gates and appearing repetitively in a circuit, constitutes a highly desirable trade-off point for the size of the exhaustive transparency identification problem and the number of primitives for the composition algorithm.

The algorithm relies neither on homogeneity nor on regularity and is similarly applicable to heterogeneous and irregular circuits. Figure (8) shows such an example on a 2-bit multiply-add circuit that comprises two different types of cells and irregular interconnection. Through the illustrated basic cell functions, a bijection is composed from bit cluster C0S2 to bit cluster P1Cout, in series and in parallel, according to the algorithm outlined above.

While the algorithm and the examples have been based on bijective functions, the methodology is readily extendible to surjective and injective functions. Bijections and surjections can be combined in order to compose surjections, while bijections and injections can be combined in order to compose injections. In short, the algorithm identifies a wide range of fine-grained transparency functions composed from cell functions, providing hierarchical test approaches with a rich space of rapidly extractable, fine-grained transparency.

5. Transparency Utilization for Test Translation

While the above methodology addresses transparency extraction, we still need to answer the question of how local to global test translation is performed through the extracted functions. If a k-bit bijection is employed in a hierarchical test path, a 2^k-entry table needs to be stored on which a search has to be performed for every vector. For large values of k, both the storage space and the search time required are inordinate and therefore in practice only simple functions with closed form descriptions are used. A modified ATPG algorithm could alternatively be employed to perform test translation without storing the function tables [7,10], but ATPG is also computationally complex.

In the proposed class of transparency functions, test translation is significantly simplified. Since transparency is composed from cell functions, we only need to store these small functions and search within the small tables in a fast ripple-like fashion. Starting from the cells that exhibit Type #1 functions, we traverse the dependency graph in a breadth-first manner to find the bits of the vector to be translated corresponding to each cell. We then search in the small cell function tables for the appropriate input values. The table entry also pinpoints the implication to the next cells in the dependency graph. This information is used along with the desired output bits corresponding to the next cell in order to find the corresponding input values; the process continues in a ripple-like manner. In contrast to the arbitrary bijection case, we thus reduce both the storage and the search time required for the composed bijection. Test translation within the outlined class of transparency functions constitutes a fast and efficient methodology, fundamentally reliant only on the size of the cell functions.

A test translation example is given in figure (9). Assume that we want CDJKOPTU=01101101. In order to store the 8-bit bijection, a 256-entry table is required, in which a search has to be performed. With our approach,

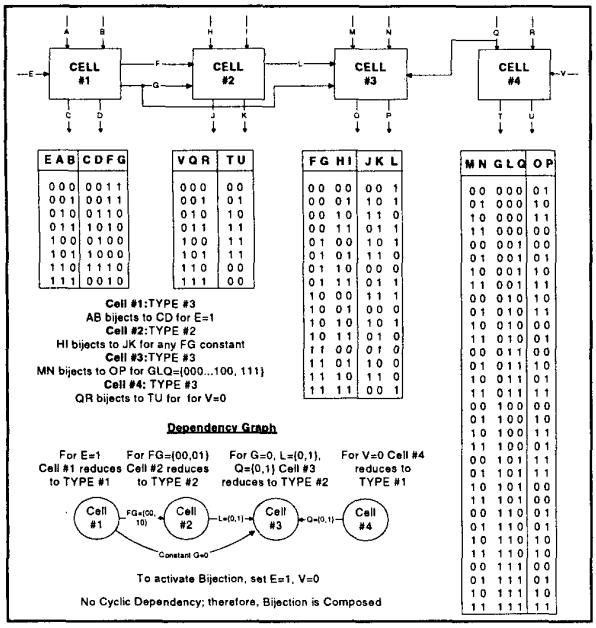

Figure (9): *Test Translation Example*

only the four tables of the much smaller cell functions need to be stored and the translation proceeds as follows: we want CD=01 and CD is produced by cell #1, so we search the cell #1 table and we find AB=00, which sets FG=00. With FG=00 we search the cell #2 table to get JK=01 and we find HI=01, which gives L=1. Cell #4 produces TU, and we want TU=01, so we search the cell #4 table and we find QR=01 which gives Q=0. With Q=0 and L=1 and G=0, we search the cell #3 table to get OP=11, and find MN=01. The final translation is ABHIMNQR=00010101.

6. Experimental Results

The proposed methodology, as well as an exhaustive gate-level bijection extractor have been implemented and applied on several circuits and the results compared in Table (2). As shown, the time required by the exhaustive method grows rapidly as the circuit bitwidth increases. Furthermore, the large number of bijections requires inordinate storage. The proposed methodology, on the other hand, identifies most of the bijections in a very short time and with negligible storage, since only cell bijections are stored from which the rest is composed on the fly. For the adder circuit the proposed method rapidly identifies all possible bijections. While not all possible bijections are identified for the multiply-adder and the random example circuit of figure (9), the method composes a very large class of bijections fast and with minimal storage needs.

	Exhaustive Extraction		Proposed Method	
	Bijections	Time (sec)	Bijections	Time (sec)
Adder-4	16,080	5.1	16,080	3.4
Adder-6	1,220,544	851.3	1,220,544	43.7
Muladd-2	47,750	36.6	34,247	12.6
Muladd-3	5,415,937	40,640.2	3,927,542	1072.9
Figure (9)	64,046	51.2	42,746	15.7

Table (2): *Comparative Experimental Results*

7. Conclusion

Cost-effective hierarchical test requires identification and utilization of inherent module accessibility behavior before resorting to expensive DFT. Given the NP-hard nature of gate-level transparency extraction, current practice limits the scope of transparency to a very small set of functions that can be extracted from high level HDL descriptions. However, such coarse, cognitively identified functions are neither always available nor consistently required. The methodology described in this paper provides a radically new approach for transparency extraction. Based on a theoretical analysis of transparency composition, the proposed algorithm extracts on the fly a wide class of transparency functions. Unlike the transparency functions identified by current approaches, this class comprises a large number of fine-grained bijective, surjective, and injective functions that are not necessarily cognitively straightforward. These functions are efficiently utilized for test translation, providing hierarchical test methods with a rich space of rapidly extractable and inexpensively utilizable transparency.

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 JSSC*, vol. 23, no. 2, pp. 421-427, 1988.

[3] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, pp. 259-263, W. H. Freeman, 1979.

[4] I. Ghosh, A. Raghunathan, N. K. Jha, "A Design For Testability Technique for RTL Circuits using Control/Data Flow Extraction", *IEEE Trans. on CAD*, vol. 17, no. 8, pp. 706-723, 1998.

[5] J. Lee, J. H. Patel, "Hierarchical Test Generation under Architectural Level Functional Constraints", *IEEE Trans. on CAD*, vol. 15, no. 9, pp. 1144-1151, 1997.

[6] Y. Makris, A. Orailoğlu, "RTL Test Justification and Propagation Analysis for Modular Designs", *JETTA*, vol. 13, no. 2, pp. 105-120, 1998.

[7] Y. Makris, J. Collins, A. Orailoğlu, P. Vishakantaiah, "TRANSPARENT: A System for RTL Testability Analysis, DFT Guidance and Hierarchical Test Generation", *CICC*, pp. 159-162, 1999.

[8] B. T. Murray, J. P. Hayes, "Hierarchical Test Generation Using Precomputed Tests for Modules", *IEEE Trans. on CAD*, vol. 9, no. 6, pp. 594-603, 1990.

[9] B. T. Murray, J. P. Hayes, "Test Propagation through Modules and Circuits", *ITC*, pp. 748-757, 1991.

[10] R. S. Tupuri, A. Krishnamachary, J. A. Abraham, "Test Generation for Gigahertz Processors using an Automatic Functional Constraint Extractor", *DAC*, pp. 647-652, 1999.

[11] P. Vishakantaiah, J. A. Abraham, M. S. Abadir, "Automatic Test Knowledge Extraction From VHDL (ATKET)", *DAC*, pp. 273-278, 1992.

[12] P. Vishakantaiah, T. Thomas, J. A. Abraham, M. S. Abadir, "AMBIANT: Automatic Generation of Behavioral Modifications for Testability", ICCD, pp. 63-66, 1993.