Locating Logic Design Errors via Test Generation and Don't-Care Propagation

Sy-Yen Kuo
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Abstract

This paper presents a new technique, the don't-care propagation method, for logic verification and design error location in a circuit. Test patterns for single stuck-line faults are used to compare the gate-level implementation of a circuit with its functional-level specification. In the presence of logic design errors, such a test set will produce responses in the implementation that disagree with the responses in the specification. In the verification phase of the design of logic circuits using the top-down approach, it is necessary not only to detect but also to locate the source of any inconsistency that may exit between the specification and the implementation. This technique can determine the region containing the error. It has very high resolution and reduces the debugging time by the designers. Extensive experimental results were obtained to demonstrate the effectiveness of the new approach.

1 INTRODUCTION

In the design of integrated circuits, at all levels of abstraction, verification tools compare the design at different levels to make sure that in the synthesis process the designers or optimization tools have not introduced errors, particularly logic errors. Due to the high complexity of VLSI design and the complexity of synthesis tools, this has become increasingly important [1]. Note that even if one has access to correct-by-construction design methods, the issue of proving that the software implementation is also correct remains open [2]. Consequently, it is necessary to detect, locate, and correct any inconsistency that may exist between the specification and the implementation.

There are three approaches at this stage of the design process. The first approach is simulation [3]. In this approach the functional-level circuit and the gate-level circuit are both simulated with same input patterns, and outputs are then compared to check for any inconsistency. Although elegant ideas on the detection of various design errors were proposed in [3], the issue of how to locate the errors was not addressed.

The second approach is Boolean comparison. The functional-level specification is converted into a Boolean expression and then compared with the Boolean equation corresponding to the manually designed gate-level circuit. In [4], the equivalence is tested by proving the graph isomorphism of binary decision diagrams and in [5] by proving the tautology of the exclusive-OR of the two functions. Again both do not offer information on the location of an error.

Recently, Tumura [6] proposed the third approach which partitions the specification, and extracts corresponding sub-functions from the gate-level circuit. The combinational circuit is modeled as a black box that has the same I/O and control signals as the functional-level specification. By using hybrid symbolic simulation, error can be located in a small region. The disadvantages are extra control circuits, and non-trivial partition and extraction for users. There are also formal verification methods [7].

Our approach treats the functional-level specification as a black box, and the gate-level implementation as an "existing product". The test patterns for single stuck-line faults are generated from the gate-level implementation by the PODEM algorithm [8]. These test patterns are applied to both the specification and the implementation to check if there is any inconsistency on a primary output. If an inconsistency is found, there is a design error in the gate-level implementation and we will further locate the site of error. This method is deterministic and has high resolution.

2 ASSUMPTIONS

1. The functional specification need not be completely simulatable, i.e., some test patterns may not exist.

0-8186-2780-8/92 $3.00 © 1992 IEEE

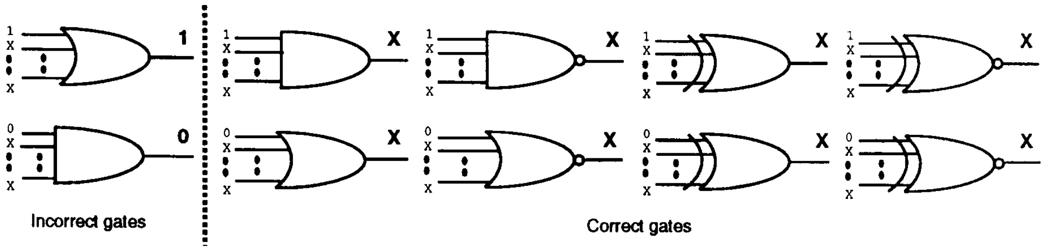


Figure 1: Don't-care propagation for n-input OR gate and AND gate.

2. Only one design error exists in a circuit.

3. Five design errors: a gate replacement, an extra/missing inverter on a gate output, an extra gate, a missing gate, and an extra wire [3], are considered.

4. The only gates used are AND, NAND, OR, NOR, XOR, XNOR, NOT, and BUFFER.

5. Three symbols 0, 1, and X(don't-care) are used.

6. The circuit is combinational.

7. Timing errors are not explicitly considered.

3 DON'T-CARE PROPAGATION ALGORITHM

3.1 Concept of Don't-Care Propagation

The don't-care propagation algorithm distinguishes two gates by applying a test pattern containing don't-care(s) to both gates such that one gate has an X output and the other has either a 0 or 1 output. This means that these two gates are different since one can propagate an X to its output but the other cannot. For instance, if the incorrect gate is an n-input OR (or AND) gate as shown in Figure 1, after applying the input pattern 1, X, ..., X (or 0, X, ..., X), the output is a 1 (or 0). However, by applying the same input pattern to the corresponding ideal (functionally correct) gate, we can detect the discrepancy by observing an X on the output.

However, an incorrect XOR or XNOR gate cannot be detected by this method and will be handled by another technique in the next section.

3.2 Details of the Don't-care Propagation Algorithm

In the following, we will describe each step of the don't care propagation algorithm in detail.

[A] Levelize the circuit under test from every primary output to the primary inputs.

[B] Process the circuit under test from the primary outputs to the primary inputs by applying PODEM [8], and the following heuristic: "the best way to generate test patterns is to start from the primary output to the primary inputs such that a gate of the circuit under test has at least one X on its inputs and a stuck-line fault on its output can be propagated to an primary output."

[C] The wire assignments :

(a) During the forward implication in PODEM, we have to assign appropriate values for the outputs of other gates in order to propagate the desired value on an input line of the target gate (a target gate is a gate which has one or more input lines to be tested for either a SA0 or SA1 fault) to an observable output. During the backtracing process, determining the target gate follows the method in [8]. For other gates, we assign a value to an input which is easiest to set and leave other inputs to have X's if the gate is:

(a1) AND gate: one input is a 0 and the others are X's, if the output is a 0;

(a2) OR gate: one input is a 1 and the others are X's, if the output is a 1;

(a3) NAND gate: one input is a 0 and the others are X's, if the output is a 1;

(a4) NOR gate: one input is a 1 and the others are X's, if the output is a 0;

However, we cannot assign an X to an input of an XOR or XNOR gate if the output is not an X. For all other cases, we need to assign either 0's or 1's to all the inputs of the gate and therefore, the input hardest to set was chosen first as in PODEM.

(b) If the output of a gate with at least one primary input has an X, assign values to non-fanout inputs if the gate is:

(b1) AND or NAND gate: assign the easiest to control input a 1 and all others X's.

(b2) OR or NOR gate: assign the easiest to control input a 0 and all others X's.

(b3) XOR gate and XNOR gate: assign a 0 or

1 to the input line easiest to set and X's to all other inputs.

If backtracking is necessary, we will change X to 0 or 1 without considering whether X can be propagated through the gate or not. This is because a don't-care by definition can be either a 0 or 1, and the only disadvantage by doing this is the decreasing probability to propagate the don't-care.

[D] If the gates (at the next lower level) connected to a suspect gate are of the same type as the suspect gate, then these gates are also suspect gates.

[E] After a test pattern has been generated, we apply this test pattern to the circuit under test, and record the corresponding value on each line. With the value on each line and the gate type of each gate, we can collect all the other stuck-line faults which are detected by this test pattern by applying deductive fault simulation method. The set of stuck-line faults collected by applying this test pattern is stored in either the suspect_line_set or the good_line_set so that they can be used in union-intrsection procedure.

[F] The following is the detailed procedure to locate the suspect gates.

- **Step 1:** Record the test pattern which propagates an X to an output of the ideal circuit.
- **Step 2:** If there is only one X in the test pattern, use Property 1 to locate the incorrect gate.
- **Step 3:** If there are more than one X in the test pattern, let only one input, the **target input line**, have an X and other **active primary inputs** have a 0 or 1 without blocking "X" from propagating to the primary output. An **active primary input** is a primary input which must be assigned a value of 0, 1, or X for the the test pattern to be valid. However, if an X is blocked to the primary output, keep the "X" on the primary input of the circuit under test. This is because the current primary input has fanout branches which are necessary for don't-care propagation.

This process continues until a test pattern propagates an X to the primary output of the ideal circuit.

[G] Assign a value on the primary output of the circuit under test based on the following heuristic so that we can propagate X to the primary output of the ideal circuit as soon as possible.

(1) 1 for OR gate and NAND gate,
(2) 0 for AND gate and NOR gate.

4 UNION-INTERSECTION AND X-OR/XNOR GATES

4.1 Diagnosis of Incorrect XOR/XNOR Gates

We will analyze this problem based on the number of inputs. Assume that the XOR/XNOR gate has odd number of inputs.

- [A] If we have an XOR or a XNOR gate in the incorrect circuit, apply the following four types of input patterns in order to distinguish it from other gates:
  - [A1] 00 $\cdots$ 0 (All zeros);
  - [A2] 11 $\cdots$ 1 (All ones);
  - [A3] Any input pattern other than the above test patterns, that causes the output of the XOR or XNOR gate to be 1;
  - [A4] Any input pattern other than the above test patterns, that causes the output of the XOR or XNOR gate to be 0;
- [B] If an XOR or XNOR gate in the ideal circuit is replaced by an AND, OR, or NOR gate, 3-input patterns can offer information for the union-intersection procedure.
- [C] If all detectable input and output lines of an AND, NAND, OR, or NOR gate have both SA1 and SA0 in the suspect_line_set, the possible design error for each case is as following :
  - [C1] AND: NAND or XNOR;
  - [C2] OR: NOR or XNOR;
  - [C3] NAND: AND or XOR;
  - [C4] NOR: OR or XOR.

From [C], we can see that there is a pair of gates on the right side of the colon in each case which cannot be distinguished by the test patterns generated from the gate on the left side. For instance, all the test patterns of a 3-input NAND gate such as 111, 011, 101, and 110 cannot distinguish the NAND gate from a 3-input XNOR gate. Similarly, we can locate an incorrect XOR/XNOR gate with even number of inputs.

4.2 Union-Intersection

The union-intersection procedure applies the negative tests in [9]. A negative test is also called an "if-then" test. The general form of the test is A → B. The conclusion B says that the primary output has a specific value. The premise A is a conjunction that looks like $\mathrm{OK}(P_1) \wedge \ldots \wedge \mathrm{OK}(P_n) \wedge [I_1 \ldots I_m]$. Therefore, the implication A → B means that a set of n paths $\{P_1 \ldots P_n\}$ of an incorrect circuit makes the primary output lines on both the incorrect circuit and its ideal circuits have the same logic value after applying the same test pattern to the primary inputs $\{I_1 \ldots I_m\}$ on both circuits.

5 THE OVERALL SYSTEM

The whole system is divided into two major modules: the error detection module and the error location module.

5.1 Error Detection

In step (1), if the primary output set po_set is not empty, we take one primary output from the po_set, and levelize all the predecessor gates of this primary output to the primary inputs. After levelizing the gates, we include all the output lines of these gates and all the primary inputs to these gates in test_line_set in step (2). Both the x_flag and the error_flag are initialized to 0. The error_flag=1 indicates that there is an inconsistency at the outputs of both circuits and therefore, a design error exists. The x_flag=1 indicates that one output has an X and the other has a 0 or 1. In steps (3) ~ (13), we will test SA0 and SA1 faults for every line in the test_line_set by applying conditions [C], [D], [G] in section 3.2, and the PODEM algorithm.

After finishing testing the subcircuit leading to the current primary output, go to step (14) to check if any fault remains in the suspect_line_set. If the error_flag is set to 1, we go to step (16) since inconsistency between the functional level circuit and the logic gate circuit has been detected during steps (3) ~ (13). When all detectable lines in the test_line_set have been tested, delete the primary output from the po_set in step (15). If po_set is not empty, take the next primary output in step (1); otherwise, execute step (16) to see if the error_flag is set or not. If it is not set, we can leave the system since no functional design error is detected as shown in step (17). Otherwise, we have to go to the error location module.

5.2 Error Location Module

In this module, the precise location of the incorrect gate will be determined. First we check the x_flag. If it is set to 1, steps (2) ~ (9) will be executed as following. First, in step(2), we have to store the test pattern which propagated an X to the P. O. of the ideal circuit in test_pattern_set which will be used later to locate the design error. In step (3), we mark all active primary input lines of the incorrect circuit because we do not need to consider other unused primary input lines. Next, we will consider how a missing wire could be detected. The missing wire error is that a wire should connect to the logic circuit, while the designers inadvertently miss to do so. It means that the missing wire is "invisible" in the incorrect circuit. Therefore, we can only detect all the possible suspect gates but not the exact position of the incorrect gate. A suspect gate is found by checking if any line of the gate has one of its stuck-line faults detected by the test pattern which generates an X on the primary output of the ideal circuit. Therefore, in step (4) all gates which could have a missing wire are included in the missing_wire_set.

In step (5), we apply conditions (b1), (b2), and (b3) of section 3.2[C] to reduce the number of X's on the primary input lines. This process can speed up our execution time since we only need to consider fewer primary input lines with the value X. In steps (7) ~ (8), we assign only one primary input line of the test pattern to X according to the conditions in section 3.2[F]. In step (9), we check the stuck-line faults of each line detected by the test pattern. Whenever we find an X on the primary output of the ideal circuit, we store all the lines to suspect_gate_set. We keep processing steps (7) ~ (9) until the design error is found. On the other hand, if nothing is wrong with that gate, we must have a missing wire to any of the gates listed in the missing_wire_set.

Otherwise, if x_flag = 0, we have to locate the design error as shown in steps (10) ~ (12). In step (10), the error location process continues if not all the lines in the suspect_line_set have been processed. Select a line to process and check if its x_flag is set or not. In step (14), we include all the elements of the exclusive_set in the suspect _gate_set since they could be gate replacement errors. After checking all the suspect lines in the suspect_line_set, the verification of the logic circuit is done.

5.3 Example Design Error

There are five typical types of design errors, only one of them will be discussed. Gate replacement error means that a gate in the ideal circuit is replaced by a different one, on the same set of inputs, when the incorrect circuit is constructed.

Example 1: To detect and locate a functional design error such that a 2-input XNOR gate in the ideal circuit is replaced by a 2-input NOR gate in the incorrect circuit as shown in Figure 2.

Solution: According to condition (2) in 3.2[G], we assign line g of the incorrect circuit to be 0. Therefore, either line e or line f should be 0 and the other one should be X. We choose line $e = 0$ and line $f = X$ since line e is easier to set to 0 than line f. To have line $e = 0$, we assign line $a = 1$ and line $b = X$ in the incorrect circuit. Also, we assign line $c = X$ and line $d = 0$ for line $f = X$ according to the condition (b2) in 3.2[C].

After test pattern #1 in Table 1 is obtained, we apply it to both the incorrect and ideal circuits. As shown in Table 1, we got an X on the primary output of the ideal circuit. Although this means that the gate replacement error has been detected, we need to apply the condition in 3.2[F] to locate the incorrect gate. From test pattern #3, we identify the incorrect gate as gate B.

6 Experimental Results

The algorithms have been implemented in C on Sun workstations under UNIX operating system. The circuits used in the experiment are taken from the set proposed at ISCAS'85 [10] as benchmarks for ATPG(Automatic Test Pattern Generation). All design errors are inserted manually.

Tables 2 give the experimental results on one benchmark circuit C5315. We can see that the number of suspect gates is usually very small and in many cases is one which means that we can locate the error exactly. The exception is that an extra/missing inverter on a certain primary output causes a huge amount of suspect gates. For example, the last row of Table 2 has 261 suspect gates. However, this is not a problem for an extra/missing inverter error. This is because we always check the primary output first.

7 CONCLUSIONS

A new technique, don't-care propagation, is used in conjunction with test pattern generation to generate a test set which is utilized in a new way to compare a gate-level implementation of a circuit with a functionaul-level specification in order to detect and locate logic design errors. This approach is deterministic and has high resolution. A large class of common design mistakes can be handled. The experimental results indeed show that our approaches are very effective.

References

[1] A. Sangiovanni-Vincentelli, "An Overview of Synthesis Systems," *Proc. of Custom Integrated Circuits Conference*, May 1985.

[2] R. K. Cavin and J. Hilbert, "Design of Integrated Circuits: Directions and Challenges," *Proc. of the IEEE*, Vol. 78, pp. 418-435, Feb. 1990.

[3] M. S. Abadir, J. Ferguson T. E. Kirkland, "Logic Design Verification via Test Generation," *IEEE Trans. computer-aided design*, Vol.7, no. 1, pp 138-148, Jan. 1988.

[4] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computer*, Vol. c-35, no. 8, pp 677-691, Aug. 1986.

[5] R. S. Wei and A. Sangiovanni-Vincentelli, "PROTEUS: A Logic Verification System for Combinational Logic Circuits," *Proc. Int. Test Conference*, Sept. 1986.

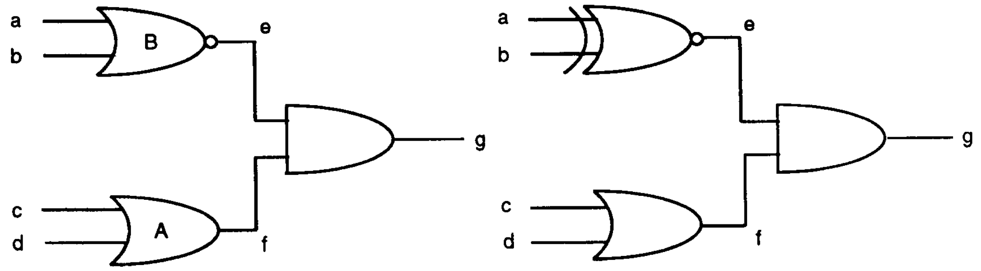
[6] Kensaburo Alfredo Tamura "Locating Functional Errors in Logic Circuits," *Proc. 26th ACM/IEEE Design Automation Conference*, pp 185-191, 1989.

[7] M. S. Chandrasekhar, J. P. Privitera, and K. W. Conradt, "Application of Term Rewriting Techniques to Hardware Design Verification," *Proc. 24th ACM/IEEE Design Automation Conference*, pp. 277-282, June 1987.

[8] P. Geol, "An implicit enumeration algorithm to generate test for combinational circuits," *IEEE Trans. on Computers*, Vol. C-30, pp. 215-222, March 1981 .

[9] C. Paulson, "Classes of Diagnostic Tests," *Proc. 20th ACM/IEEE Design Automation Conference*, pp. 316-322, June 1983.

[10] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target translator in Fortran," *Proc. Int. Symposium on Circuits and Systems*, June 1985.



Incorrect Circuit

Ideal Circuit

Figure 2: A gate replacement design error in Example 1.

Table 1: Test patterns for the circuit in Example 1.

Test Pattern No.		1	2	3	4	5	6	7	8	9
P. I.	a	1	1	1	0	0	1	0	0	0
	b	x	0	x	0	1	0	0	0	0
	c	x	x	1	1	1	1	0	1	0
	d	0	0	0	x	x	x	0	0	1
P. O.	Incorrect	0	0	0	1	0	0	0	1	1
	Ideal	x	0	x	1	0	0	0	1	1
Faults Detected		g SA1	g SA1	g SA1 e SA1	a SA1 b SA1 e SA0 f SA0 g SA0	b SA0 e SA1 g SA1	a SA0 e SA1 g SA1	c SA1 d SA1 f SA1 g SA1	c SA0 f SA0 a SA1 b SA1 e SA0	d SA0 g SA0 a SA1 b SA1 e SA0

Table 2: Experimental results of the benchmark circuit C5315

Error Site (Line No.)	Error Type	Algorithm/Type	# of Suspect Gates	Exe. Time (Seconds)
2204	AND to NOR	X_inv	14	1.02
4655	NAND to XOR	UI_error	1	268.38
4749	Extra Wire	X_prop	1	3.22
5162	Extra Gate	X_prop	1	3.19
5166	AND to OR	X_prop	1	1.45
5172	Missing Gate	X_prop	1	1.59
792	Inverter	X_miss	2	1.06
5315	Inverter	UI_error	261	280.3