VLSI DESIGN
1996, Vol. 4, No. 3, pp. 167–179
Reprints available directly from the publisher
Photocopying permitted by license only

A Novel Path Delay Fault Simulator Using Binary Logic

ANANTA K. MAJHI[a,*], JAMES JACOB[a,†] and LALIT M. PATNAIK[b,‡]

[a]*Department of ECE, Indian Institute of Science, Bangalore, INDIA;* [b]*Microprocessor Applications Laboratory, Indian Institute of Science, Bangalore, INDIA*

A novel path delay fault simulator for combinational logic circuits which is capable of detecting both robust and nonrobust paths is presented. Particular emphasis has been given for the use of binary logic rather than the multiple-valued logic as used in the existing simulators which contributes to the reduction of the overall complexity of the algorithm. A rule based approach has been developed which identifies all robust and nonrobust paths tested by a two-pattern test $<V_1,V_2>$, while backtracing from the POs to PIs in a depth-first manner. Rules are also given to find probable glitches and to determine how they propagate through the circuit, which enables the identification of nonrobust paths. Experimental results on several ISCAS'85 benchmark circuits demonstrate the efficiency of the algorithm.

Keywords: Delay Faults; Test Generation; Fault Simulation; CAD Tools; Benchmark

1. INTRODUCTION

There has always been an increasing demand for faster digital systems. The maximum allowable clock frequency in a synchronous system is determined by the propagation delay of a signal in the combinational network between the latches. Due to some physical defects, statistical process variations or stray capacitances, if the delay of the manufactured network exceeds specifications, there is a chance of unstabilized and possibly incorrect logic values being latched at the outputs. *Delay fault testing* can be used to ascertain that manufactured digital circuits meet their timing specifications and operate correctly at desired clock rates. Thus, delay fault testing has achieved great theoretical and practical importance for the design of high-speed logic circuits. In this paper, we have presented a novel path delay fault simulator for combinational circuits which detects the faulty paths under the application of two-pattern test pairs.

A delay fault only causes logic values to respond slower than the normal which leads to the malfunctioning of the logic network. Unlike a stuck-at fault, a delay fault does not affect the steady state logical operation of a system, but it affects the timing behavior of the system and degrades the overall system performance. From the operational point of view, a combinational logic system is said to be free of delay faults if the transition (rising or falling) initiated at the primary inputs (PIs) arrives at the primary outputs

*Phone: (91) 80-3342451. Fax: (91) 80-3341683. E-mail: majhi@ece.iisc.ernet.in
†Phone: (91) 80-3344411-2285. Fax: (91) 80-3341683. E-mail: james@ece.iisc.ernet.in
‡Corresponding author. Phone: (91) 80-3342451. Fax: (91) 80-3341683. E-mail: lalit@vigyan.iisc.ernet.in

(POs) in less time than the system clock timing specifications.

In the recent past, two different fault models have been proposed in literature, viz, the *gate delay fault* model and the *path delay fault* model. The gate delay fault model as described in [4,7,12,16] was also referred as *transition fault* model. In this model, the lumped gate delay fault is localized to a particular gate input or output. This is also analogous to dc stuck-at fault model, i.e., the slow-to-rise and slow-to-fall transitions correspond to the dc stuck-at zero and stuck-at one respectively, since it behaves a stuck-at zero or stuck-at one temporarily [16]. However, it does not model the cumulative effect of the gate delays along a path from the PIs to POs. On the other hand, the path delay fault model [3,6,14,15] alleviates this deficiency. In this model, the delay fault is associated with a physical path in the circuit and the path is declared to be free of delay faults if the transition provoked at the input propagates to the outputs through the specified path in less time than the operational system clock interval. Thus, the path delay fault model provides the advantageous capability of modeling the distributed failures which are mainly caused by statistical process variations and physical defects during the manufacturing process.

There is a major bottleneck in selecting the paths for which the test generation and fault simulation are to be carried out, since as the circuit size grows the number of paths grow exponentially with circuit depth and the number of fanouts. Hence, prior to test generation and fault simulation process it may be necessary to focus on a subset of all possible paths in the logic network. There are several methods available in literature like *worst-case path selection* and *threshold-based path selection* [9], and a *polynomial time* algorithm to find a minimum cardinality path set has been described in [5].

During the last few years, a considerable number of test generation methods for path delay faults have been developed [3,6,8,9,13], whereas the problem of fault simulation has only been addressed in a few [2,11,14,15]. Smith [15] has proposed a six-valued logic, which identifies the paths tested for delay faults independent of the delays of any individual gate in the network. Schulz *et al* [14] have presented a four-valued logic for an accelerated fault simulation approach for the delay faults, which applies parallel processing of patterns at all stages of the calculation procedure. Bose *et al* [2] have used Smith's six-valued algebra for delay fault simulation of synchronous sequential circuits. Pomeranz *et al* [11] have used a non-enumerative method to estimate path delay fault coverage. Similarly, multiple-valued logic has been used for the test generation process of the delay faults; e.g., eleven-valued logic in [7], ten-valued logic in [3,8], and five-valued logic in [6,9]. The number of logic states used is a factor that determines the time and memory complexity of the algorithms based on them; fewer logic symbols lead to less complex implementations [6]. Hence, we have employed the simple *two-valued* logic for path delay fault simulation.

During event-driven logic simulation with respect to the first vector (*initialization vector*) of the two-pattern test $< V_1, V_2 >$, we evaluate the gate sensitivity, and classify the gate inputs as *controlling* (CO) and *noncontrolling* (NC) based on the logic values on them. After evaluating the true logic values with respect to the second vector (*propagation vector*) using the same event-driven approach, the possibility of a *glitch* event at the output of a gate is determined taking both the initialization and propagation vectors into account. An event-queue is also maintained for the propagation of the glitch events from PIs to POs in order to determine the robustness/nonrobustness of a path. Finally, we backtrace from the POs to PIs in a depth-first manner based on some specified rules to trace the faulty paths. Thus, our algorithm detects both robust and nonrobust paths during the simulation procedure. Additionally, the program can also count the total number of all possible paths in the network. We have not used any of the path selection algorithms mentioned earlier to create a target path list from the enormous number of possible paths. Once a path is detected as faulty by a vector pair, it is added to the global path list to keep track of the fault coverage.

2. THEORETICAL BACKGROUND AND BASIC DEFINITIONS

Hardware Model and Clock Timings

Unlike a single pattern test as used in case of dc stuck-at fault testing, delay fault testing requires a two-pattern test $<V_1, V_2>$. Considering the well accepted hardware model as described in literature [3,6,8,13,15] illustrated in Fig. 1, we assume that the *initialization vector* V_1 is loaded into the input latches at time t_0. After the signals of the network get stabilized under V_1, the *propagation vector* V_2 is applied at time t_1 by activating the clock C_1. Finally, the logic values are sampled from the output latches at time $t_2 = t_1 + t_c$ by pulsing clock C_2, where t_c represents the desired system clock interval. The main objective of delay fault testing is to ensure that the maximum propagation delay of a path in the circuit is less than the system clock interval t_c.

Robust and Nonrobust Paths

Robust path: A structural path P in the logic network can be termed as a robust path with respect to a two-pattern test $<V_1, V_2>$, iff,

- a transition provoked at the input to the path propagates to output through the structural path P.
- it is guaranteed that all signals on the structural path P cannot attain their final values with respect to the propagation vector V_2 of the two-pattern test $<V_1, V_2>$, unless the provoked transition at the input has arrived at them.

A delay fault in the robust path will cause a faulty logic value at the output of the path independent of other path delays in the network. The corresponding vector pair which detects the faulty robust path is defined as a *robust test.*

Nonrobust Path: A structural path P in the logic network can be termed as a nonrobust path with respect to a two-pattern test $<V_1, V_2>$, iff,

- a transition is provoked at the input of path P by the test $<V_1, V_2>$.
- the excessive delay on path P can be detected by the two-pattern test $<V_1, V_2>$ under the assumption that there does not exist any other faulty paths in the network. In other words, the propagation vector V_2 causes all *off-path sensitizing inputs* [9,14] along the structural path P to assume their *noncontrolling* values to propagate the provoked transition at the input.

The corresponding vector pair which detects the faulty nonrobust path is termed as *nonrobust test.*

Fig. 2 illustrates the two-pattern test $<V_1, V_2>$ consisting of the initialization vector V_1 = (11110) and the propagation vector V_2 = (11010). The structural path P_1 = (C-E-I-J-L-N-Q) is referred as a robust path with respect to the test $<V_1, V_2>$, since all lines in the path P_1 cannot attain their final values unless the transition provoked at input C arrives at them. The structural path P_2 = (C-E-I-J-L-M-P) is referred as a nonrobust path with respect to the test $<V_1,V_2>$, because an excessive delay in the rising transition on line H may cause the PO (line P) to have its expected true final logic value 1 at the sampling time t_2 as shown in Fig. 2(b), regardless the delay on path P_2. Thus, it leads to the conclusion that the circuit is fault-free, although there are delays on both lines H and L, M. But if there is no delay on line H as shown

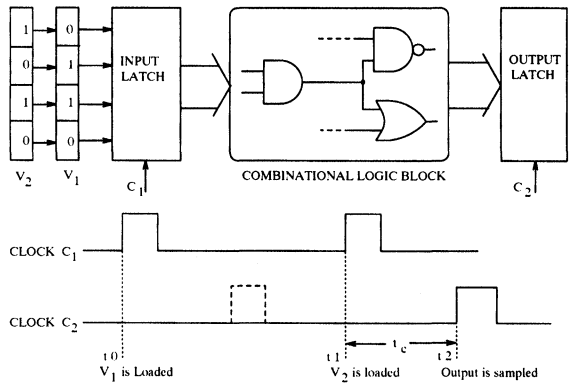

FIGURE 1 Hardware Model & Clock Timings

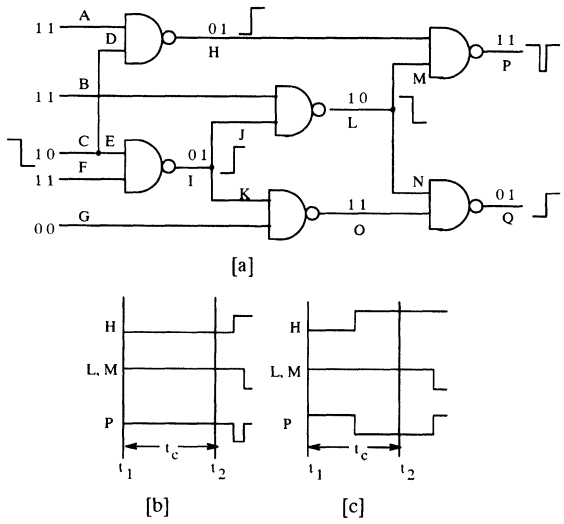

FIGURE 2 Example of Robust & Nonrobust Paths

in Fig. 2(c), we will get the faulty logic value 0 at the PO (line P) at the sampling time t_2. Thus, path P_2 can be detected as a nonrobust path with respect to the test pair $<V_1, V_2>$, iff there is no delay on line H.

Sensitivity and Gate Evaluation

During event-driven logic simulation with respect to the initialization vector of the two-pattern test $<V_1, V_2>$, we classify each gate into the following classes:

- **Globally Sensitive (GS):** A logic gate with all inputs at *noncontrolling* (NC) value is classified as a GS gate. Noncontrolling values are 1(0) for AND/NAND(OR/NOR) gates.
- **Potentially Sensitive (PS):** A logic gate with at least one input at *controlling* (CO) value is classified as a PS gate. Controlling values are 0(1) for AND/NAND(OR/NOR) gates.
- **Odd-Parity Sensitive (OPS):** A logic gate whose output is complemented when an odd number of inputs have events is classified as an OPS gate, e.g. XOR/XNOR gates. (We have restricted the OPS gates to two input gates only throughout our discussion.)
- **Input Sensitive (IS):** A logic gate with single input is classified as IS gate, since output is complemented by complementing the input. Inverters/Buffers are IS gates.

3. PATH DELAY FAULT SIMULATION USING BINARY LOGIC

In order to perform the path delay fault simulation with respect to the given two-pattern test vector, we follow the procedures given below:

- While doing event-driven logic simulation with respect to the initialization vector V_1 of the two-pattern test $<V_1, V_2>$, the gate sensitivity (i.e. GS, PS, OPS, IS) as well as the *controlling* (CO) and *noncontrolling* (NC) inputs of the gates are determined based on the rules specified in the previous section. An example of the same has been given in Fig. 3, where we have taken the test pair $<V_1, V_2>$ as (11110,11010). The gate inputs having a star (*) represent the CO inputs with respect to the initialization vector V_1.
- Secondly, event-driven logic simulation is performed with respect to the propagation vector V_2 and the true final logic values are evaluated. Along with the logic simulation we additionally check for the possibility of a glitch event at the gate output. An example showing the generation of a glitch at the gate output is shown in Fig. 4. There are both rising and falling transitions at the inputs of the AND gate. According to the rules of event-driven logic simulation, we do not have a logic event on the output of the AND gate. However, this condi-

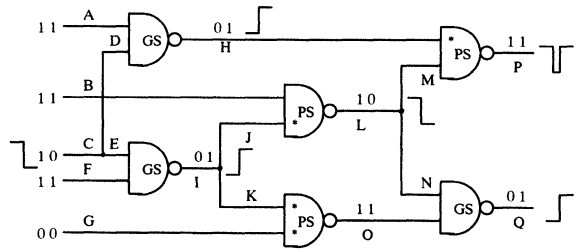

FIGURE 3 Evaluation of Gate Sensitivity, CO & NC Inputs

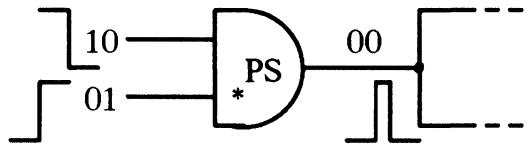

FIGURE 4 Glitch Generation

tion can cause a glitch event at the gate output, if the falling transition at the NC input is delayed with respect to the rising transition on the CO input. Hence, we put the fanouts of this gate into the event-queue in order to propagate the glitch event towards the POs. The existence of a glitch helps us to determine the robustness/nonrobustness of the path. We have developed a set of rules to accurately model the generation and propagation of a glitch and these are explained in the next section.

- After evaluating the true logic values with respect to V_2, we perform glitch propagation (for those glitch events which were scheduled in the previous step) in an event-driven manner towards the primary outputs.
- Finally, we backtrace from the POs to PIs in a depth-first manner to determine the faulty paths (both robust as well as nonrobust) based on some specified rules which are explained in a following section.

4. GLITCH GENERATION AND PROPAGATION

In order to determine the robustness/nonrobustness of a path, we need to check for the possibility of a glitch event at the gate output. It is also required to determine whether the glitch can propagate to the primary outputs or not. Only gates whose output does not have a logic event need be considered for glitch generation and propagation. We have developed a set of simple rules that govern the generation and propagation of the glitch event as stated below:

Glitch Generation

- **Rule 1:** If the gate is PS, all CO inputs have events and exactly one NC input has event, then output will have a glitch as shown in Fig. 5(a).
- **Rule 2:** If the gate is OPS and both inputs have events, then output will have a glitch as shown in Fig. 5(b),(c).

Glitch Propagation

- **Rule 3:** If the gate is GS or OPS, a glitch on any input will propagate to the output as shown in Fig. 6(a),(b).
- **Rule 4:** If the gate is PS, exactly one CO input has a glitch, all other CO inputs (if more than one CO input present) have events and no NC input has event, then the glitch on the CO input will propagate to the gate output as shown in Fig. 6(c). We do not restrict the presence of glitches on the NC inputs.
- **Rule 5:** If the gate is IS, the glitch on the single input will propagate to the output as shown in Fig. 6(d),(e).

Illustration: Fig. 7 shows the generation of glitch events at lines I and J according to Rules 1 and 2 respectively. The glitch events are then propagated towards primary output based on the Rules 3, 4 and 5 mentioned above. Finally, the path (E-G-M-P) can be detected as a nonrobust path (based on the Rules described in the next section) under the test (011100, 101010).

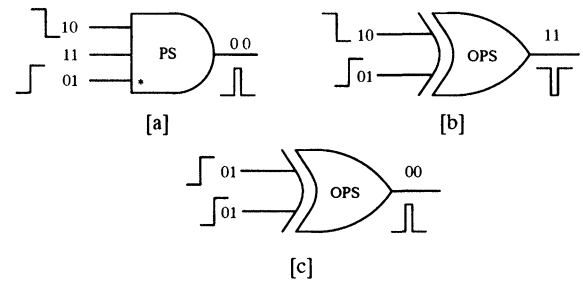

FIGURE 5 Rules for Glitch Generation

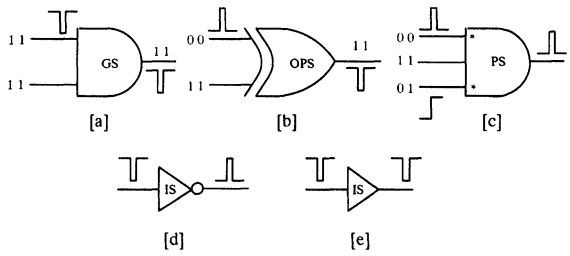

FIGURE 6 Glitch Propagation

5. BACKTRACING FOR ROBUST AND NONROBUST PATHS

Our strategy in identifying robust and nonrobust paths detected by a vector pair is to backtrace from POs to PIs in a depth first manner and mark each input of gates along the path as robust or nonrobust based on a set of simple rules. Once we reach a PI, we have identified a path and it is classified as a robust or nonrobust path depending on the status of the lines along the path. Backtrace employs a recursive procedure and to start with, a PO having an event is marked as robust and a PO with glitch is marked as nonrobust. A gate is declared robust (nonrobust) if its output is robust (nonrobust). Now the following rules are applied to compute the robust/nonrobust status of the inputs of a gate.

Rules for Evaluating the Inputs of a Robust Gate

- **Rule 1:** If the gate is PS, all CO inputs are marked as robust and all NC inputs with glitches are marked as nonrobust.

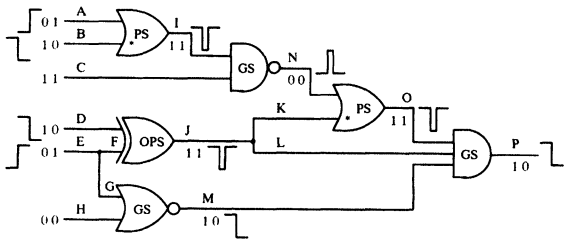

FIGURE 7 Example of Glitch Generation & Propagation

- **Rule 2:** If the gate is GS, exactly one input, say input j, has event and no other input has a glitch, then input j is marked as robust. If at least one input (except j) has a glitch then input j will be marked as nonrobust.
- **Rule 3:** If the gate is OPS, the input with event is marked as robust if there is no glitch on the other input; else both inputs are marked as nonrobust.

Example 1: Fig. 8 illustrates the logic underlying Rules 1–3 for determining the robust and nonrobust inputs of a robust gate. In Fig. 8(a), the output D of AND gate (of type PS) has an event and has been marked as robust. The controlling inputs A and B with rising transition ($0 \rightarrow 1$) will be marked as robust, since the path delay fault on these inputs can robustly propagate to output D. The delay fault on input C (i.e., glitch $1 \rightarrow 0 \rightarrow 1$) can be propagated to output D, if there is no delay on inputs A and B. Thus, input C is marked as nonrobust and Rule 1 is justified.

Example 2: In Fig. 8(b), the output D of the OR gate (of type GS) has an event and has been marked as robust. The input B having a rising transition ($0 \rightarrow 1$) will be marked as robust, since the delay fault on this input can be robustly propagated to output D. But the same input B will be marked as nonrobust as shown in Fig. 8(c), since there is a glitch ($0 \rightarrow 1 \rightarrow 0$) on input A. The glitch on A may cause the true final logic value 1 on output D at the sampling time and the delay fault on B may go undetected. The delay fault on B can propagate to output provided

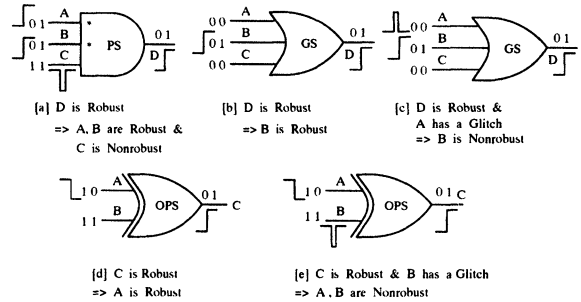

FIGURE 8 Robust & Nonrobust Inputs of a Robust Gate

there is no delay on input A and thus B is marked as nonrobust satisfying Rule 2. Again, if there are more than one input of the GS gate having events, then none of the inputs will be marked as robust, because the delay fault on one input will be masked by the transition on the other input causing the gate output to have its true final logic value at the sampling time.

Example 3: In Fig. 8(d), the delay fault on input A of the XOR gate (of type OPS) can robustly propagate to output C and thus it is marked as robust. In Fig. 8(e) , the delay fault on A can be masked by the delay fault on B (i.e. glitch $1 \rightarrow 0 \rightarrow 1$) and vice versa. As a result the output can have its true final logic value at the sampling time though there are delay faults on both inputs. The delay fault on one input will propagate to the output provided there is no delay on the other input and thus both inputs A and B are marked as nonrobust satisfying Rule 3.

Rules for Evaluating the Inputs of a Nonrobust Gate

- **Rule 4:** If the gate is PS and output has a glitch, then the single NC input having event is marked as nonrobust provided all CO inputs have events. Again, the single CO input with a glitch is marked as nonrobust if there is no event on other NC inputs. If the gate is PS and output has event, all CO inputs having events as well as all NC inputs with glitches are marked as nonrobust.
- **Rule 5:** If the gate is GS and output has a glitch, all inputs with glitches are marked as nonrobust. If the gate is GS and output has event and exactly one input, say input j, has event, then input j is marked as nonrobust (independent of the presence of glitches on other inputs).
- **Rule 6:** If the gate is OPS and output has event or glitch, then input with event or glitch is marked as nonrobust.

Example 1: Fig. 9 illustrates the logic behind Rules 4–6 for determining the nonrobust inputs of a nonrobust gate. It is to be noted that the inputs of a robust gate can be marked as robust as well as nonrobust

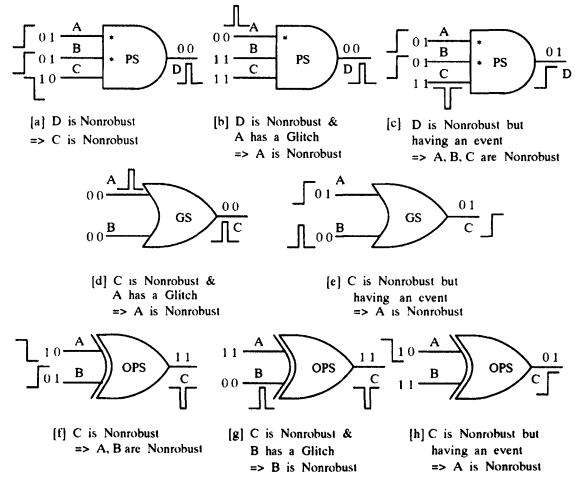

FIGURE 9 Nonrobust Inputs of a Nonrobust Gate

whereas the inputs of a nonrobust gate will be only nonrobust. In Fig. 9(a), the output D of the AND gate (of type PS) has a glitch and has been marked as nonrobust. The delay fault on the single NC input C can propagate to the output if all CO inputs have events and there is no delay fault on any one of the CO inputs. Thus, input C is marked as nonrobust. In Fig. 9(b), the glitch ($0 \rightarrow 1 \rightarrow 0$) on the CO input A can propagate to output D, if there is no event on other NC inputs and thus input A is marked as nonrobust. In Fig. 9(c), the output D has been marked as nonrobust though it has an event. (This possibility has been explained in the previous example used for illustrating Rules 1-3). Referring to Fig. 9(c), the delay fault on inputs A and B can robustly propagate to output D, but these inputs will be marked as nonrobust since output D is nonrobust. Again, input C having a glitch ($1 \rightarrow 0 \rightarrow 1$) will be marked as nonrobust as explained earlier and thus Rule 4 is justified.

Example 2: Consider the case when the output of the GS gate has a glitch and has been marked as nonrobust. The delay fault (i.e., glitch) on any input will propagate to the output. An example is given in Fig. 9(d) in which input A has a glitch and thus marked as nonrobust. In Fig. 9(e), the output C of the GS gate has an event and has been marked as nonro-

bust. The delay fault on input A having event will robustly propagate to output C independent of the presence of glitch on other input B and thus A will be marked as nonrobust since the output C is nonrobust satisfying Rule 5.

Example 3: Consider the case when the output of an OPS gate has a glitch and has been marked as nonrobust. The delay fault on any input having a logic event or a glitch event can propagate to the output. Thus, inputs A and B having logic events will be marked as nonrobust as shown in Fig. 9(f) and input B having a glitch event will be marked as nonrobust as shown in Fig. 9(g). If the output of the OPS gate has an event and has been marked as nonrobust, then the delay fault on the input having a logic event will propagate to the output. In Fig. 9(h), the input A having a logic event will be marked as nonrobust and hence Rule 6 is justified.

Rules for Evaluating the Input of an IS Gate

Rule 7: If the gate is IS, its input is marked as robust (nonrobust) provided the output is robust (nonrobust).

Illustration: Fig. 10 shows an example circuit where we have applied Rules 1 to 7 while backtracing from POs to PIs and determined the status of each line. We have applied the two pattern test (110, 101) at the inputs of the logic circuit and propagated the logic events as well as the glitch events through the circuit. Finally, the output P will have a logic event (0 $\rightarrow$ 1) whereas the output Q will have a glitch event (1 $\rightarrow$ 0 $\rightarrow$ 1). Hence, the output P will be marked as robust and Q as nonrobust. First, we backtrace from the robust output P towards primary inputs in a depth first manner. The inputs G and M of the AND gate (of type PS) will be marked as robust based on Rule 1. Backtracing along the line G, the stem line F will be marked as robust since its fanout branch G is robust. The input C of the XOR gate (of type OPS) will be marked robust based on Rule 3 and thus primary input B which is a stem line for C, will also be marked as robust. After reaching the primary input B, we found a path (B-C-F-G-P) which is robust since all the lines along the path have been marked as robust. Next, backtracing along line M, the stem line L will be marked as robust. The inputs H and J of the XOR gate (of type OPS) will be marked as nonrobust based on Rule 3. Backtracing along the line H, the stem line F (which has already been marked as robust) will be marked as nonrobust. The input C of the XOR gate will be marked as nonrobust based on Rule 6 and thus the primary input B will also be marked as nonrobust. Hence, we found another path (B-C-F-H-L-M-P) which is nonrobust since some of the lines along this path have been marked as nonrobust. Next, backtracing along the line J, the stem line I will be marked as nonrobust. The input D of the AND gate (of type PS) will be marked as nonrobust based on Rule 4 and thus the primary input B will also be marked as nonrobust. Hence, we found another path (B-D-I-J-L-M-P) as nonrobust. After enumerating all the paths whose output converge on P, we then backtrace from the nonrobust output Q. The input K of the NAND gate (of type PS) will be marked as nonrobust based on Rule 4 and thus the stem line I will also be marked as nonrobust. Backtracing again, the input D will be marked as nonrobust according to Rule 4 and the stem line B will also be marked as nonrobust. Thus, we found another path (B-D-I-K-Q) as nonrobust, and now all robust and nonrobust paths detected by the vector pair (110, 101) have been enumerated. As shown in Fig. 10, we have used the following notation to mark the status of a line, e.g., a line is denoted as r which is only robust, nr which is only nonrobust and rnr which is both robust as well as nonrobust. Thus, we conclude that

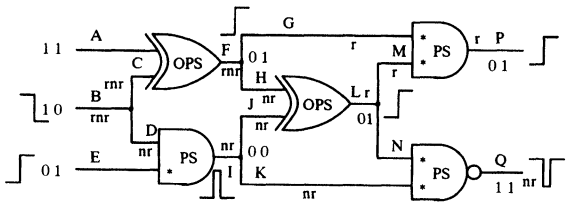

FIGURE 10 Example of Robust & Nonrobust Path Detection

1. a line can be robust as well as nonrobust with respect to different paths.

2. a functional path will be robust iff all lines of the path are marked as *r* or *rnr*.
3. a functional path will be nonrobust iff at least one of the lines on the path is marked as *nr*.

6. SIMULATION RESULTS AND CONCLUSIONS

We have implemented the proposed path delay fault simulation algorithm in the C language (about 1500 lines of code) on an IBM RS-6000/580 computer system running UNIX. Table I shows the number of primary inputs, primary outputs, number of levels in the circuit and total number of physical paths for the ISCAS'85 combinational benchmark circuits. The CPU times for counting the number of paths have been included in Table I. We have not enumerated the path lists. Path counting was carried out as described in [11]. The number of logical path delay faults modeled is equal to twice the number of physical paths present in a circuit since both the rising and falling transitions are considered at the input of each path. The number of logical path faults varies from 17284 for circuit c880 to 197.88×10^{18} for circuit c6288.

In order to derive a set of deterministic delay fault test vectors for use in simulation we have employed S. Patil's test generator[9] on the ISCAS'85 benchmark circuits. In Table II, the total number of paths generated by *worst-case path selection procedure [9]* is given in column *Examined*. The number of two-pattern test vector pairs generated are given in the column *Vectors*.

TABLE I Number of possible physical paths, PIs, POs, and levels

Circuit	# PI	# PO	# Levels	# Physical paths	Time (secs.)
c432	36	7	30	83926	0.090
c499	41	32	17	9440	0.130
c880	60	26	35	8642	0.210
c1355	41	32	40	4173216	0.320
c1908	33	25	61	729057	0.460
c2670	233	140	53	679960	0.670
c3540	50	22	71	28676671	0.870
c5315	178	123	68	1341305	1.340
c6288	32	32	218	98.94×10^{18}	1.480
c7552	207	108	62	726494	1.880

TABLE II Test generation results

Circuit	Examined	Tested	Dropped	Notest	Vectors	Time(secs.)
c432	330	38	191	101	33	31.73
c499	808	162	626	20	146	165.41
c880	729	685	21	23	506	10.28
c1355	848	64	582	202	64	140.86
c1908	1282	399	333	550	337	193.07
c2670	1871	619	902	350	531	405.64
c3540	2559	159	910	1490	123	784.27
c5315	4353	2650	1074	629	2322	858.83
c6288	3875	14	1680	2181	10	2379.47
c7552	5432	674	2907	1851	477	2478.18

Table III shows the delay fault simulation results generated by our fault simulator implementation on the same circuits. We have simulated three sets of vector pairs to detect both the robust and nonrobust

TABLE III Path delay fault simulation results

Circuit	# Vector pairs	# Robust paths	# Nonrobust paths	Time (secs.)
c432	65 (dpd)	179	278	0.20
	45 (dsa)	29	819	0.23
	5000 (r)	556	6893	17.53
c499	291 (dpd)	318	4279	1.24
	59 (dsa)	109	899	0.28
	5000 (r)	71	10729	19.22
c880	1011 (dpd)	2002	3027	8.04
	29 (dsa)	112	891	0.39
	5000 (r)	1069	5178	29.81
c1355	127 (dpd)	64	1476	1.49
	94 (dsa)	69	13625	3.04
	5000 (r)	243	171099	215.04
c1908	673 (dpd)	1322	10270	14.21
	140 (dsa)	220	5870	3.12
	5000 (r)	1395	21976	92.70
c2670	1061 (dpd)	1872	6171	34.78
	66 (dsa)	592	4778	2.57
	5000 (r)	2388	39655	429.05
c3540	245 (dpd)	593	9771	14.00
	109 (dsa)	293	13108	8.54
	3000 (r)	2404	174759	3161.80
c5315	2000 (dpd)	4761	27388	185.33
	55 (dsa)	810	9179	4.06
	4500 (r)	5488	93124	487.76
c6288	19 (dpd)	31	NA	2.14
	15 (dsa)	35	NA	1.97
	5000 (r)	170	NA	136.50
c7552	953 (dpd)	5760	35797	231.10
	86 (dsa)	619	20989	11.63
	1500 (r)	3085	90983	628.36

(dpd)–deterministic test patterns for delay faults. (dsa)–deterministic test patterns for stuck-at faults. (r)–random test patterns.

paths for all circuits. The first row of each circuit shows the simulation results of the deterministic test pattern pairs for path delay faults (denoted as *dpd*) obtained from Table II. In our simulation procedure, we use the second pattern (propagation vector) of the present simulation as the first pattern (initialization vector) for the next simulation. Hence, we get $2n-1$ vector pairs where n is the number of deterministic vectors generated, which are given in Table II. The second row in Table III for each circuit shows the simulation results of the deterministic test patterns obtained for stuck-at faults and these are denoted as *dsa*. The deterministic test patterns for single stuck faults were obtained using COMPACTEST[10] and these had complete coverage of all non-redundant stuck faults. The third row for each circuit shows the simulation results for *random* test patterns.

In our implementation, once a faulty path is detected by a test vector pair, it is added to the global path list of detected paths provided the path does not already exist in the global list of detected paths. The circuit c6288 was not simulated for nonrobust paths since the number of nonrobust paths detected were extremely large and it did not complete within reasonable CPU time. Hence, the CPU time mentioned for this circuit is the time taken only for detecting the robust paths. We have not imposed any restriction on the path lengths. All faulty paths detected by the test patterns are added to the global path list.

Table IV shows the overall statistics. The simulation results of circuit c6288 has not been considered for the statistics in Table IV since we have not considered the nonrobust paths for this circuit. It is to be noted that the number of robust paths detected per vector pair by *dpd* patterns are greater than that of *random* patterns. For example, 291 *dpd* test patterns detect 318 robust paths whereas 5000 *random* patterns detect only 71 robust paths for circuit c499 and 1011 *dpd* test patterns detect 2002 robust paths whereas 5000 *random* patterns detect only 1069 robust paths for circuit c880 as given in Table III. On an average the number of nonrobust paths detected per vector pair by *dsa* patterns is greater than that of *dpd* patterns since *dpd* patterns were obtained using the deterministic test generator generating only robust tests. As shown in Table IV, the per vector coverage of nonrobust paths by *dsa* patterns is 102.72 whereas that of *dpd* patterns is 15.32. For example, 45 *dsa* patterns detect 819 nonrobust paths whereas 65 *dpd* patterns detect only 278 for circuit c432 and 94 *dsa* patterns detect 13625 nonrobust paths whereas 127 *dpd* patterns detect only 1476 nonrobust paths for circuit c1355 as shown in Table III. On the whole the per vector coverage by *dsa* patterns is better than that of *dpd* and *random* patterns for both robust and nonrobust paths as shown in Table IV.

TABLE IV Overall statistics

Type of vectors	# Vector pairs	# Robust paths	# Nonrobust paths	Paths det./vector pair	
				Robust	Nonrobust
dpd	6426	16871	98457	2.63	15.32
dsa	683	2853	70158	4.18	102.72
random	39000	16699	614396	0.43	15.75

We believe that our novel path delay fault simulator which uses the simple two-valued algebra will be faster and require less memory than the existing simulators, since the computational complexity is drastically reduced in our approach. Further, it is not necessary to have a look-up table as required in the multiple-valued logic evaluation.

Acknowledgments

The authors would like to thank to Dr. Vishwani D. Agrawal of AT&T Bell Labs, U.S.A. for valuable discussions. They also thank Dr. Srinivas Patil and Dr. Lakshmi N. Reddy of IBM Corp., U.S.A. for providing the test generator for path delay faults.

References

[1] M. A. Abramovici, P. R. Menon, and D. T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation", *IEEE Design & Test*, 83–92, February 1984.

[2] S. Bose, P. Agrawal, and V. D. Agrawal, "A Path Delay Fault Simulator for Sequential Circuits", *Proc. of 6th Int'l Conf. on VLSI Design*, India, 269–274, January 1993.

[3] K. Fuchs, F. Fink, and M. H. Schulz, "DYNAMITE: An Efficient Automatic Test Pattern Generation System for Path

Delay Faults", *IEEE Trans. on CAD*, Vol. 10, 1323–1335, October 1991.
[4] J. D. Lesser and J. J. Shedletsky, "An Experimental Delay Test Generator for LSI Logic", *IEEE Trans. on Computers*, Vol.C-29, 235–248, March 1980.
[5] W. N. Li, S. M. Reddy, and S. K. Sahni, "On Path Selection in Combinational Logic Circuit", *IEEE Trans. on CAD*, Vol. 8, 56–63, January 1989.
[6] C. J. Lin and S. M. Reddy, "On Delay Fault Testing in Logic Circuits", *IEEE Trans. on CAD*, Vol. CAD-6, 694–703, September 1987.
[7] E. S. Park and M. R. Mercer, "An Efficient Delay Test Generation System for Combinational Logic Circuits", *IEEE Trans. on CAD*, Vol. 11, 926–938, July 1992.
[8] E. S. Park and M. R. Mercer, "Robust and Nonrobust Tests for Path Delay Faults in a Combinational Circuit", *Proc. IEEE Int'l Test Conf.*, 1027–1034, September 1987.
[9] S. Patil and S. M. Reddy, "A Test Generation System for Path Delay Faults", *Proc. IEEE Int'l Conf. on Computer Design*, 40–43, 1989.
[10] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits", *Proc IEEE Int'l Test Conf.*, 194–203, 1991.
[11] I. Pomeranz and S. M. Reddy, "An Efficient Non-Enumerative Method to Estimate Path Delay Fault Coverage", *Proc. IEEE Int'l Conf. on CAD*, 560–567, November 1992.
[12] A. K. Pramanick and S. M. Reddy, "On the Detection of Delay Faults", *Proc. IEEE Int'l Test Conf.*, 845–856, 1985.
[13] S. M. Reddy, C. J. Lin, and S. Patil, "An Automatic Test Pattern Generator for the Detection of Path Delay Faults", *Proc. IEEE Int'l Conf. on CAD*, 284–287, 1987.
[14] M. H. Schulz, F. Fink, and K. Fuchs, "Parallel Pattern Fault Simulation of Path Delay Faults", *Proc. IEEE Design Automation Conf.*, 357–363, June 1989.
[15] G. L. Smith, "Model for Delay Faults Based on Paths", *Proc. IEEE Int'l Test Conf.*, 342–349, 1985.
[16] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar, "Transition Fault Simulation", *IEEE Design & Test*, 32–38, April 1987.

APPENDIX

Pseudo Code for Fault Simulation Algorithm

```
main( )
{
  read_ckt( ); /* reads the circuit information from
                  netlist */
  get_init_vector( ); /* gets the initialization vector
                         V1 */
  logsim( ); /* event-driven logic simulation w.r.t V1,
                also evaluates the gate sensitivity as
                well as the CO and NC inputs */
  get_prop_vector( ); /* gets the propagation vector
                         V2 */
  logsim( ); /* event-driven logic simulation w.r.t V2,
                also finds the probability of a glitch at
                the gate output */
  glitch_propagation( ); /* propagates the glitch to
                            POs */
  rob_nonrob_test( ); /* backtrace from POs to PIs in
                         a depth-first manner to trace the faulty
                         robust and nonrobust paths */
  print_paths( ); /* displays the faulty paths detected
                     by the test pattern */
}
```

Pseudo Code for the Main Program

```
rob_nonrob_test( )
{
  for each primary output (i) {
    if there is no event, but a glitch on (i) {
      mark output (i) as nonrobust;
      backtrace_nonrobust_path(i);
    }
    else if there is event on output (i) {
      mark output (i) as robust;
      backtrace_robust_path(i);
    }
  }
}
```

Pseudo Code for Robust & Nonrobust Test

```
glitch_propagation( )
{
  if the gate output does not have event {
   if the gate is GS {
    if one of the input is having a glitch {
      glitch will propagate to output;
    }
   }
   else if the gate is PS {
    if((only one CO input has a glitch) && (other
        CO inputs if any have events) && (no NC
        input is having event)) {
      glitch will propagate to the output;
    }
   }
   else if gate is OPS {
```

```
    if at least one input has a glitch, output will have
        a glitch;
  }
  else if gate is IS {
    if input has a glitch, output will have a glitch;
  }
 }
}
```

Pseudo Code for Glitch Propagation

```
backtrace_robust_path(n)
int n;
{
  if gate is not a PI {
    if gate is PS {
      for each CO input (i) {
        mark (i) as robust; backtrace_robust-
            _path(i);
      }
      for each NC input (j) with glitch {
        mark (j) as nonrobust; backtrace_nonro-
            bust_path(j);
      }
    }
    else if gate is GS {
      if only one NC input (m) has event {
        if none of the input has a glitch {
          mark input (m) as robust; backtrace_ro-
              bust_path(m);
        }
        else if at least one input has a glitch {
          mark input (m) as nonrobust; back-
              trace_nonrobust_path(m);
        }
      }
    }
    else if gate is OPS {
      if only one input (p) has event {
        if there is no glitch on the other input {
          mark input (p) as robust; backtrace_ro-
              bust_path(p);
        }
        else if there is a glitch on the other input {
          mark both inputs as nonrobust;
          for each input (q) {
            backtrace_nonrobust_path(q);
          }
        }
      }
    }
    else if gate is IS {
      mark input (r) as robust; backtrace_robust-
          _path(r);
    }
  }
}
```

Pseudo Code for the Recursive Backtrace Procedure of Robust Paths

```
backtrace_nonrobust_path(n)
int n;
{
  if gate is not a PI {
    if there is no event but a glitch on gate output {
      if gate is PS {
        if all CO inputs have events & only one NC
            input (i) has event {
          mark input (i) as nonrobust; backtrace_n-
              onrobust_path(i);
        }
        else if no events in the inputs & only one
            CO input (j) has a glitch {
          mark input (j) as nonrobust; backtrace_n-
              onrobust_path(j);
                    }
      }
      else if gate is GS {
        for each input (k) having a glitch {
          mark (k) as nonrobust; backtrace_nonro-
              bust_path(k);
        }
      }
      else if gate is OPS {
        for each input (l) having an event or glitch
            {
          mark (l) as nonrobust; backtrace_nonro-
              bust_path(l);
        }
      }
      else if gate is IS {
```

```
        mark input (m) as nonrobust; backtrace_n-
            onrobust_path(m);
      }
    }
    else if gate output has an event {
     if gate is PS {
      for each CO input (i) {
        mark (i) as nonrobust; backtrace_nonro-
            bust_path(i);
      }
      for each NC input (j) with glitch {
        mark (j) as nonrobust; backtrace_nonro-
            bust_path(j);
      }
     }
     else if gate is GS or OPS {
      if only one NC input (m) has event {
          mark input (m) as nonrobust; back-
              trace_nonrobust_path(m);
        }
     }
     else if gate is IS {
      mark input (q) as nonrobust; backtrace_non-
          robust_path(q);
     }
    }
  }
}
```

Pseudo Code for the Recursive Backtrace Procedure of Nonrobust Paths

Authors' Biographies

Ananta K. Majhi received the B.S. degree in Electrical Engineering from College of Engineering and Technology, Bhubaneswar, India, in 1988 and M.Tech degree in Electronics Engineering from Banaras Hindu University, Varanasi, India, in 1991. Presently he is persuing his Ph.D degree in the Department of Electrical Communication Engineering at Indian Institute of Science, Bangalore, India. His research interests include test genaration and fault simulation for VLSI circuits, CAD of VLSI systems, and parallel and distributed computing.

James Jacob received the B.E. and Ph.D degrees in Electronics and Communication Engineering from Indian Institute of Science, Bangalore, India, in 1983 and 1988, respectively. Currently he is an Assistant Professor in the Department of Electrical Communication Engineering at Indian Institute of Science. His research interests are in the areas of VLSI testing, simulation and logic synthesis.

L. M. Patnaik obtained his Ph.D in 1978 in the area of Real-Time Systems and D.Sc. in 1989 in the area of Computer Systems and Architectures. He has published over 120 papers in refereed International Journals and over 130 papers in refereed International Conference Proceedings, in the areas of Computer Architecture, Parallel and Distributed Computing, VLSI CAD, Neural Networks, Real-Time Systems, and Genetic Algorithms. He is a Fellow of the IEEE. He serves on the Editorial Boards of several International Journals including VLSI Design, The Computer Journal, International Journal of High Speed Computing, Parallel Algorithms and Applications, and Computer-Aided Design.