CRITICAL PATH TRACING - AN ALTERNATIVE TO FAULT SIMULATION

M. Abramovici
P. R. Menon
D. T. Miller

Bell Laboratories
Naperville, Illinois 60566

ABSTRACT

We present an alternative to fault simulation, referred to as *critical path tracing*, that determines the faults detected by a set of tests using a backtracing algorithm starting at the primary outputs of a circuit. Critical path tracing is an approximate method, but the approximations introduced occur seldom and do not affect its usefulness. This method is more efficient than conventional fault simulation.

1. INTRODUCTION

Fault simulation is performed for the following applications:

- grade a test by determining its fault coverage
- construct a fault dictionary
- in the context of test generation, determine a yet undetected fault as the next target for test generation
- post-test diagnosis [ArWa81].

There are three general methods for fault simulation, namely parallel, deductive, and concurrent [BrFr76]. In addition, two other methods deal only with combinational circuits, namely single fault propagation [Roth67, Ozgu79] and Hong's method [Hong78]. A combination of these two techniques is used in the PODEM-X test generation system [Goel80b, GoRo81]. Specialized methods for combinational circuits are justified by the widespread use of design for testability techniques, such as LSSD [EiWi77], that transform a sequential circuit into a combinational one for testing purposes. Even for combinational circuits, the time spent in fault simulation alone is proportional to n^2, where n is the number of gates in the circuit [Goel80a]. The increase in the gate count in VLSI circuits requires more efficient methods for test evaluation and test generation.

In this paper we present an alternative to fault simulation, referred to as *critical path tracing*. It consists of simulating the fault-free circuit (true-value simulation) and using the computed signal values for tracing paths from primary outputs (POs) towards primary inputs (PIs) to determine the detected faults. Compared with conventional fault simulation, the distinctive features of critical path tracing are:

- *it directly identifies the faults detected by a test*, without simulating the set of all possible faults. Hence all the work involved in propagating the faults that are not detected by a test towards the POs is avoided.
- *it deals with faults only implicitly*. Therefore we no longer need fault enumeration, fault collapsing, fault partitioning (for multipass simulation), fault insertion and fault dropping.
- *it is based on a path tracing algorithm* that does not require computing values in the faulty circuits by gate evaluations or fault list processing
- *it is an approximate method.*

Consequently, critical path tracing is faster and requires less memory than conventional fault simulation. The approximation occurs seldom and consists in not marking as detected some faults that are actually detected by the evaluated test. We shall show later that this approximation does not affect the usefulness of the method.

Critical path tracing can be extended to synchronous sequential circuits using an iterative array model [BrFr76]. In this paper we will restrict ourselves to combinational circuits.

2. MAIN CONCEPTS

2.1 Criticality and Sensitivity

We will use the concept of a *critical value* as defined in [Wang75].

Definition 1: A line l has a *critical value* v in the test (vector) t iff t detects the fault l $s\text{-}a\text{-}\bar{v}$. A line with a critical value in t is said to be *critical* in t.

Note that unlike the D notation of the D-Algorithm [Roth67], a critical value always indicates the detection of a fault.

Clearly, the POs are critical in any test. Our test evaluation method consists of determining paths of critical lines, called *critical paths*, by a backtracing process starting at the POs. Finding the critical lines in a test t, we immediately know the faults detected by t.

Definition 2: A gate input i is *sensitive* if complementing the value of i changes the value of the gate output.

Critical path tracing starts after the true-value simulation of the circuit for a test t has been performed. To aid the backtracing, we mark the sensitive gate inputs during the true-value simulation. The sensitive inputs of a unate gate with two or more inputs are easily determined as follows:

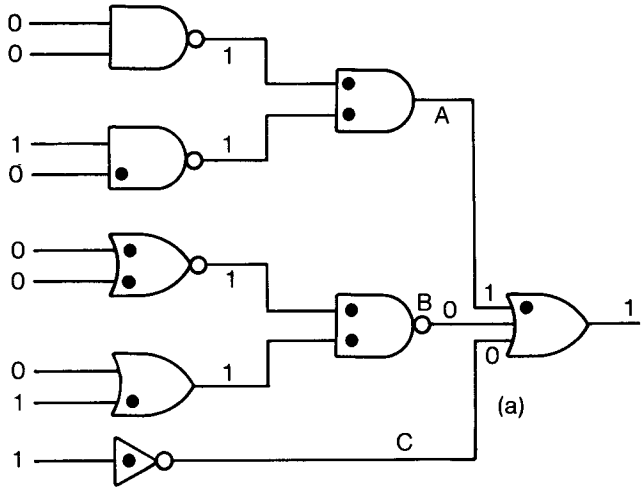
1) if only one input i has a Dominant Logic Value (DLV), then i is sensitive. (AND and NAND have DLV 0; OR and NOR have DLV 1)
2) if all inputs have value $\overline{DLV}$, then all inputs are sensitive,
3) otherwise no input is sensitive.

The marking of the sensitive gate inputs during true-value simulation involves little overhead, since scanning the gate inputs for DLVs is inherent in commonly used methods for gate evaluation.

2.2 Fanout-Free Circuits

First we illustrate critical path tracing in a fanout-free circuit, using the example in Figure 1. Figure 1a shows the results of true-value simulation with the sensitive inputs marked by dots. Figure 1b shows the critical paths by heavy lines. For a fanout-free circuit (which always has a tree structure), critical path tracing is a simple tree traversal procedure that marks as critical and recursively follows in turn every sensitive input of a gate with critical output. This is based on the obvious fact that if a gate output is critical, then its sensitive inputs, if any, are also critical. For the example in Figure 1, note how critical path tracing completely ignores the part of the circuit bordered by the lines B and C, since working backwards from the PO it first determines that B and C are not critical.

0738-100X/83/0000/0214$1.00



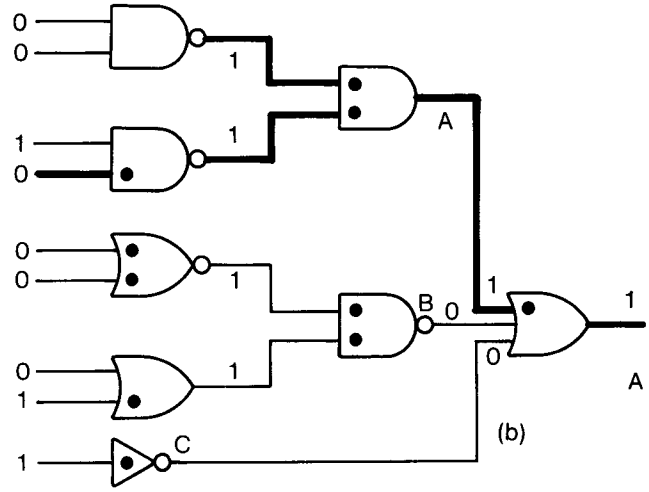


Figure 1. Critical Path Tracing in a Fanout-Free Circuit

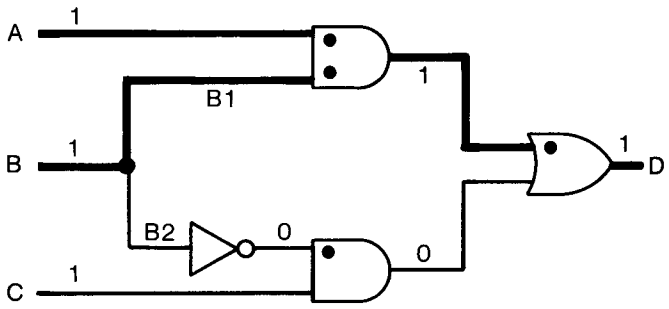
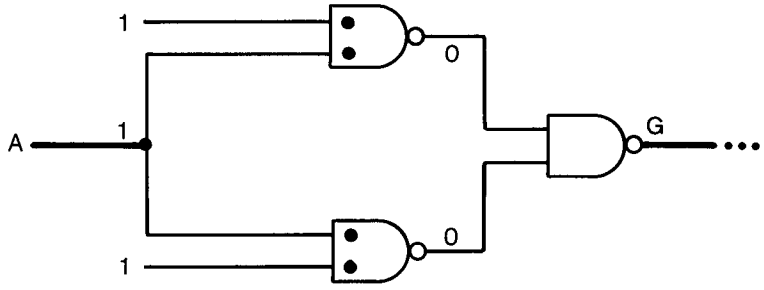


Figure 2. Example of Self-Masking



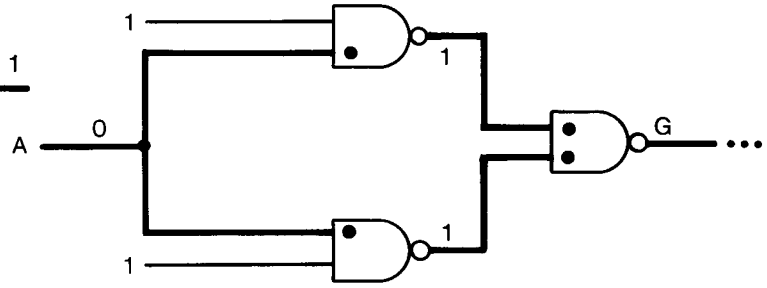


Figure 3. Impact of Multiple Path Sensitization on Critical Path Tracing

2.3 Effects of Reconvergent Fanout

Now we shall discuss the general case of a circuit with reconvergent fanout. Under the stuck fault model, for a signal with fanout we distinguish between its *stem* and its *fanout branches* (FOBs). For example, in Figure 2 the lines *B1* and *B2* are FOBs of the stem *B*. The difficulty in extending critical path tracing to circuits with reconvergent fanout is determining when a stem is critical, given that at least one of its FOBs is critical. In Figure 2 we cannot extend the critical path from *B1* to *B* because the effects of the fault *B s-a-0* propagate on two paths with different inversion parities such that they cancel each other when reconverging at the gate D. This phenomenon is referred to as *self-masking*.

Following Hong's approach [Hong78], we separate the test evaluation problem into two subproblems: one deals with *fanout-free regions* (FFRs) of the circuit, and the other with stems. The strategy adopted in [Hong78] is to first determine the detectability of the stem faults by explicit fault simulation, then trace critical paths in every FFR whose stem fault has been detected. This mixed strategy represents an improvement compared with the explicit simulation of all faults. Our strategy is to determine detectability of the stem faults as they are reached during backtracing by a novel type of analysis that, as we will show later, is much more efficient than their explicit fault simulation. This analysis is described in the next section.

Another effect of reconvergent fanout is that faults that are detected only by multiple-path sensitization may be declared to be undetected, as illustrated in Figure 3a. Here none of the inputs of the gate *G* is critical, but the fault on *A* is detected by double path sensitization. As critical path tracing involves establishing unbroken chains of critical lines, it will not recognize the stem *A* as critical in that test; therefore it is only an approximate method. We shall discuss the implications of this approximation in Section 4.

Note that the case when multiple paths are sensitized from a stem such that we have $\overline{DLV}$ values at a reconvergence gate [see Figure 3(b)] does not pose any problem for our method.

3. ALGORITHM FOR CRITICAL PATH TRACING

3.1 Preprocessing

First we preprocess the circuit to determine its *cones*, where a cone is the subcircuit feeding one PO. We represent a cone as an interconnection of FFRs. Figure 4(b) shows these structures for the

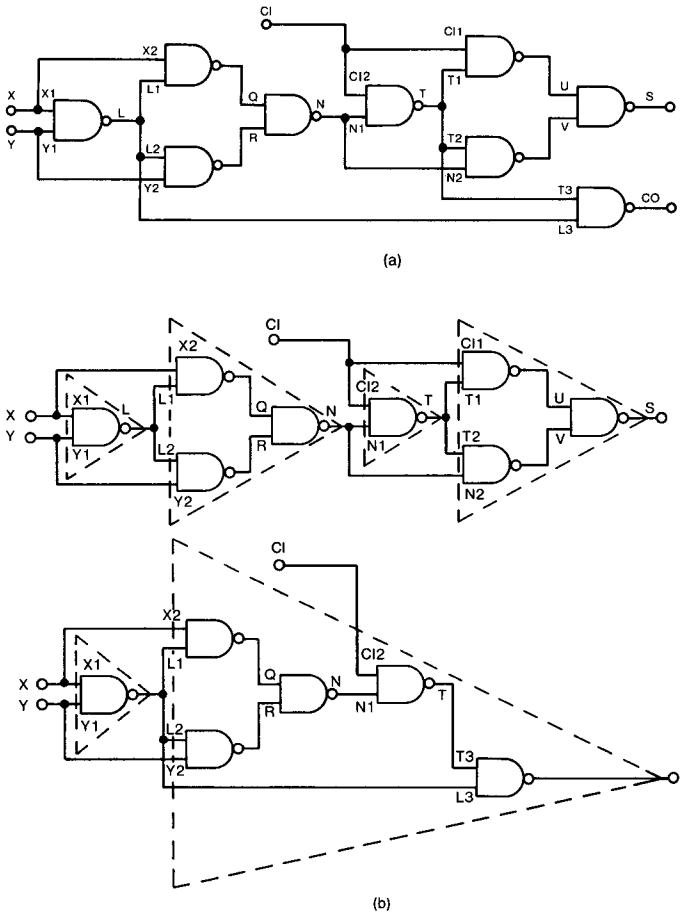


Figure 4(a). Example Circuit
(b). Partition Into Cones and FFRs

adder in Figure 4(a). The inputs of a FFR are FOBs and/or PIs. The output of a FFR is either a PO or a stem. Note that whether a line i is a stem may depend on the cone containing it. For example, N and CI are stems in the cone of S, but not in the cone of CO. Constructing cones and FFRs is commonly done as a preprocessing step in test generation for LSSD circuits [Yama78, Goel80b].

3.2 The Basic Structure

Figure 5 outlines the algorithm for evaluating a given test. It assumes that true-value simulation, including the marking of the sensitive gate inputs, has been performed. The algorithm processes every cone starting at its PO and alternates between two main operations: critical path tracing inside a FFR, represented by the procedure *Extend*, and checking a stem for criticality, done by the procedure *Critical*. Once a stem j is found to be critical, critical path tracing continues from j.

Figure 6 shows the recursive procedure *Extend(i)* that backtraces all the critical paths inside a FFR starting from a given critical line i by following lines marked as *sensitive*. *Extend* stops at FFR inputs and collects all the stems reached in the set *Stems_to_check*.

Every stem in *Stems_to_check* has at least one critical FOB. The algorithm always selects the highest level stem for analysis (i.e., the closest to the PO), and hence guarantees that the status (critical/non-critical) of all its FOBs is known. The key element in the algorithm is the routine *Critical(j)* that determines whether the stem j is critical.

3.3 Determining the Criticality of a Stem

To determine the criticality of a stem, we will find out whether self-masking occurs.

```
for every cone
  {Stems_to_check = φ
   Extend(PO)
   while (Stems_to_check ≠ φ)
      {j = highest level stem in Stems_to_check
       if (Critical(j)) Extend(j)
      }
  }
```

Figure 5. Algorithm for Evaluating One Test

```
Extend(i)
{mark i as critical
 if i is a FOB
   add stem(i) to Stems_to_check
 else
   for every input j of i
      if (sensitive(j)) Extend(j)
}
```

Figure 6. Backtracing inside a FFR

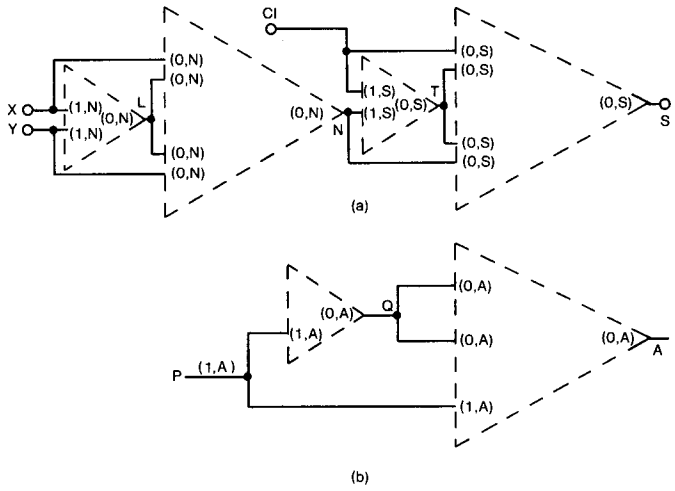


Figure 7. (p,l) Labeling

A Simple Case

First we will show a simple preprocessing technique that allows us to directly identify some stems as critical without any analysis. When a cone is built by topological backtracing starting from its PO, with every FFR input and output i we associate a label (p,l) defined as follows.

Definition 3: A line i has a label (p,l) if all paths from i to the PO of the cone pass through l, and every path from i to l has parity p.

Figure 7(a) shows the (p,l) labels in the cone of S. Clearly all inputs of a FFR get the same l. If all the FOBs of a stem j have the same (p,l) label, then this is assigned to the stem; otherwise j is assigned the label $(0,j)$. The following lemma allows us to mark stems as critical without additional analysis.

Lemma 1: If the label of a stem j is (p,l), where $l \neq j$, then whenever j is reached by backtracing it can always be marked as critical.

Proof: Line j can be reached during backtracing only if l has already been proved to be critical. Since all the paths from j to l have the same parity, self-masking cannot occur between j and l. Hence the fault on j propagates to l, and because l is critical, it also propagates to the PO. Therefore j is also critical. □

Figure 7(a) shows the (p,l) labels for the S cone of Figure 4. According to Lemma 1, *Critical* will immediately return TRUE for the

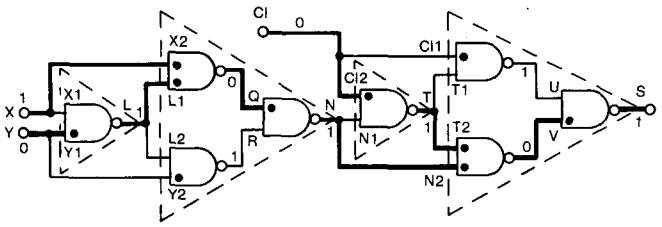


Figure 8. Complete Critical Path Tracing in a Cone

stems *L* and *T*. Note that the labeling is done per cone, as *L* has to be checked for criticality in the cone of *CO*. In our example, both *L* and *T* have all their FOBs feeding the same FFR; Figure 7(b) shows a different situation in which a stem (*P*) need not be checked.

General Case

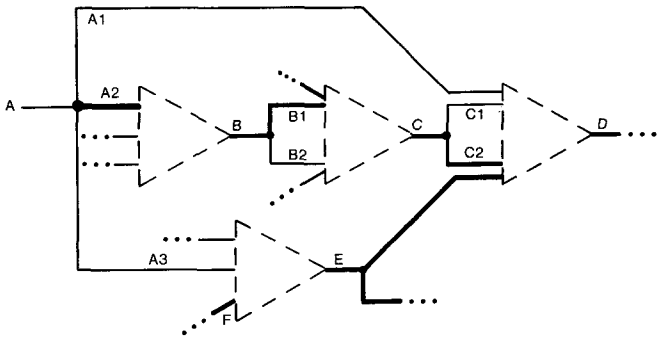
To illustrate the principle used by *Critical(j)* to check a stem *j* that does not satisfy Lemma 1, consider the cone of *S* for the test (*X,Y,CI*) = 100 (see Figure 8). After performing *Extend(S)*, we determine that *T2* and *N2* are critical. At this point *Stems_to_check* = {*T,N*}. *Critical(T)* returns TRUE immediately and *Extend(T)* flags *CI2* as critical. Now *Stems_to_check* = {*N,CI*} and *N* is selected for analysis. *For the purpose of explanation only*, suppose we insert a *s-a-0* fault on *N*. The effects of this fault start propagating along two paths, one starting at the critical FOB (*N2*), and the other at the non-critical one (*N1*). Self-masking may occur only if the propagation along the latter path, referred to as *potentially masking*, "kills" the propagation along the critical path as illustrated in Figure 2. It is easy to determine that this type of reconvergence does not occur in Figure 8, since the potentially masking path is immediately stopped at the gate *T*, as it enters the gate via an input not marked as sensitive. Hence we can immediately conclude that *N* is critical. We emphasize that propagation along the critical path beyond *N2* is not needed because we have determined that all the potentially masking fault effects have disappeared. This is similar to the concepts used in single fault propagation.

The Algorithm

Figure 9 outlines the procedure *Critical(j)*. First it examines the (*p,l*) label of the stem *j* to determine whether it satisfies Lemma 1. If not, the check for the criticality of *j* implicitly keeps track of the propagation of the fault effects of *j* along critical paths versus propagation on potentially masking paths. The two propagation frontiers are maintained in the sets *Prop_crit* and *Prop_noncrit*. These sets consist only of FOBs and stems. *FFRlist* is the set of FFRs directly fed by the FOBs in *Prop_crit* and *Prop_noncrit*. The algorithm always selects the lowest level FFR from *FFRlist* for checking, and hence it guarantees that the status of all the FFR inputs with respect to the propagation of the fault on *j* is known. The procedure *FFRcheck(i)* determines the propagation of the two frontiers through FFR *i*; hence it implicitly determines whether the fault effects arriving on FFR inputs reach its output. The salient feature of *FFRcheck(i)* is that it usually "jumps" from the inputs of FFR *i* directly to its output without tracing through any gate inside the FFR. The following example illustrates different cases of FFR jumping. (A detailed description of the algorithm is beyond the scope of this paper.)

```
Boolean Critical(j)
{if (l(j) ≠ j) return TRUE
 Prop_crit = {critical FOBs of j}
 Prop_noncrit = {non-critical FOBs of j}
 build FFRlist
 repeat
     {i = lowest level FFR in FFRlist
      FFRcheck(i)
      if (Prop_crit ≠ φ) return FALSE
     }
 until (Prop_noncrit = φ and |Prop_crit| = 1)
 return TRUE
}
```

Figure 9. Procedure to Determine Criticality of a Stem



Step	*Prop_crit*	*Prop_noncrit*	*FFRlist*
1	*A2*	*A1,A3*	*B,E,D*
2	*B*	*A1,A3*	*E,D*
3	*B1*	*A1,A3,B2*	*E,D*
4	*B1*	*A1,B2*	*C,D*
5	*C*	*A1*	*D*
6	*C2*	*A1,C1*	*D*
7	*D*	ϕ	

Figure 10. Illustration for the execution of *Critical(A)*

Example

Suppose that the lines currently identified as critical are those shown in Figure 10 and now the problem is to determine whether *A* is critical. The table traces the execution of *Critical(A)*.

Step 1: Starting with *Prop_crit*={*A2*} and *Prop_noncrit*={*A1,A3*}, the first FFR checked is *B*, which is reached only by lines in *Prop_crit* (*A2*). Here no masking can occur inside the FFR, hence the fault effect on *A2* propagates on *B*; therefore we add *B* to *Prop_crit*.

Step 2: The FOBs of *B* (*B1* and *B2*) are added to the two frontiers based on their previously determined criticalities.

Step 3: Next we analyze FFR *E*, which is reached only by one line in *Prop_noncrit* (*A3*). Since *E* has a critical input (*F*), the propagation from *A3* to *E* is bound to stop inside the FFR. The reason is that there must exist a gate *G* where the path from *A3* will converge with the critical path from *F*, such that the critical input of *G* has a DLV.

Step 4: FFR *C* is reached by lines in both frontiers - *B1* in in *Prop_crit* and *B2* in *Prop_noncrit*. But the propagation from *B2* cannot stop the propagation from *B1*, because the algorithm had already determined this when it established that *B* was critical. Therefore we add *C* to *Prop_crit*.

Step 5: The FOBs of *C* are added to the two frontiers.

Step 6: The criterion we used to skip FFR *C* cannot be used to skip FFR *D*, because of the additional potentially masking line reaching it (*A1*); *A1* did not play any role in determining the criticality of *C*. Now we apply a different criterion to avoid tracing inside the FFR. Let *p(A1), p(C1), p(C2)* denote, respectively, the inversion parities between *A1, C1, and C2*, and *D*. Let *v(A1), v(C1)* and *v(C2)* be their current values. It can be shown that if $v(A1)\oplus p(A1)=v(C1)\oplus p(C1)=v(C2)\oplus p(C2)$, then masking cannot occur inside the FFR *D*. Assuming that this relation is satisfied in Figure 10(a), we add *D* to *Prop_crit*.

Step 7: At this point *Prop_noncrit*=ϕ and |*Prop_crit*|=1 (since *Prop_crit*={*D*}); hence the procedure returns identifying *A* as critical. □

One may wonder why the condition *Prop_noncrit*=ϕ alone is not sufficient for stopping the forward tracing. The answer is provided by the example in Figure 11, which shows a stem (*X*), all of whose FOBs are critical (hence *Prop_noncrit*=ϕ). However, contrary to one's

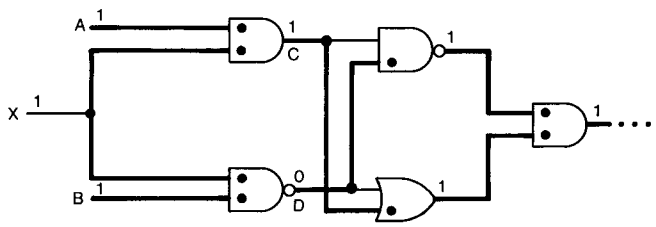


Figure 11. A Noncritical Stem With All Its FOBs Critical

intuition, X is not critical. The reason is that the potentially masking lines appear after propagation through C and D. Requiring $|Prop\_crit|=1$ *and* $Prop\_noncrit=\phi$ guarantees that no masking can further occur.

If none of the "FFR jumping" criteria applies, then path tracing proceeds inside a FFR. This is similar to the path tracing at the FFR level, i.e., propagating the frontiers *Prop_crit* and *Prop_noncrit* in a breadth-first manner. Note that no gate evaluations are involved, and even the gate types and the signal values are unnecessary for this analysis; the only information needed is provided by the *sensitive* markings.

We emphasize that *in most cases self-masking does not occur* and the propagation of the potentially masking fault effects is "short-lived"; therefore, little effort is usually needed to determine that a stem is critical.

3.4 Further Improvements

We have identified (but not yet implemented) two additional speed-up techniques whose objective is to reduce the area of the circuit where critical path tracing is performed. As presented in the previous sections, in every test the algorithm starts at the POs and extends the criticality of the POs as far as possible towards the PIs.

The goal of the first speed-up technique is to *determine start lines* different from POs for critical path tracing. This is based on the fact that the criticality of some lines may remain the same in consecutive tests. As an extreme example, consider the case of a cone whose PIs have the same value in two consecutive tests; obviously the critical path tracing in the second test would be completely redundant. On the other hand, if the PO value changes, the critical path tracing must start from the PO. In the general case falling between these two extremes, we can always identify a set of start lines, such that the criticality in the test t_{i+1} of the lines in the area between the start lines and the PO is the same as in the test t_i (see Figure 12a). The identification of potential start lines is done during true-value simulation by simply marking the gates that satisfy the following conditions (see Figure 12b):

(1) the gate is evaluated as a result of some input change(s) but its output does not change, and

(2) the gate output has been critical in the previous test.

The critical trace starts at the highest level potential start line, which is always a true start line. The true start lines can be determined by processing the potential start lines in their decreasing level order.

The goal of the second speed-up technique is to *determine stop points* different from the PIs for critical path tracing. The basic idea is illustrated in Figure 12(c). Suppose that the values in the first test are as shown. Then in any further test in which D has a critical 0 value, continuing the backtracing from D is useless, since all the faults that could make D=1 have already been detected. Thus D can be flagged as a *0-stop line*, and the same reasoning applies to E as well. Moreover, after two other tests that apply (A,B,C)=010 and 100, D becomes a *1-stop line* and then, after (A,B,C)=101, E also becomes a 1-stop line. During the evaluation of a set of tests, the stop lines generally keep advancing towards the POs, so the area in which the critical paths are extended becomes smaller and smaller. This is similar to fault dropping in fault simulation.

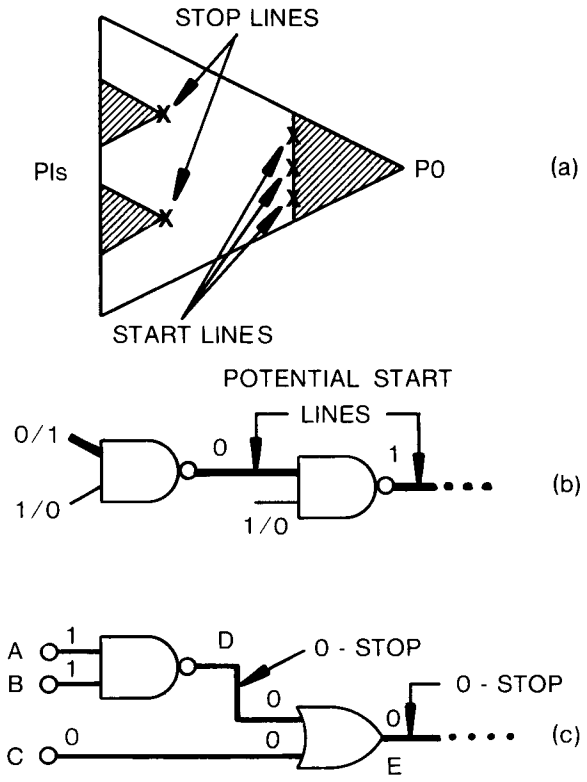


Figure 12(a). Start Lines and Stop Lines in a Cone
(b). Determining Potential Start Lines
(c). Determining Stop Lines

3.5 Summary of the Method

Our method consists of the following steps:

(1) *Preprocessing* the circuit model, to determine its cones and FFR structure and the (p,l) labels.

(2) *True value simulation* of one test and identification of the sensitive gate inputs.

(3) *Critical Path Tracing*, which is a backtracing procedure that identifies the critical lines (and hence the detected faults) in the test simulated in step 2.

Steps 2 and 3 are repeated for every test in the set of tests under evaluation.

4. IMPACT OF APPROXIMATION

In Section 2.3 we pointed out that critical path tracing is an approximate method. The approximation consists in not marking as detected a stem fault actually detected by multiple path sensitization (MPS) such that we have DLVs at the reconvergence gate (see Figure 3a). In this section we shall analyze the impact of the approximation on the test evaluation process.

First note that there are many circuits whose structure and function preclude the occurrence of this phenomenon; the adder in Figure 4a provides such an example. Second, in circuits where it can occur we still need a test that will create the necessary conditions for that type of MPS. How likely is this to happen depends on the way the evaluated test is generated. If a test is randomly generated, then there is a non-zero probability that it may happen. However, a test generated by random single-path sensitization (SPS) [GoRo81] or by any deterministic algorithm is unlikely to sensitize multiple paths. All test generation algorithms first try to sensitize only one path and attempt MPS only when the target fault cannot be detected by SPS. Experimental results presented in [Cha78] confirm that cases where

MPS is required for fault detection occur seldom in practical circuits. Therefore, the approximation will occur seldom, and the affected tests (if any) are likely to be randomly generated.

The impact of the approximation should be analyzed in the context of evaluating a set of tests, rather than a single test.

For *test grading*, it does not matter in which test a fault is detected, but whether it is detected by any of the tests in the evaluated test set. Even if a fault is detected by MPS in one test, there is a good chance that it will be detected by SPS in other tests. Thus the only faults not recognized as detected are those that are detected only by MPS. If this unlikely situation occurs, then the computed fault coverage will be slightly pessimistic.

In the context of *test generation* the function of critical path tracing is to help in the selection of the next target fault. If the approximation occurs, the test generation algorithm may be needlessly asked to generate a test for an already detected fault, say f. In most cases f will be detected by SPS, and then critical path tracing will mark f as detected. If the test generator has obtained a test for f, but f is not marked as detected by critical path tracing, then we directly mark the corresponding line as critical and restart critical path tracing from that line. Thus the approximation has practically no impact on the test generation process.

The use of critical path tracing for *constructing a fault dictionary* may result, if the approximation does occur, in some loss of diagnostic resolution. This will happen only when a missed detected fault resides in a different replaceable unit from its equivalent fault(s) marked as detected. (For example, in Figure 3a the missed fault A s-a-0 is equivalent to G s-a-0, which is marked as detected.) This is quite an unlikely case in VLSI circuits, characterized by large replaceable units. However, this potential loss of resolution is more than compensated by the gain in diagnostic resolution obtained because critical path tracing does not require fault dropping. Fault dropping, done to contain the high cost of conventional fault simulation, results in many non-equivalent faults being represented by the same fault signature.

The applicability of the fault dictionary approach to the diagnosis of VLSI circuits is becoming increasingly questionable; this approach is being replaced by post-test diagnosis techniques [Hsu81, ArWa81] that attempt to directly identify the faults that are consistent with the entire obtained result. The critical path tracing is an ideal tool for this type of analysis, since

1) tracing from a "failing" PO directly identifies a set of possible fault locations,

2) tracing from a "passing" PO identifies a set of faults that are *not* present in the circuit under test.

Successive applications of these two criteria, coupled with a judicious selection of the POs and tests in which critical path tracing is performed, leads to an efficient post-test diagnosis method. Although some loss of resolution may occur due to the approximation, such a technique would be more efficient than the post-test diagnostic methods based on topological path backtracing and conventional fault simulation [Hsu81, ArWa81].

5. COMPARISON WITH FAULT SIMULATION

Goel has shown that the deductive method is faster than parallel fault simulation [Goel80a]. Experimental results presented in [Ozgu79] indicate that the deductive and single fault propagation methods are comparable in speed. Hong estimates that his method is faster than deductive simulation [Hong78]. We shall compare critical path tracing with Hong's method and with concurrent fault simulation. The advantages of our method compared to the explicit simulation of all the stem faults, as done by Hong, are:

(1) We analyze only a subset of the stem faults, namely only those stems reached by backtracing; some of them can be immediately identified as critical due to their (p,l) labels (determined by preprocessing).

(2) For every stem checked for criticality, the paths involved in our forward propagation are usually a small subset of the paths involved in the explicit simulation of a stem fault.

(3) Along the paths traced forward, our method often jumps directly from a FFR input to its output.

(4) Even inside a FFR our method does only path tracing and does not involve gate evaluations.

Preliminary results of our initial implementation of critical path tracing (without the speed-up techniques mentioned in Section 3.4) showed a 20 to 40 percent speed-up compared to concurrent fault simulation with fault dropping after first detection. Without fault dropping in concurrent simulation, critical path tracing was 6 to 8 times faster.

6. CONCLUSIONS

The key factors contributing to the increased efficiency of critical path tracing compared to fault simulation are as follows:

- it deals directly only with the detected faults rather than all possible faults,
- it deals with faults only implicitly rather than explicitly,
- it is an approximate rather than an exact technique.

The approximation introduced is pessimistic and consists in not marking as detected some faults detected by multiple path sensitization with DLVs at a reconvergence gate. This phenomenon occurs seldom. However, we have shown that the approximation does not affect the test generation process and has practically no impact on the other applications of critical path tracing.

The advantages of test evaluation by critical path tracing over conventional methods strongly suggest that solutions to the VLSI testing problems should be based on approximate algorithms that are fast and generally accurate. A gain of one order of magnitude in execution time is much more important than an "exact" algorithm whose only advantage is that it is capable of correctly processing situations that occur seldom. Furthermore, one can question the wisdom of using exact and costly algorithms for an approximate fault model.

Acknowledgement: We gratefully acknowledge the contributions of J. J. Kulikowski in the development of the data structures for critical path tracing.

REFERENCES

[ArWa81] Y. Arzoumanian and J. Waicukauski, "Fault Diagnosis in an LSSD Environment," *Proc. 1981 International Test Conference*, pp. 86-88, October, 1981.

[BrFr76] M. A. Breuer and A. D. Friedman, "*Diagnosis and Reliable Design of Digital Systems*," Computer Science Press, 1976.

[Cha 78] C. W. Cha, W. E. Donath and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Trans. Comput.*, Vol. C-27, No. 3, pp. 193-200, March, 1978.

[EiWi77] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proc. 14th Design Automation Conference*, pp. 462-468, June, 1977.

[Goel80a] P. Goel, "Test Generation Costs Analysis and Projections," *Proc. 17th Design Automation Conference*, pp. 77-84, June, 1980.

[Goel80b] P. Goel, *et al*, "LSSD Fault Simulation Using Conjunctive Combinational and Sequential Methods," *Proc. 1980 Test Conference*, pp. 371-376, November, 1980.

[GoRo81] P. Goel and B. C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," *Proc. 18th Design Automation Conference*, pp. 260-268, June, 1981.

[Hong78] S. J. Hong, "Fault Simulation Strategy for Combinational Logic Networks," *Proc. 8th International Symp. on Fault-Tolerant Computing*, pp. 96-99, June, 1978.

[Hsu81] F. Hsu, P. Solecky, and R. Beaudoin, "Structured Trace Diagnosis for LSSD Board Testing - An Alternative to Full Fault Simulated Diagnosis," Proc. 18th Design Automation Conference, pp. 891-897, June, 1981.

[Ozgu79] F. Ozguner, W. E. Donath and C. W. Cha, "On Fault Simulation Techniques," *J. of Design Automation and Fault Tolerant Computing*, Vol. 3, No. 2, pp. 83-92, April, 1979.

[Roth67] J. P. Roth, W. G. Bouricius and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. Comput*, Vol. EC-16, No. 10, pp. 567-579, October, 1967.

[Wang75] D. T. Wang, "An Algorithm for the Generation of Test Sets for Combinational Logic Networks," *IEEE Trans. Comput.*, Vol. C-24, No. 7, pp. 742-746, July, 1975.

[Yama78] A. Yamada, *et al*, "Automatic System Level Test Generation and Fault Location for Large Digital Systems," *Proc. 15th Design Automation Conference*, pp. 347-352, June, 1978.