RIDDLE: A FOUNDATION FOR TEST GENERATION ON A HIGH LEVEL DESIGN DESCRIPTION

by

G.M. Silberman and I. Spillinger

Technical Report #503
March 1988
abstract}

We present a formal approach to the analysis of a VLSI design described at the high level, which produces information conducive to the acceleration of test generation algorithms. This analysis yields as its main product, information which can be used to reduce the amount of effort expended during backtracing, by guiding this process towards decisions (assignments) less likely to cause conflicts, and minimizing the amount of work between backtracks. RIDDLE, an algorithm which performs this analysis in time linear in the number of signals, is introduced. Experimental results for the special case of combinatorial gate level designs are also given.

keywords
backtracing, design decomposition, high level description, test generation.

1. Introduction

The increase in VLSI circuit complexity has turned test generation into a major bottleneck in the design cycle. One way to tackle the complexity of test generation for these circuits, is to use a higher level of abstraction in the design description, thus reducing the amount of detail to be dealt with. In this work, we assume that the design is described as a network of interconnected functional blocks (e.g., adders, multiplexers, etc), as would be the case of designs using a library approach [1,2].

Many algorithms for test pattern generation using a gate level description of a design have appeared in the literature. Most of these algorithms have one thing in common, namely a search procedure, which is usually implemented as a backtracing process (e.g., [3-6]). This process is used to identify conditions or constraints on the design’s inputs, which are necessary to excite a fault and propagate its effect to an observable point. Furthermore, extensive research has been conducted in the area of improving the performance of the backtracing process at the gate level (e.g., [5-7]). On the other hand, although several attempts have been made to produce similar test pattern generation algorithms for designs expressed at a higher level (for a survey see [8]), none of these have addressed the efficiency aspect of the backtracing process.

In the following, we take a backtracing-based test pattern generation algorithm for a high level design description [9], and we show how pre-processing of the design can improve the algorithm’s performance. This pre-processing, performed by RIDDLE (Reconvergence-Independent Design Decomposition and Labeling of Edges), generalizes and extends the concepts in FAN [5], LAMP2 [6] and TOPS [7], so that they apply...
to the high level design description, and its results can be used to reduce the amount of effort expended by guiding the backtracing process along two separate lines. First, the design is decomposed into backtrace-independent segments, so that less work is done in-between backtracks caused by conflicts in the justification of values. Second, whenever several possibilities for justifying a value exist, we set the order in which choices are made, so that conflicts are less likely to happen, and thus the number of backtracks is reduced.

In FAN [5], backtracing is allowed to stop at certain internal lines (called head lines), which separate fanout-free segments of the design which are directly fed by primary inputs. Since a given value can always be justified on these lines, there is never a need to backtrack through these segments. Head lines effectively partition the circuit by removing the fanout-free regions which are rooted at these lines, from consideration during backtracing. In [6], LAMP2 extends the concept of head lines in FAN to that of backtrace-stop lines. This extension allows the inclusion of fanout branches which do not reconverge (as determined by the primary outputs they reach), among those lines which partition the circuit. Furthermore, they distinguish between lines for which backtracing can be stopped, according to the (0,1) value being assigned to them, thus allowing also to make choices which are likely to cause conflicts. TOPS [7] extends the definition of head lines, to include points of complete reconvergence, which can be directly identified by examining the topology of the design.

The next section presents the basic definitions to extend the concept of backtrace-stop lines, from the gate level to a high level design description. This extended concept includes all lines which can be assigned without conflicts, including some which are located after the appearance of reconvergent fanout nodes in the design. In section 3, we formally introduce the guidelines for establishing the order in which alternatives are chosen, so as to reduce the possibility for conflicts. Section 4 presents RIDDLE, an algorithm in time linear in the number of signals for the pre-processing of the design, according to our definitions.

In order to evaluate the advantages of RIDDLE, and provide a meaningful comparison to other methods for test pattern generation, we tested our algorithm on the special case in which each block represents a Boolean gate. Experimental results obtained on designs from [10] are given in section 5.

2. Design Decomposition

In this section we introduce the basic concepts which allow us to decompose a design into backtrace-independent segments. This partitioning imposes a certain order on the backtracing process, in which a segment is completely dealt with (including resolution of any conflicts within it), before any other segment is chosen for processing. In this manner, the assignments being undone during backtracks (because of conflicts) are all restricted to those within the segment being processed.

First, we define a directed graph $G(V, E)$ of the given design, where the set of nodes $V$ is defined as $V = V_{PI} \cup V_{PO} \cup V_{FO} \cup V_{BL}$, where:

- $v \in V_{PI}$ if $v$ is a Primary Input (or other directly controllable point) of the design,
- $v \in V_{PO}$ if $v$ is a Primary Output (or other directly observable point) of the design,
- $v \in V_{FO}$ if $v$ is a Fanout point in the design,
- $v \in V_{BL}$ if $v$ is a Functional Block in the design.

The edges $e \in E$ represent the signals in the design, and their orientation shows the direction in which information flows in the design, from the source node $source(e)$ to the drain $drain(e)$. Also, for each node $v \in V$, we define the sets of edges $outedges(v) = \{ e | source(e) = v \}$ and $inedges(v) = \{ e | drain(e) = v \}$. Accordingly, we define for each node $v$ the values $outdegree(v) = |outedges(v)|$ and $indegree(v) = |inedges(v)|$. If a primary input $v$ fans out to a number of blocks, we add a new node $v'$ and an edge $e$ such that $source(e) = v'$ and $drain(e) = v$. This means that $v \in V_{PO}$, and clearly the new node $v' \in V_{PI}$. Thus, we get...
(∀v ∈ V_FO)[outdegree(v) = 1]. Figure 1 shows an example of a design, given as an interconnection of functional blocks, and its representation as a directed graph, according to the above description.

Figure 1. A Design (16-Input Priority Encoder) and Its Directed Graph

In the following we assume that the digraph G is acyclic, which is the case for combinatorial designs, or those based on scan-in scan-out techniques (e.g., LSSD [11]).

Definition 1: Assignment of a value to an edge e ∈ E is the labeling of that edge with a symbol s ∈ {0, 1, X}. (We denote this label by value(e).) Assignment of values to a set of edges E′ ⊆ E, is the assignment of a value to each edge e ∈ E'.

Definition 2: The transfer function of a node v, denoted by θ_v, which relates the values assigned to the edges in the set inedges(v) with those assigned to the edges in the set outedges(v), is given by:

- For a node v ∈ V_FO (∀e ∈ outedges(v))[value(e) = value[l]], where {l} = inedges(v).
- For a node v ∈ V_BL, the transfer function is the function defined by the truth table relating the inputs to the outputs of the functional block represented by node v. For example, the transfer function for the least significant output bit from an adder is given by the exclusive-or of the two least significant input bits with the carry-in bit.
- Clearly, there is no transfer function for nodes v ∈ V_PI ∪ V_PO.

We refer to the relation from the values assigned to some subset of the edges in the set outedges(v), to those assigned to the edges in the set inedges(v) (if this is possible), as the inverse transfer relation of node v, and denote it by θ_v⁻¹. For example, for any set of values assigned to a set of output edges from an adder, we can find a set of values on its inputs (in the case of an adder, several such sets exist) which will produce the desired outputs, and thus the θ_v⁻¹ exists for all adder output values. On the other hand, for a functional block representing a BCD encoder, the inverse transfer relation is defined only for those outputs which are valid BCD codes.

Definition 3: A node v ∈ V_FO ∪ V_BL is said to have a conflict, if for the current assignment to some set of edges E′ ⊆ outedges(v), there is no possibility of assigning values to the set inedges(v) which is consistent with θ_v. (In other words, θ_v⁻¹ is not defined for the given assignment of values to E').
Definition 4: A valid assignment to a set of edges $E' \subseteq E$ is an assignment to the set $E'$ such that for all nodes $v \in \bigcup (\text{source}(e))$, $v$ has no conflict.

Definition 5: An edge $l$ influences an edge $e$, denoted by $l \rightarrow e$, if $\text{drain}(l) = \text{source}(e)$, and a change in the value of $l$ may cause a change in the value of $e$. For example, for a node $v \in V_F$ (a fanout node) and its stem edge $l$, $(\forall e \in \text{outedges}(v))[l \rightarrow e]$. Accordingly, if $l \rightarrow e$, we say that edge $e$ is influenced by edge $l$, and denote this by $e \leftarrow l$.

Definition 6: For each edge $l \in E$, we define its influence set as the set $\kappa(l) = \{e | l \rightarrow e\}$. It is clear from this definition, that the function of each block directly determines the influence sets for each of its inputs. For example, in the case of an adder, the influence set for the least significant bit of each addend contains all the outputs from the adder, while the influence set for the most significant input bits comprises only the two highest bits in the result (the most significant result bit and the carry-out).

Definition 7: The cone of edge $e$, $\kappa(e)$, is defined as the set containing itself and all the edges in the graph $G$ which influence $e$, i.e., $\kappa(e) = \{e\} \cup \{l \rightarrow e\}$, where $\rightarrow$ denotes the transitive closure of the relation $\rightarrow$. The cone of node $v$ is then given by $K(v) = \bigcup_{e \in \text{inedges}(v)} \kappa(e)$.

For example, in Figure 2 below, we find the following cones, $\kappa(e_1) = \{e_1, ..., e_6\}$, $\kappa(e_2) = \{e_2, e_3\}$, $\kappa(e_3) = \{e_3, ..., e_6\}$, $\kappa(e_4) = \{e_4, e_6\}$, $\kappa(e_5) = \{e_5\}$, and $\kappa(e_6) = \{e_6\}$.

Say that we want to backtrack from the inputs to block $bl$, to the primary inputs in the design. For convenience, we restrict our attention to the graph $\Gamma(V_r, E_r)$, where $E_r = K(bl)$ and $V_r = \{bl\} \cup \{v | v = \text{source}(e) \text{ and } e \in E_r\}$. We also use the notation $I_r(l) = \{e | l \rightarrow e \text{ and } e \in E_r\}$, which is the influence set, restricted to contain only those edges in the cone of block $bl$. (Similarly, the notations $\text{inedges}(v)$, $\text{indegree}(v)$, $\text{outedges}(v)$ and $\text{outdegree}(v)$, are used when the edges being considered are only those in the set $E_r$.) Clearly, $\Gamma$ is analogous to a single output circuit.

Definition 8: An edge $e \in E_r$ is a stop edge, if it is not singular (i.e., it can attain both logical values, 0 and 1), and for all valid assignments to the set of edges $E' = E_r - \kappa(e)$, there exists a valid assignment on $\kappa(e)$.

Stop edges are important in the backtracing process, since whenever we reach such an edge, we can always find a value for it (and the edges in its cone), which will not cause conflicts with the assignments to other edges being backtraced at this point. Notice that this does not mean that the value on the stop edge is independent from other values, but rather that for any assignment of values to the edges outside its cone, the necessary assignment to the stop edge can always be backtraced to the primary inputs. Therefore, whenever we encounter a stop edge during backtracing, we can postpone its processing until we have taken care of all possible conflicts among edges outside its cone. (The lines identified in [5,6,7] where backtrace can stop, regardless of the value being assigned to them, are special cases of this definition.)

The example below illustrates the definition of a stop edge.

Example 1: Let us consider two small circuits which share the same topology (i.e., their design graphs are identical), represent the same Boolean function ($C = A \land \overline{B}$), but differ in their implementation. The two circuits are shown in Figure 2 and Figure 3.
We want to determine whether the edge marked as $e_j$ in both circuits, is a stop edge. For the first circuit (refer to Figure 2), Table 1 shows the complete decision tree which shows all assignments to the edges in the circuit, needed to justify each assignment (0 or 1) to $e_j$. Table 2 shows the equivalent decision tree for the second circuit (refer to Figure 3).
We see that in the first circuit (Figure 2), edge $e_1$ is a stop edge, since for all valid assignments on $e_1$, $e_3$ there is a valid assignment on $\kappa(e_1) = \{e_1, e_4, e_5, e_6\}$. On the other hand, the corresponding edge is not a stop edge in the second circuit (Figure 3), since for the case of $\text{value}(e_1) = 1$ and $\text{value}(e_2) = 0$, there is no valid assignment on $\kappa(e_2)$.

We see from the above example, that whether an edge is or is not a stop edge, cannot be decided only by examining the topology of $\Gamma$. Furthermore, because of the dependency on the functionality of the design, this question can be answered (in the worst case) only by examining all valid assignments to the edges of the design graph. Therefore, in the interest of efficiency, in section 4 we show an algorithm which attempts to find as many stop edges in $\Gamma$ as possible, in time $O(|E|)$. 

In the following section we introduce other features conducive to better performance of the backtracing process, by reducing the number of times backtracking is needed. These features are based on characterizing the edges in the graph $\Gamma'$, according to whether they are less likely to cause conflicts for a certain assignment, and thus can be used to guide the backtracing process.

3. Avoiding Potential Conflicts

In this section we introduce a value-based approach for characterizing lines in the design. Whenever a certain assignment is being justified at the outputs from a block, and a number of choices exist in the form of assignments to the block’s inputs (as determined by the inverse transfer relation), this characterization serves to guide the selection towards those alternatives which are less likely to cause conflicts.

Definition 9: An edge $e \in E_y$ is said to be $b$-suspect, if $\exists l \ni e \in I_y(l)$ and there is an assignment to the set $I_y(l)$ where $\text{value}(e) = b$ such that $\theta_{\text{inv}(e)}$ is not defined.

For example, for all fanout nodes with more than one branch in $\Gamma'$, all these branches are suspect. (In our notation, $(\forall v \in V_{FO}$ and $\text{outdegree}(v) > 1)(\forall e \in \text{outedges}(v)[e \text{ is suspect}]).$ Notice that this does not imply that these edges are not stop edges, according to Definition 8, but since we limited ourselves to a linear time algorithm, we are not able to ascertain that every assignment for which there is no inverse relation will not occur during backtracing, and thus we are not allowed to mark these edges as stop edges. Furthermore, these edges are potential sources of conflict, and thus they are processed ahead of other edges during backtracing.

Definition 10: We say that an edge $e$ is $b$-safe ($b \in \{0,1\}$), if

- $\text{source}(e) \in V_p$; or
- $e$ is not $b$-suspect, and whenever $e$ is assigned the value $b$, there is a valid assignment to the set $\text{inedges}(\text{source}(e))$ for which
  1. all $b$-safe edges ($b \in \{0,1\}$) in the set which are not $\overline{b}$-safe, are assigned a value $b$ or $X$;
  2. all edges in the set which are neither safe (an edge is safe if it is both 0-safe and 1-safe), nor stop edges, are assigned $X$ values; and
  3. all other edges in the set are freely assigned.

An assignment which fulfills the above conditions for an edge to be considered as $b$-safe, is called a safe assignment.

For example, in Figure 3, $e_5$ and $e_6$ are both safe (their source nodes are primary inputs) and stop edges, $e_5$ and $e_6$ are suspect, $e_3$ is 0-safe (the corresponding safe assignment is $\text{value}(e_3) = X$ and $\text{value}(e_5) = 0$), and $e_1$ is a stop edge.

The motivation for defining $b$-safe edges is to minimize the number of conflicts during backtracing. By using a safe assignment we avoid possible conflicts, since assigning an $X$ value to an edge actually obviates the need for further backtracing through that edge. (If we assume that the edge is non-singular, we are assured that the desired value, 0 or 1, can be assigned to it.) Notice that a stop edge may also be a $b$-safe edge, in which case this characterization will help both in guiding the backtracing process (by postponing handling of the stop edge), and in the assignment of a value $b$ to the $b$-safe edge (knowing the possibility of a safe assignment). On the other hand, we must remember that a safe edge is not necessarily a stop edge.

In the following section we introduce RIDDLE, an algorithm with time complexity $O(\|E_y\|)$, which identifies some of the stop edges in a graph $\Gamma'(V_y,E_y)$. This algorithm is applied, as a pre-processing step, to the graph $\Gamma'$ corresponding to the block targeted for backtracing.
4. The RIDDLE Algorithm

We begin by presenting the following definition.

Definition 11: A node \( v \) is said to be **problematic with respect to an edge** \( l \), if for at least one of the edges in \( \text{inedges}_y(v) \) (say the edge \( j \)), the corresponding influence set \( \text{I}_y(l) \) contains edges, some of which are in the cone of the given edge \( \kappa(l) \), while others are not in it, and there is at least one assignment to the set \( \text{I}_y(l) \) which is not valid. (For the case of a graph representing a circuit at the gate level, where each \( v \in V_{bl} \) has a single output, this is equivalent to stating that there is a fanout node "above" \( l \) which has not reconverged inside \( \kappa(l) \). More formally, a node \( v \) is problematic with respect to edge \( l \), if

\[
(\exists f \in \text{inedges}_y(v)) (\exists e_1, e_2 \in \text{I}_y(f), e_1 \in \kappa(l) \text{ and } e_2 \not\in \kappa(l) \text{[3 an assignment which is not valid].}
\]

For example, the fanout node \( v \) in Figure 3 is problematic with respect to edge \( e_3 \), since \( e_4 \in \kappa(e_3) \) and \( e_2 \not\in \kappa(e_3) \).

We now present the *RIDDLE* algorithm, which traverses the edges in \( \Gamma(V, E) \), using a variant of topological sort [12], and qualifies them as either stop, b-safe or b-suspect edges (as defined above), or leaves them unqualified.

**THE RIDDLE ALGORITHM**

1. Initialize \( \forall v \in V, \text{count} (v) = \text{indegree}_y(v) \).
2. Initialize the queue \( Q := (V_p \cap V_y) \).
3. If \( Q = \phi \) then stop.
4. Pick \( v \), the first node (i.e., closest to the head) of \( Q \). (There is at least one such node, since \( \Gamma \) is acyclic.)
5. If \( \text{outdegree}_y(v) = 1 \) (say \( \text{outedges}_y(v) = \{e\} \)), then
   a. check if the edge \( e \) can be labeled as "b-safe" (for \( b = 0,1 \)) using Procedure B-SAFE below; and
   b. if there are no problematic nodes with respect to \( e \) (which can be determined by simply counting these nodes, and noting the edges they may be problematic for, as they are encountered by RIDDLE), then label \( e \) as a "stop" edge. (Notice that \( e \) can be both b-safe and a stop edge.)
6. If \( \text{outdegree}_y(v) > 1 \), then
   a. \( \forall l \in \text{inedges}_y(v) \), if there is an assignment \( \alpha \) to the set \( \text{I}_y(l) \) for which \( \theta^{\alpha^{-1}}_v \) is not defined (i.e., Function APV below returns a FALSE value), then label all edges \( e \in \text{I}_y(l) \) as "b-suspect," where \( b_e = \text{value}(e) \) in \( \alpha \); and
   b. \( \forall e \in \text{outedges}_y(v) \)
      1) \( \exists b \in \{0,1\} \) and \( e \) was not labeled as b-suspect before, check if the edge \( e \) can be labeled as "b-safe" using Procedure B-SAFE below; and
      2) if \( (\forall l \in \text{I}_y(l) \text{[3 a stop edge]) and } e \text{ is neither } 0 \text{-suspect nor } 1 \text{-suspect, then label } e \text{ as a "stop" edge.}
7. \( \forall e \in \text{outedges}_y(v) \)
   a. for node \( w = \text{drain}(e) \), \( \text{count}(w) := \text{count}(w) - 1 \); and
   b. if \( \text{count}(w) = 0 \), then append \( w \) to the tail of \( Q \).
8. \( Q := Q - v \), and resume at step (3).

Definition 12: Let us denote by \( \bar{\theta} \), the complement of the transfer function \( \theta \), for node \( v \). We define the primitive cube (pc) set \([3] \) of \( \bar{\theta}_v \), denoted by \( \pi(\bar{\theta}_v) \), to contain all the prime implicants of the function \( \bar{\theta}_v \). (These pc-s are obtained using a straightforward logic minimization technique [13].)

Using the notation \( \text{outedges}_y(v) = \{e_1, e_2, \ldots, e_n\} \), where \( n = \text{outdegree}(v) \), we represent each pc as the vector \( pc = (b_1, b_2, \ldots, b_n) \) (or \( pc = (b) \) for short) with \( b_i = \text{value}(e_i) \).

The parameters to APV are the node \( v \) and an edge \( l \in \text{inedges}_y(v) \).
Function APV (All Possible Values)

1. Build the modified primitive cube set \( \pi'(\bar{\theta}) = \{pc'\} \) by defining \((\forall pc \in \pi(\bar{\theta}))[pc' = (b')]\), where

\[
b'_i = \begin{cases} 
  b_i & \text{if } e_i \notin I_i(0); \\
  X_i & \text{otherwise}.
\end{cases}
\]

2. If \( \exists pc' \in \pi'(\bar{\theta}) \exists \forall [1 \leq i \leq n][b'_i = X] \), then return(FALSE) (there is at least one assignment of values on the set \( I_i(0) \) which is not valid, see Claim in Appendix A). Otherwise, return(TRUE).

The example below illustrates the application of Function APV.

Example 2: Let us assume the node \( v \) is a 4-input priority encoder. Its transfer function \( \theta_1 = \{000,001,011,101,111\} \) and \( \theta_2 = \{010,100,110\} \). (Notice that both \( \theta_1 \) and \( \theta_2 \) are constant throughout the backtracing process.) The corresponding primitive cube set is then given by \( n(\theta) = \{(X,1,0),(1,X,0)\} \).

Suppose we are interested in evaluating Function APV for an edge \( I \) for which \( I_i(0) \) contains the two edges matching the two leftmost bits in each \( pc \in \pi(\bar{\theta}) \). In this case, \( \pi'(\bar{\theta}) = \{(X,X,0),(X,X,0)\} \), and thus we see that Function APV returns TRUE, i.e., all combination of values on the edges in the set \( I_i(0) \) are valid assignments. On the other hand, if we look at the two rightmost bits, \( \pi'(\bar{\theta}) = \{(X,X,X),(1,X,X)\} \), and since we have \( (3 pc' \in \pi'(\bar{\theta}))[pc' = (X,X,X)] \), Function APV returns the value FALSE.

Definition 13: For every edge \( e \in E_y \) we define \( V'_b(V'_b) \) to be the set of all input vectors for which \( value(e) = 0 \) \( (1) \). We denote by \( n(V'_b) \) the set of primitive cubes corresponding to all the prime implicants of the set \( V'_b \), for \( b \in \{0,1\} \).

Procedure B-SAFE (parameter: the edge \( e \))

1. If \( source(e) \in V_{in} \), then edge \( e \) is safe (refer to Definition 10).
2. For \( b = 0 \) \( (1) \) determine the corresponding safe assignment on the set \( inedges(v) \) \( (v = source(e)) \). Represent this assignment by a safe cube, \( sc = (s_1, s_2, \ldots, s_{\text{indegree}}) \), where each \( s_i \) corresponds to the value of an edge \( e_i \in inedges(v) \), and is given by (with \( \lambda \in \{0,1,X\} \) and \( b \in \{0,1\} \)):

\[
s_i = \begin{cases} 
  \lambda & \text{if } e_i \text{ is a } b\text{-safe edge and } e_i \text{ is not a } \bar{b}\text{-safe edge; } \\
  X & \text{if } e_i \text{ is neither a safe edge nor a stop edge; } \\
  \lambda & \text{otherwise.}
\end{cases}
\]

3. The edge \( e \) is b-safe if \( (\exists pc \in \pi(V'_b))[\forall i \in \text{indegree}[\lambda]](s_i \leq b) \), with the relation \( \leq \) defined by the set of ordered pairs \( \{(0,0),(0,X),(1,1),(1,X),(X,X),(\lambda,0),(\lambda,1),(\lambda,X)\} \). I.e., if there is at least one \( pc \in \pi(V'_b) \) which "covers" (according to the relation \( \leq \)) the safe assignment, then edge \( e \) is b-safe (refer to Definition 10).

Lemma 1: All the edges \( e \in E_y \) which are labeled as stop edges by RIDDLE, are also stop edges according to Definition 8. (For the proof, see Appendix B.)

5. Experimental Results

We have characterized certain edges in a high-level design, which are conducive to the acceleration of the backtracing phase in a test generation algorithm. Also, we have shown RIDDLE, an algorithm with time complexity \( O(|E_x|) \), for identifying some of the edges having the above characteristics. Since a gate level
description is simply a special case of a high level design, our results apply in this case as well. This fact is of particular interest, since it allows us to evaluate the effectiveness of our edge-labeling process, using a popular benchmark of gate level designs, namely those in [10].

The procedure we utilized to check the performance of RIDDLE consists of two main phases. First, we perform fault simulation on the designs with (pseudo-) random stimuli, using the fault simulator HSS [14], until the coverage curve flattens (the last group of 32 input patterns did not detect any additional fault). Table 3 shows the fault coverage and the number of faults which remain undetected after the first phase. The second phase consists of running RIDDLE, followed by the D-Algorithm for deterministic test pattern generation on the faults undetected in the first phase. We modified the D-Algorithm to defer justification of values on stop edges, and biased its selection of cubes towards safe-assignments.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Number of Faults</th>
<th>Coverage Random(%)</th>
<th>Number of Patterns</th>
<th>Faults Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>524</td>
<td>97.32</td>
<td>352</td>
<td>14</td>
</tr>
<tr>
<td>C499</td>
<td>758</td>
<td>98.02</td>
<td>320</td>
<td>15</td>
</tr>
<tr>
<td>C880</td>
<td>942</td>
<td>94.90</td>
<td>384</td>
<td>48</td>
</tr>
<tr>
<td>C1355</td>
<td>1574</td>
<td>93.96</td>
<td>320</td>
<td>95</td>
</tr>
<tr>
<td>C1908</td>
<td>1879</td>
<td>94.46</td>
<td>800</td>
<td>104</td>
</tr>
<tr>
<td>C2670</td>
<td>2747</td>
<td>83.14</td>
<td>640</td>
<td>463</td>
</tr>
<tr>
<td>C3540</td>
<td>3428</td>
<td>93.93</td>
<td>736</td>
<td>208</td>
</tr>
<tr>
<td>C5315</td>
<td>5350</td>
<td>98.26</td>
<td>640</td>
<td>93</td>
</tr>
<tr>
<td>C6288</td>
<td>7744</td>
<td>99.50</td>
<td>128</td>
<td>38</td>
</tr>
<tr>
<td>C7552</td>
<td>7550</td>
<td>91.40</td>
<td>448</td>
<td>649</td>
</tr>
</tbody>
</table>

Our first question is whether RIDDLE improves the performance of the D-Algorithm. In order to answer this question we took those ("random resistant") faults which remained undetected at the end of the first phase, and run the D-Algorithm with and without using RIDDLE. (For this set of experiments, RIDDLE was applied once to the whole circuit, rather than on a block-by-block basis. This approach provides a lower bound on the potential improvements offered by its application.) Results, given as fault coverage percentages (over all faults in the design), are shown in Table 4, where the number of backtracks has been limited to 20. This was done to maintain compatibility with the results given in [15] (which are also listed in Table 4), for the D-Algorithm applied to all faults in the designs.

Another aspect of the D-Algorithm's performance is its running time, which we are only capable of comparing for our two versions (with and without using RIDDLE). This comparison is shown in Table 5, with the number of backtracks limited to 10, so as not to unduly bias the results because of a few faults for which this limit is exceeded. We can clearly see that the D-Algorithm which uses the edge labels produced by RIDDLE has equal or better performance, both in the average number of backtracks, its increment of the overall fault coverage, and time efficiency, than its counterpart which ignores these labels.
Table 4 - Fault Coverage (%) by D-Algorithm with and without RIDDLE

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>97.4</td>
<td>98.85</td>
<td>99.23</td>
</tr>
<tr>
<td>C499</td>
<td>68.5</td>
<td>98.02</td>
<td>98.94</td>
</tr>
<tr>
<td>C880</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>C1355</td>
<td>58.2</td>
<td>93.96</td>
<td>93.96</td>
</tr>
<tr>
<td>C1908</td>
<td>95.0'</td>
<td>97.33</td>
<td>99.30</td>
</tr>
<tr>
<td>C2670</td>
<td>95.3</td>
<td>95.23</td>
<td>95.41</td>
</tr>
<tr>
<td>C3540</td>
<td>94.4</td>
<td>95.97</td>
<td>95.97</td>
</tr>
<tr>
<td>C5315</td>
<td>98.5</td>
<td>98.86</td>
<td>98.89</td>
</tr>
<tr>
<td>C6288</td>
<td>99.1</td>
<td>99.50</td>
<td>99.50</td>
</tr>
<tr>
<td>C7552</td>
<td>96.3</td>
<td>97.69</td>
<td>97.88</td>
</tr>
</tbody>
</table>

Table 5 - Performance Advantages Using RIDDLE

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Backtracks(Avg) without</th>
<th>Coverage(%) without</th>
<th>Coverage(%) with</th>
<th>Time Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>98.85</td>
<td>99.23</td>
<td>1.50</td>
</tr>
<tr>
<td>C432</td>
<td>4.2</td>
<td>98.02</td>
<td>98.94</td>
<td>1.39</td>
</tr>
<tr>
<td>C499</td>
<td>4.7</td>
<td>100</td>
<td>100</td>
<td>1.06</td>
</tr>
<tr>
<td>C880</td>
<td>0.6</td>
<td>93.96</td>
<td>93.96</td>
<td>1.00</td>
</tr>
<tr>
<td>C1355</td>
<td>9.1</td>
<td>97.01</td>
<td>99.25</td>
<td>1.90</td>
</tr>
<tr>
<td>C1908</td>
<td>5.2</td>
<td>95.23</td>
<td>95.41</td>
<td>1.44</td>
</tr>
<tr>
<td>C2670</td>
<td>1.1</td>
<td>95.97</td>
<td>95.97</td>
<td>1.12</td>
</tr>
<tr>
<td>C3540</td>
<td>0.4</td>
<td>98.72</td>
<td>98.89</td>
<td>1.52</td>
</tr>
<tr>
<td>C5315</td>
<td>1.2</td>
<td>99.50</td>
<td>99.50</td>
<td>1.16</td>
</tr>
<tr>
<td>C6288</td>
<td>1.5</td>
<td>97.69</td>
<td>97.80</td>
<td>1.17</td>
</tr>
<tr>
<td>C7552</td>
<td>3.6</td>
<td>97.69</td>
<td>97.80</td>
<td></td>
</tr>
</tbody>
</table>

Having determined that the D-Algorithm benefits from RIDDLE, we ask ourselves how this combination compares to other test generation algorithms. Table 6 addresses this question with respect to the results published in [16] for PQDEM and FAN. Again, the backtrack limits are set to 10 and 1000 for compatibility with the above publication. (The results for C499 are not given, since apparently there is a mismatch in the number of faults with that used in [16]. We attained the maximum achievable fault coverage (because of 8 redundant faults) of 98.94% with a limit of 10 backtracks.) Table 7 shows the number of faults which were found to be redundant (there is no test for them), and the number of those for which 1000 backtracks was exceeded ("aborted").
Table 6 - Fault Coverage Comparison (D + R = D-Algorithm with RIDDLE)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>PODEM 10 BACKTRACKS</th>
<th>FAN 10 BACKTRACKS</th>
<th>D+R 10 BACKTRACKS</th>
<th>PODEM 1000 BACKTRACKS</th>
<th>FAN 1000 BACKTRACKS</th>
<th>D+R 1000 BACKTRACKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>91.5</td>
<td>93.7</td>
<td>99.23</td>
<td>91.8</td>
<td>93.7</td>
<td>99.23</td>
</tr>
<tr>
<td>C880</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>C1355</td>
<td>99.5</td>
<td>97.5</td>
<td>93.96</td>
<td>99.5</td>
<td>99.5</td>
<td>93.96</td>
</tr>
<tr>
<td>C1908</td>
<td>99.5</td>
<td>99.3</td>
<td>99.25</td>
<td>99.5</td>
<td>99.5</td>
<td>99.52</td>
</tr>
<tr>
<td>C2670</td>
<td>94.6</td>
<td>95.7</td>
<td>95.41</td>
<td>95.4</td>
<td>95.7</td>
<td>95.74</td>
</tr>
<tr>
<td>C3540</td>
<td>95.5</td>
<td>95.8</td>
<td>95.97</td>
<td>95.5</td>
<td>96.0</td>
<td>96.00</td>
</tr>
<tr>
<td>C5315</td>
<td>98.3</td>
<td>98.9</td>
<td>98.89</td>
<td>98.8</td>
<td>98.9</td>
<td>98.89</td>
</tr>
<tr>
<td>C6288</td>
<td>99.5</td>
<td>99.4</td>
<td>99.50</td>
<td>99.5</td>
<td>99.5</td>
<td>99.56</td>
</tr>
<tr>
<td>C7552</td>
<td>96.8</td>
<td>98.2</td>
<td>97.80</td>
<td>97.8</td>
<td>98.2</td>
<td>98.04</td>
</tr>
</tbody>
</table>

Table 7 - Summary of Undetected Faults

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Redundant Faults</th>
<th>Aborted Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>C432</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>C499</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>C880</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C1355</td>
<td>8</td>
<td>87</td>
</tr>
<tr>
<td>C1908</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>C2670</td>
<td>106</td>
<td>11</td>
</tr>
<tr>
<td>C3540</td>
<td>137</td>
<td>0</td>
</tr>
<tr>
<td>C5315</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>C6288</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>C7552</td>
<td>129</td>
<td>19</td>
</tr>
</tbody>
</table>

Again, we see that the D-Algorithm with RIDDLE performs as well, or better than both PODEM and FAN, except for the case of design C1355. This design is functionally equivalent to C499, with XORs expanded into NANDs. The D-Algorithm is known to have inherent difficulties with this type of circuit (refer to Table 4), and in this case RIDDLE cannot help much. Furthermore, we believe that the high performance of FAN, can be greatly improved by incorporating in it the information supplied by RIDDLE.

We should recall at this point, that the concepts introduced in this paper apply to designs expressed as a structural interconnection of functional blocks, of which a gate level description is only a special case. Thus, we regard the importance of the concepts in RIDDLE to transcend the gate level in their generality.
Appendix A

In this Appendix we prove the correctness of Function APV.

Claim 1: Function APV, as defined in the paper, returns the value $\text{FALSE}$ if there is at least one assignment of values on the set $I_h(l)$, where $\text{drain}(l) = v$, which is not valid.

Proof:

$\Rightarrow$: Say there is an assignment of values on the set $I_h(l)$ which is not valid. Let us call this assignment $\alpha$. Clearly, there is no assignment to the set $\text{outedges}(v)$ which belongs to $\theta_v$ and agrees with $\alpha$ on the edges in $I_h(l)$. (An assignment $\alpha$ on a set A is said to agree with an assignment $\beta$ on a set B, or vice-versa, if both $\alpha$ and $\beta$ have the same value on the edges in $A \cap B$.) Therefore, all possible assignments on $\text{outedges}(v)$ which agree with $\alpha$ belong in the set $\pi(\theta_v)$ and thus the set $\pi(\theta_v)$ will contain a pc with value $X$ for all edges $e \in \text{outedges}(v) - I_h(l)$. This means that after applying Function APV - step (1), we will have in $\pi'(\theta_v)$ a $\text{pc}' := (X)$.

$\Leftarrow$: Let us assume that there is a $\text{pc} \in \pi(\theta_v)$ for which all the edges in $\text{outedges}(v) - I_h(l)$ have the value $X$. Furthermore, the values corresponding to the edges in the set $I_h(l)$ for this $\text{pc}$ agree with an assignment, say $\beta$. This $\text{pc}$ clearly represents all possible assignments on the set $\text{outedges}(v)$ which agree with $\beta$, and thus all these assignments belong in $\theta_v$. Therefore, there are no assignments in $\theta_v$ which agree with $\beta$.

Appendix B

In this Appendix we show that all edges which were labelled as "stop" edges by RIDDLE are indeed so, according to Definition 8.

Definition 14: Assume we have edges $h, e_1, e_2 \in E$, where $e_1 \in \nu(h), e_2 \notin \nu(h)$, and $\text{source}(e_1) = \text{source}(e_2) = v$. Node $v$ is called a split node with respect to $h$, if $(\exists \ell \in \text{inedges}(v))[e_1, e_2 \in I_h(l)]$. (In words, some subset of edges in $I_h(l)$ are in the cone of $h$, while a different subset of edges are outside this cone.) We refer to the edges $e \in I_h(l)$, where $v = \text{drain}(l)$ is a split node, as split edges with respect to $h$, induced by $l$.

Claim 2: For any directed path $p = e_1, e_2, \ldots , e_n$ in $\Gamma$, where $e_i$ is labeled as a stop edge (by RIDDLE) and $(\forall i, 1 \leq i < n)[e_{i-1} \rightarrow e_i \in \Gamma], \text{if there is an edge labeled as a stop edge in } p \text{, then for the one closest to } e_i, \text{ say } e_i, \text{ outdegree(source}(e_i)) = 1$. (This means that this edge was labeled by RIDDLE - step (6.b).)

Proof: By way of contradiction. Assume the edge $e_i$, as defined in the claim, was labeled as a stop edge in RIDDLE - step (6.b.2). This implies, that $(\forall e \in \text{inedges(source}(e_i))$ and $e \rightarrow e_i[e$ is labeled as a stop edge$]$. Since $e_{i-1}$ is one such edge, it should have also been labeled as a stop edge, and since it is closer to $e_i$, this is a contradiction.

Claim 3: Every edge $l$ which induces an edge $e$ to be a split edge with respect to some edge $h$ which is labeled as a stop edge (by RIDDLE), is also labeled as a stop edge.
Proof: Clearly, if \( e \) is labeled as a stop edge, \((\forall k \in I_e(k))[k \text{ is labeled as a stop edge}]\) (from RIDDLE - step (6.b.2)), and thus the Claim is trivially true. Therefore, in the following we handle the case where \( e \) is not labeled as a stop edge.

Let us define the set \( J = \{ j | \exists j, e \in I_j(l) \text{ and } j \neq \kappa(l) \} \). (This set contains all edges \( j \neq \kappa(h) \) which share with \( e \) at least one edge \( l \equiv [e \rightarrow j \text{ and } j \rightarrow l] \).) Now we have two possible cases, namely either \((\exists j \in J)[j \text{ is not labeled as a stop edge}]\), or \((\forall j \in J)[j \text{ is labeled as a stop edge}]\).

Case 1 \((\exists j \text{ not labeled as a stop edge})\): Clearly, there is a directed path \( p \) from \( v = \text{source}(e) \) to \( u = \text{drain}(h) \), and from Claim 2, there must be an edge \( k \in p \) which was labeled as a stop edge by RIDDLE - step (5.b). Also, since \( j \) is not labeled as a stop edge, by Definition 11 \( v \) is a problematic node for all edges in the path \( p \), which is a contradiction to the fact that edge \( k \) is labeled as a stop edge by RIDDLE - step (5.b).

Case 2 \((\forall j \text{ are labeled as stop edges})\): Since \( v \) is a split node, \( \text{outdegree}(v) > 1 \) and therefore we must have used RIDDLE - step (6.b.2), thus \((\forall j)(\forall k \in I_j(l))[k \text{ is labeled as a stop edge}]\), which again proves the Claim.

In the following we present a constructive process for making a valid assignment to the edges in \( \kappa(e) \), given that the edge \( e \) is labeled as a stop edge by RIDDLE, and that there is a valid assignment to the edges in \( E_e - \kappa(e) \). Notice that this process, called VALID, does not find a specific valid assignment, rather it just shows that such an assignment exists. We then use this process, after proving its correctness, to show that the edges labeled as stop edges by RIDDLE indeed fulfill the conditions for being such, according to Definition 8.

**VALID**

1. Initialize the set \( E_1 = \emptyset \) and the set \( E_2 := \kappa(e) \).
2. If \( E_2 = \emptyset \) then stop.
3. Select an edge \( l \in E_2 \equiv \{ j \in \kappa(h) \} \text{ and } l \text{ is labeled as a stop edge} \).
4. If \( l \text{ was labeled as a stop edge by RIDDLE - step (5.b)} \), we define \( \kappa'(l) = \kappa(l) - \{ f | f \in \kappa(l) \text{ and } f \neq \kappa(j) \} \), where \( j \in E_2 - \kappa(l) \). Then, \( E_1 := E_1 \cup \kappa'(l) \) and \( E_2 := E_2 - \kappa(l) \), and resume at step (2).
5. If \( l \text{ was labeled as a stop edge by RIDDLE - step (6.b.2)} \), then \( E_1 := E_1 \cup \{ l \} \) and \( E_2 := E_2 - \{ l \} \), and resume at step (2).

Claim 4: There is always at least one edge which can be selected in VALID - step (3), i.e., \((\exists l \in E_2 \equiv \{ j \in \kappa(h) \} \text{ and } l \text{ is labeled as a stop edge})\).

**Proof:** First, since the graph \( \Gamma \) is acyclic, and the edges in \( E_2 \) are a subset of \( E_e \), initially there is at least one edge in \( E_2 (\neq \emptyset) \) which does not influence any other edge in \( E_2 \). Deletion of edges from \( E_2 \), as performed by VALID, leaves the graph still acyclic, and thus there is always an edge which fulfills this condition. Now, we shall show (by induction) that the edges which are picked in VALID - step (3) (which we already know do not influence other edges in \( E_2 \)) are always labeled as stop edges.

The edge \( e \), which is the first edge selected in VALID - step (3), clearly is such an edge (by the initial conditions to the process VALID), and thus we have the basis for the induction. Clearly, an edge \( j \) belongs to \( E_2 \) at least as long as there is at least one edge \( k \in E_2 \equiv j \in I_j(k) \). These edges \( k \) (which may or may not be stop edges) will be moved to \( E_1 \) either in VALID - step (4) or in VALID - step (5). Furthermore, such an edge \( j \) can be selected in VALID - step (3), only if it remained in \( E_2 \) in spite of having \((\forall k \in I_j(k))[k \in E_1] \).

The induction step now assumes that the stop edge \( l \) is picked in VALID - step (3). Let us first look at the case where \( l \) is moved to \( E_1 \) by VALID - step (4), together with all the edges in \( \kappa(l) \) which do not influence...
other edges in \( E_2 \). All the edges \( e_i \in \kappa(l) \) remaining in \( E_2 \) and influence some edge \( e_n \in \kappa(l) \) (i.e., \( e_i \rightarrow e_n \)) which was moved to \( E_1 \) (we know that the edge \( e_n \) in this case is actually a split edge with respect to \( l \) (which is a stop edge by the induction step), induced by \( e_i \), by Claim 3 are labeled as stop edges.

For the case where the stop edge \( l \) is moved to \( E_1 \) by VALID - step (5), we have that all the edges \( e, e_i \in \kappa(l) \)

remaining in \( E_2 \) and influence edge \( l \), are labeled as stop edges according to RIDDLE - step (6.b.2).

**Claim 5:** All edges \( h \in E_1 \) can be given a valid assignment.

**Proof:** By induction.

Initially, all the edges in \( E_1 = E_y - \kappa(e) \) can be given valid assignments. Furthermore, and also because of the initial conditions at the beginning of VALID, the edge \( e \) can be given a valid assignment.

Let us assume that after \( n \) iterations of VALID, valid assignments have been made to all the edges in \( E_1 \). Furthermore, let us say that edge \( l \), which is labeled as a stop edge, is selected in the current iteration of VALID (refer to Claim 4). Clearly, \((\forall k \in E_2 \Rightarrow k \sim f)(k \in E_2)\), and by the induction step, they all have valid assignments (i.e., the inverse transfer relation exists), and thus there is an assignment to \( l \). Also, since \( l \) is a stop edge, it belongs only to influence sets for which all assignments are valid (i.e., Function APV returns TRUE, when invoked from RIDDLE). Therefore, it is trivial to see that the assignment imposed on \( l \) by the values assigned to edges already in \( E_1 \), is a valid assignment.

There are two possibilities for processing the edge \( l \), namely using VALID - step (4) or VALID - step (5). If \( l \) is processed by VALID - step (4), we have that \( \text{outdegree}(\text{source}(l)) = 1 \). Assuming that there is no redundancy in \( \kappa(l) \), there is at least one valid assignment to all the edges in \( \kappa(l) \) for each assignment (0 or 1) to \( l \). The valid assignment for the edges in \( \kappa'(l) \) is clearly achievable, because of the non-redundancy assumption, and if \( \kappa(l) = \kappa'(l) \) the Claim if proven for this case.

Let us now assume that \( \kappa(l) \neq \kappa'(l) \), and examine the edges in the set \( F = \kappa(l) - \kappa'(l) = \{ f | f \in \kappa(l) \text{ and } f \in \kappa'(l) \} \), where \( j \in E_2 - \kappa(l) \). It is clear that \((\exists k \in E_2 - \kappa(l)) [k \sim f \text{ and } f \in F] \). We now show that there is no possible conflict between the assignment to \( \forall e_n \in \kappa'(l) \Rightarrow e_n \sim f \) and any assignment to be made to \( k \). Since \( l \) is being processed by VALID - step (4), there are no problematic nodes with respect to \( l \) in \( \kappa(l) \), and thus (by Definition 11) all possible assignments to the edges in \( L(f) \) are valid assignments.

The second case assumes that \( l \) is processed by VALID - step (4). Since \( l \) is the only edge being moved to \( E_1 \), and we have already shown that \( l \) can be given a valid assignment, the proof is complete.

**Claim 6:** VALID as described above terminates.

**Proof:** Since \( E_2 \) is initialized with a finite number of edges, and during each iteration we take out edges from \( E_2 \), and no edges are added to it, after a finite number of iterations \( E_2 = \emptyset \) and VALID terminates.

**Lemma 1:** All the edges \( e \in E_y \) which are labeled as stop edges by RIDDLE, are also stop edges according to Definition 8.

**Proof:** By construction.

We have shown that process VALID, which starts with any given valid assignment on the set \( E_y - \kappa(e) \), where \( e \) is labeled as a stop edge, terminates (Claim 6) in such a way that \( E_1 = E_y \). Furthermore, we proved that the set \( E_y \) can be given a valid assignment (Claim 5).
Acknowledgements

Many thanks to Israel Berger for the helpful discussions throughout the course of this research. Also, we would like to thank Danit Ben-Ari for suggesting improvements on a preliminary version of the manuscript.

References