STATIC TIMING ANALYSIS OF
MICROPROCESSORS WITH EMPHASIS
ON HEURISTICS

by

SIVASUBRAMANIAM KRISHNAMURTHY

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis Adviser: Dr. Daniel G. Saab

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

May, 2008
We hereby approve the thesis/dissertation of

SIVASUBRAMANIAM KRISHNAMURTHY

candidate for the MASTERS degree *

(signed) DANIEL G. SAAB (chair of the committee)

CHRISTOS A. PAPACHRISTOU

SWARUP BHUNIA

(date) 01/14/2008

*We also certify that written approval has been obtained for any proprietary material contained therein.
ACKNOWLEDGEMENTS

I owe my loving thanks and gratitude to my parents, Mr. Krishnamurthy Lakshminarayanan and Mrs. Janaki Krishnamurthy who have been supportive throughout my M.S program. I would like to express my gratitude to my sister and family for their loving support.

I would like to express my deep and sincere gratitude to my advisor Dr. Daniel G. Saab, Ph.D, for his extended support and guidance throughout my coursework and thesis. His wide knowledge and his logical way of thinking have been of great value for me. His encouragement and personal guidance provided a good basis for this study.

I warmly thank Dr. Swarup Bhunia, Ph.D, for the constructive criticism and support in the duration of this thesis. I also had the pleasure of using this work for the ISQED '07 conference paper titled “Adaptation to Temperature-Induced Delay Variations in Logic Circuits Using Low-Overhead Online Delay Calibration.” and ICCAD '07 conference paper titled “Low-Overhead Design Technique for Calibration of Maximum Frequency at Multiple Operating Points”.

During this study I have collaborated with many colleagues at the VLSI group, Electrical Engineering and Computer Science department of Case Western Reserve University, to whom I’m very grateful.

Finally, I would like to thank Case Western Reserve University and Mentor Graphics Inc. for providing the hardware and software tools to help complete this work.
# Contents

1 Introduction ................................................................. 1  
1.1 Preliminary ................................................................. 2  
1.2 Types of timing analysis ............................................... 7  
1.3 Common Timing Analysis Techniques ............................... 8  
1.4 Automatic Test Pattern Generation ................................. 10

2 Survey of Static Timing Analysis Algorithms ....................... 12  
2.1 False Paths and Path Sensitization ................................. 13  
2.2 Symbolic Simulation and Statistical Timing Analysis .......... 15

3 ATPG Techniques ............................................................. 18  
3.1 D-Algorithm ............................................................... 19  
3.2 PODEM-X ................................................................. 27  
3.3 FAN Algorithm ........................................................... 28  
3.4 Pseudorandom test generation ...................................... 28  
3.5 VIPER ...................................................................... 29

4 Algorithm ................................................................. 35  
4.1 Circuit Levelization ..................................................... 35  
4.2 Preprocessing ............................................................ 36  
4.2.1 Partitioning ........................................................ 37  
4.3 Path Sensitization ....................................................... 39  
4.4 Algorithm ............................................................... 39

5 Results ....................................................................... 44  
5.1 Conclusion ................................................................. 47
List of Tables

5.1 ISCAS ’85 Results without Heuristics . . . . . . . . . . . . . . 45
5.2 ISCAS ’85 Results with Heuristics . . . . . . . . . . . . . . . 45
5.3 ISCAS ’89 Results without Heuristics . . . . . . . . . . . . . . 46
5.4 ISCAS ’89 Results with Heuristics . . . . . . . . . . . . . . . . 46
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Sequential Circuit and ILA Model</td>
<td>2</td>
</tr>
<tr>
<td>1-2</td>
<td>Sequential Circuit at Time Frame $T_i$</td>
<td>4</td>
</tr>
<tr>
<td>1-3</td>
<td>Circuit Paths</td>
<td>5</td>
</tr>
<tr>
<td>3-4</td>
<td>Primary D-cube and singular cover of a two input NAND gate</td>
<td>20</td>
</tr>
<tr>
<td>3-5</td>
<td>Example circuit for D-Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>3-6</td>
<td>The phases of simulation of D-Algorithm</td>
<td>22</td>
</tr>
<tr>
<td>3-7</td>
<td>(Continued) The phases of simulation of D-Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>3-8</td>
<td>(Continued) The phases of simulation of D-Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>4-9</td>
<td>Example Bench format</td>
<td>36</td>
</tr>
<tr>
<td>4-10</td>
<td>Partitions in an example circuit</td>
<td>38</td>
</tr>
<tr>
<td>4-11</td>
<td>STA pseudo code</td>
<td>40</td>
</tr>
<tr>
<td>4-12</td>
<td>Initial Assignment and First Implication</td>
<td>41</td>
</tr>
<tr>
<td>4-13</td>
<td>The phases of simulation</td>
<td>42</td>
</tr>
</tbody>
</table>
Static Timing Analysis of Microprocessors with Emphasis on Heuristics

Abstract

by

SIVASUBRAMANIAM KRISHNAMURTHY

As designers build complex digital circuits with ever diminishing device sizes, there is a need to obtain fast circuits with low hardware overhead. Critical path is the longest sensitizable path in a digital logic circuit which determines the operating frequency of the circuit. Static timing analyzers enumerate critical paths in a circuit and determine the optimal operating frequency. This work presents a static timing analyzer that makes use of an ATPG technique based on the PODEM engine. To improve the efficiency of the technique, a partitioning scheme is used in the path sensitization subroutine to reduce the search space. The results from this implementation for the ISCAS ’85 and ’89 benchmarks demonstrates performance improvements.
Chapter 1

Introduction

Static Timing Analysis is a technique to determine the expected timing of a digital circuit without the need for rigorous circuit simulation. Traditionally high-performance circuits have been characterized by their operating clock frequency. Determining the clock frequency at which a circuit operates requires identifying the slowest path at various steps of the design process. The slowest path can be identified by performing exhaustive simulation on the circuit. The exhaustive simulation is also referred to as dynamic simulation. This process is time intensive and highly error prone. Contrary to exhaustive circuit simulation, static timing analysis facilitates in a fast and reasonably accurate estimate of the circuit timing. This is due to the fact that the during preprocessing the circuit is flattened and the delay models are simplified. Also, the interactions between the circuit elements are limited in logic.
1.1 Preliminary

In a synchronous digital circuit, data travels one circuit stage during each clock cycle. A stage is a combinational circuit bounded by flip-flops/latches. An example sequential circuit composed of combinational logic stage and flip-flops is shown in Figure 1-1(a). To emulate the clock cycles, an Iterative logic array (ILA) model[15] is used to represent a sequential circuit by unrolling the circuit into a set of time frames/stages. To construct this model, the flip-flops are removed and represented as the interconnection between time frames/stages as shown in Figure 1-1(b). Note that the combinational logic are duplicated in each time frame as shown in Figure 1-1(b) where time frame $T_i$ represents the state of the circuit in the $i$th clock cycle.

![Sequential Circuit and ILA Model](image)

Figure 1-1: Sequential Circuit and ILA Model

Each $T_i$ has two input vectors $S_i$ and $PI_i$, and two output vectors $S_{i+1}$
and $PO_i$. $PI_i$ is the set of primary inputs (PIs) and $PO_i$ is the set of primary outputs (POs) in Figure 1-1(a). $S_i$ is the set of present state and $S_{i+1}$ is the set of next state to frame $T_i$. $S_{i+1}$ is the present state input to time frame $T_{i+1}$.

Figure 1-2 represents the time frame $T_i$ of a sequential circuit. The input vector $SI$ is the set of present states for time frame $T_i$ stored in the flip-flops $Q_i$'s, also referred to as the pseudo-input. The output vector $SO$ is the set of present states for time frame $T_{i+1}$ stored in the flip-flops $D_i$'s, also referred to as pseudo-output. The time before the clock edge that the signal at the $D_i$'s has to be stable is called the setup time ($T_s$) and the time after the clock edge that the $D_i$ input has to be stable is called the hold time. In the figure 1-2, the dashed line between pseudo-input (SI) and the pseudo-output (SO) represents a circuit path.

The combinational circuit in each time frame $T_i$ is represented by an acyclic directed graph $G := (V, E)$. The gates are represented by the set of nodes $V$, and each gate input is represented by the an edge $E$, where the source of the edge is a gate output and the destination is a gate input. The delays associated with a gate $M$ is represented by a weight on the corresponding node $g$ in $V$. In this representation we would like to define the following:

**Definition 1**: A path $P$ in $G$ is a set of nodes $\{g_1, g_2, \ldots, g_n\}$, where $<g_i, g_{i+1}>$ is an edge in $E$. $g_1$ is an input/pseudo-input and $g_n$ is an
Figure 1-2: **Sequential Circuit at Time Frame** $T_i$

output/pseudo-output.

*Example:* Figure 1-3 shows a combinational circuit. $\{I_0, G_0, G_4, G_5\}$ is a path. Node $I_0$ is an input, node $G_5$ is an output, and each pair of adjacent nodes are edges. The set of all path from input to output are referred to as *structural paths.* Each gate $g_i$ along the path has an input from a gate $g_{i-1}$ that is on the path which is called on-path input. All other inputs of that gate are called off-path inputs. For this path, the on-path inputs are $\{I_0, G_0, G_4, G_5\}$ and the off-path inputs are $\{I_1, G_1\}$

*Definition 2:* A path $P$ in $G$ is a *sensitizable path,* if there exists a pair of inputs for which a transition can be launched at the input of the path, traverse every node along the path, and is observed at the output of the path.

*Example:* Path $\{I_2, G_1, G_4, G_5\}$ is sensitizable with the combination of
primary inputs \((I_0, I_1, I_2, I_3) = (11, P_t, X)\) and \((I_0, I_1, I_2, I_3) = (1, 1, N_t, X)\), where \(P_t\) represents a positive transition (logic 0 to logic 1), \(N_t\) represents a positive transition (logic 1 to logic 0), and \(X\) is undefined (logic 0 or logic 1). The pair of inputs forces all the off-path inputs to non-controlling values. A non-controlling value is a logic value which does not determine the output value of a gate independently. For the path \(\{I_2, G_1, G_4, G_5\}\), the off-path inputs for gate \(G_1\) is \(I_1\), gate \(G_4\) is \(G_0\) and the on-path inputs for gate \(G_1\) is \(I_2\), gate \(G_4\) is \(G_1\), and gate \(G_5\) is \(G_4\) respectively.

![Circuit Paths](image)

**Figure 1-3: Circuit Paths**

**Definition 3**: A path \(P\) is false, if there exists no pair of inputs for which a transition can be launched at the input of the path, traverse every node along the path, and is observed at the output of the path.

**Example**: Path \(\{I_2, G_1, G_4, G_6\}\) is a false path. It can be shown that
there exists no input combination which will set all the off-path inputs to non-controlling values.

Definition 4: A path that can be sensitized during the normal operation of the circuit, using input combinations that are permissible under normal circuit operation are called *functional sensitizable paths*. Note, these paths are derived from inputs that are based on circuit functions or instructions.

Definition 5: *Path delay* of a synthesizable path is the sum of all the weights on the nodes along that path. This is expressed as,

$$\text{Path Delay } D(P) = \sum_{i=1}^{p} D(g_i)$$

where $g_i$ represents the gates along the path and $D(g_i)$ represents the propagation delay of gate $g_i$.

Example: Consider a path $\{I_2, G_1, G_4, G_5\}$, in figure 1-3

Path Delay = Sum of gate delays of $G_1$, $G_4$, and $G_5$

$$= 5 + 9 + 1$$

$$= 15$$

These delays are derived from circuit model to represent the propagation delay from the gate inputs to the gate outputs.

Definition 6: A *critical path* is a sensitizable path with the largest path delay.

Example: In figure 1-3, the path $\{I_2, G_1, G_4, G_5\}$ is a critical path with path delay 15. All the other synthesizable paths have path delay less than 15. For the example, there’s only one critical path. In general, there may
exist more than one critical path in the circuit.

All paths that are not specified by the definitions above are false paths. Similar definitions to the ones above can also be found in the work by Abramovici [1] and Jha [9].

1.2 Types of timing analysis

There are two types of timing analysis, static and dynamic. Dynamic timing analysis is the process of enumerating circuit input combinations and simulating these inputs and observing simulation outputs to detect timing violations, and to determine the optimal circuit clock frequency. It is a time-intensive, labor-intensive and error-prone technique. However, this process has the advantage to detect glitches and race conditions. In addition, simulation is the main verification technique, it is easy to use, and it is applicable to clock-less asynchronous design. On the other hand, this process does not guarantee full path coverage, i.e., enumerating and exercising all sensitizable path.

Static timing analysis is the process of enumerating all the sensitizable paths. Designers can use this information to identify and correct timing violation, and to determine the optimal operating circuit clock frequency. This process results in better path coverage than dynamic timing analysis. It identifies critical path much more efficiently than dynamic timing analysis. These advantages of this static timing analysis over dynamic analysis has
promoted its widespread use.

Two important types of timing errors [16] are possible in sequential circuits, namely, hold time violation and setup time violation. Hold time violation is an error when the signal arrives early and setup time violation is the error when the signal arrives too late. These timing errors can occur due to numerous factors, some of which are:

- Unexpected input data
- Temperature or voltage fluctuations
- Manufacturing defects

The primary use of static timing analysis in digital circuits is to verify if the circuit meets the timing requirements and determine the critical path, thus estimating the operating frequency of a synchronous digital circuit. Static timing analysis also proves to be a vital tool to catch design errors during the various phases of circuit manufacture. Additionally, static timing analysis can detect potential glitches during path search phase, clock skew, and slow paths by verifying the paths in the circuit.

### 1.3 Common Timing Analysis Techniques

The behavior of an asynchronous circuit depends on the delays of its components. The most commonly used delay models for asynchronous circuits are,
- Fixed delay
- Bounded delay
- Unbounded delay

In the fixed delay model, the delays of each gate is assumed to be fixed. Since the delays are not a precise representation of the actual delay of the component on chip, it is not very useful in timing verification. However, this model has been used for computing performance metrics by using the average delay of each component. This model has been used in the research work by Burns[4] and Lee[12].

In the bounded delay model, the components are assumed to take an uncertain delay value between a set upper and lower bounds. This model takes into account potential variations due to statistical fluctuations during fabrication, variations in ambient operating temperature, power supply. The unbounded delay model uses a finite non-negative value for representing the delays of each component of the circuit. This model provides a more robust representation of a circuit under delay variations. However, it also yields conservative results.

With diminishing device sizes in the nanometer scales, wire delays have become equally important to gate delays. Also, there is a need to identify the effects of the interaction between wires, known as cross-talk. The work done by Hassoun[7] attempts to address some of these issues. Here, the range of
delay values composes of a variation value $\Delta$ and the predicate. The value of $\Delta$ represents the variation in the gate delay over a fixed range. It can take both positive and negative values. The predicate qualifies the conditions under which the delay variations are valid. Predicates could represent arrival times, slew rates, temperature variations, etc.

\section*{1.4 Automatic Test Pattern Generation}

Automatic Test Pattern Generation (ATPG) is a technique that is used to find input vectors to help distinguish between correct circuit behavior and faulty circuit behavior. ATPG algorithms have been widely used in circuit testing, timing analysis and false path problem. ATPG algorithms use fault models to hypothesize faulty circuit behavior. One popular fault model is the single stuck-at fault model. This model assumes that one of the signal lines in the circuit is assumed to be stuck at a fixed logic value, regardless of the applied input value. Some of the popular ATPG engines include D-Algorithm, Path Oriented DEcision Making (PODEM), Fan-out Oriented (FAN), Satisfiability and pseudo-random test generation.

ATPG algorithms are used in the path sensitization subroutine of static timing analyzers. A signal transition is launched at the input also known as fault effect excitation. It is then propagated until the signal is observed at an output. If the signal cannot be reach an output, then an alternate signal transition is applied at the same input or another input is chosen for
launching the transition.

The thesis work is based on the PODEM ATPG engine. To improve the efficiency of our technique a circuit partitioning scheme has also been implemented. Chapter 3 describes these algorithms in detail.
Chapter 2

Survey of Static Timing Analysis Algorithms

Deterministic static timing analysis programs have been used in the past exhibiting several advantages like linear runtime complexity with respect to the circuit size, extensible for design optimizations, and provides full path coverage. The primary goal of these algorithms have been to determine the worst case execution time of a circuit. This is estimated by identifying the critical path, which is a path between an input and an output of a circuit with maximum delay. Arrival time of a signal at a node is the time taken to propagate that signal to that node in the circuit. Arrival times are calculated by performing a forward topological traversal from inputs to the output using the node and edge delays. Required time of a signal is the latest time at which a signal can arrive at a node conforming to the timing performance constraints particularly, clock cycle. It is calculated by backward topological traversal from the outputs to the inputs. Delay slack
is defined as the difference between required time and arrival time. It is an indicator of how fast or slow a path is performing. Static timing analysis algorithms make use of these concepts to compute the circuit timing.

Circuit timing calculation techniques in the past have met with numerous challenges. Some of the algorithms focusing on techniques to overcome the challenges have been presented below. One of the earliest was the identification and elimination of false paths. False paths are defined as circuit paths which lack an input sensitization vector, thus resulting in a conservative timing estimate. False path elimination can result in a computationally intensive algorithm.

2.1 False Paths and Path Sensitization

Path sensitization techniques and functional relationships between signals have been employed in the past to eliminate false paths. Several efficient methodologies are described in [2], [3]. One technique to find the longest statically sensitizable path in acyclic asynchronous logic circuits is described in [13]. It is based on the assumption that a statically nonsensitizable path can be dynamically sensitizable. Their work also defines the conditions when a path can be identified as dynamically sensitizable.

The algorithm begins by creating a directed graph of the circuit, called event graph. This graph represents both the logic and timing view of the circuit. Nodes in the graph represent events and each node has two events
defined. Events names are formed by combining the circuit element name and logic value it can assume. Logic values are limited to 0 and 1. The edges connecting the nodes are represented in the form of \((n, o)\) where the signal from event \(x\) can be propagated to event \(o\). Propagation delay values between events are represented as weights on the edges, \(d(n, o)\).

The worst case arrival time of a vertex is the latest time when a signal arrives at that vertex with the logic value TRUE (1). Delay of a path \(P\), represented as \(D(P)\) is defined as the sum of weight on all the edges on the path and the worst case arrival time of its root vertex.

To identify the longest sensitizable path from the root vertex to a point in the circuit, which could be an output node,

- Let \(T\) represent the set of paths from root vertex to a target node.

  Let \(P \in T\),

  where the node \(n\) is the target node.

- Define \(M(n, P)\):

  if \(n = \text{target}\): \(M(\text{target}, P) = \infty\)
  
  if \(n \neq \text{target}\): \(M(n, P) = \min\{T(n), M(o, P) - d(n, o)\}\),

  where \(o\) is the successor of \(n\) along path \(P\)

- Define \(\text{min-max}(n)\):

  \(\text{min-max}(n) = \max(M(n, P))\), where \(P \in T\) and,
vertex \( n \) is in the path \( P \)

If a path \( P \) has a delay value greater than the \( \text{minmax}(n) \), then the path is a false path. Once all the true paths have been identified, the program proceeds with determining if the edges in the path are consistent. The consistency of the edges are determined using D-algorithm, where the edge must satisfy the propagation cubes. If an edge does not meet the propagation condition, then the path is cannot be activated and is eliminated. This check for consistency is sometimes referred to as path sensitization criteria.

### 2.2 Symbolic Simulation and Statistical Timing Analysis

Statistical timing analysis and delay models have been widely used in the past to predict the performance of VLSI circuits with respect to clock speed. One such work was presented by Jyu et al.\[10\]. In their work, the delay probability distribution of a given combinational circuit is computed, given the probability distribution of individual gate delay and wire delays. Also, their technique eliminates false paths, that might yield conservative performance probabilities. The delay \( d_g \) of a gate \( g \) is assumed to be a random variable within the bounds \([d_g^l, d_g^u]\). The assumption on the delay bounds have been made from the limits to process variations. Also, the joint probability density function denoted by \( p(\vec{d}_g) \), where \( \vec{d} \) is the vector of all the delay variables of the circuit, is assumed to be provided. The joint probability density func-
tion is a characteristic of the circuit under consideration. If the delays are correlated, then the probability density function is obtained from the layout information.

The algorithm relies on symbolic simulation of the circuit under test. The signal transitions are represented using symbolic waveform expressions. The delays of circuit elements are assumed to be bounded delays. The signal values are assigned functions instead of fixed set of possible values. Symbolic transition sequences are used to represent the timing diagrams for signals. An example symbolic transition sequence is of the form \{f_1(1), f_2(2), f_3(4)\}. This can be translated to, signal evaluates to the function \(f_1\) at time \(t = 1\) and signal evaluates to the function \(f_2\) at time \(t = 2\) and signal evaluates to the function \(f_3\) at time \(t = 4\). The transition sequences on the inputs of a circuit element can be used to compute the output transition sequence of the same gate. This can be performed on all the circuit elements, until the transition sequence is at the primary output for a given signal.

For every combination of delay values, there exists a waveform, \(w\). The set of waveforms, \(W_i\), consists of all the constituent waveforms \(w_{i,j}\). Each \(w_{i,j}\), consists of the terms region of validity \(D_{i,j}\), and the transition sequence \(T_{i,j}\) for that signal \(i\). The region of validity, \(D = \cap_k \{d_k^1 \leq d_k \leq d_k^n\}\) represents the values that \(\vec{d}\) is restricted to. The region of validity, \(D\), is a region within \(R^n\), where \(R^n\) is a set of real numbers of the delays of \(n\) gates.

For each of the waveforms in the set \(W\), if the delay at the last transition
sequence is not equal to the delay at the penultimate transition sequence for a given input combination, \( \{x^p_1, x^p_2, \ldots x^p_m, x^n_1, x^n_2, \ldots x^n_m\} \), then that path is a transitional false path. If the transition at the primary output happens at time \( t_{i,l} \), then \( t_{i,l} \) is said to be the delay for the region \( D_i \). The delay distribution of the circuit is determined by finding the distribution for each \( t_{i,p_i} \), for a given region of validity. Also, only the last transition times for each of the waveforms need to be computed to determine the worst case delay and the delay distribution of the circuit. Several pruning techniques have been mentioned in the paper to make the algorithm more efficient, by dropping those paths that can never be the longest path lengths at the primary outputs.

The results of the algorithm on the ISCAS’85 benchmarks are listed in the paper. The aforementioned algorithm avoids redundant delay computations. The algorithm can predict the variations in the delays and worst case delay due to intra-die, inter-die and process variations. A major disadvantage of the algorithm is that it can result in an exponential complexity on large circuits. Also, the upper and the lower bounds on the gate delays from the library are computed using simple multiplication with fractions. Thus, the algorithm does not adapt the delay bounds to different circuits with varying complexities.
Chapter 3

ATPG Techniques

Automatic test pattern generation (ATPG) is a popular method/technology for identifying sensitization vectors of a circuit to help distinguish between correct circuit behavior and faulty circuit behavior. This method is used widely in circuit testing, timing analysis and false path identification. Additionally, the ATPG algorithms have been adapted for logic optimizations and circuit verification. Accuracy of the ATPG method is dependent on the type of circuit under test (CUT), fault model, and the level of abstraction of the circuit elements. Metrics like test quality, which indicates the amount of modeled defects, and execution time which indicates the number of patterns identified, measure the effectiveness of the ATPG method.

ATPG methodologies have proven as a valuable tool in the area of timing analysis in synchronous, asynchronous and full scan digital circuits. This chapter describes some of the popular and widely used techniques. These techniques are a part of the test system and does not represent a whole test
system.

3.1 D-Algorithm

The D-Algorithm developed by Roth[14] is a technique for computing tests for failures. The algorithm is mainly used as a mechanism for generating a set of test cases for detecting single failure in acyclic logic circuits. D-Algorithm is used to detect single failures s-a-0 and s-a-1 in circuits composed of AND, NAND, OR, NOR, and XOR logic gates.

Computation of D-cubes

D-cube is the representation of a logic function using the symbols $D$ and $\bar{D}$. The following rules apply to the symbols $D$ and $\bar{D}$.

- Symbol $D$ may assume only two values 1 or 0. For a good circuit, $D$ is 1. For a faulty circuit, $D$ is 0.

- Symbol $\bar{D}$ may assume only two values 0 or 1. For a good circuit, $\bar{D}$ is 0. For a faulty circuit, $\bar{D}$ is 1.

The symbol $\bar{D}$ is essentially a complement of $D$. It means that if $D$ takes a value of 1, then $\bar{D}$ takes a value of 0 in a good circuit and vice versa. The set of D-cubes of a logic function is termed primitive D-cubes abbreviated to $pdc$. Figure 3-4(a) represents a two-input NAND gate with it’s singular cover in figure 3-4(b) and corresponding pdc in figure 3-4(c).
Figure 3-4: **Primary D-cube and singular cover of a two input NAND gate**

To construct a D-cube, choose a coordinate from the singular cover and complement the value and the corresponding output. Intersect this new D-cube with the cube cubes in the singular cover. If a non-empty intersection exists, the resulting cube is the D-cube after replacing the coordinates chosen earlier. If the coordinate in the intersected D-cube is 1, then replace it with $D$. If it is a 0, then replace it with $\bar{D}$. This process is repeated on all the combinations of the input and output to compute the pdc. The D-cube $\bar{D}1D$ in the pdc has the following interpretation. If input $X$ is set to 0 and $Y$ is set to 1, then the output $Z$ will be 1. On the other hand, if the input $X$ is set to 1 and $Y$ is set to 1, then the output $Z$ will be 0. These are the two embryonic tests denoted by the D-cube $\bar{D}1D$. 
The D-Algorithm - An execution example

The following section presents a manual simulation of the D-Algorithm. The simulation is performed on the circuit shown in Figure 3-5. A six-valued system is utilized to represent the values at the nodes in the circuit. The six-valued system consists of the symbols $0_6$, $1_6$, $D_6$, $\overline{D}_6$, $\chi_6$ and $X_6$. The symbols $\chi_6$ and $X_6$ are used to denote the incompletely specified values \{0, 1\} and \{0, 1, D, \overline{D}\} respectively. As the first step to simulation, circuit initialization is performed, where all the primary inputs are assigned the value $\chi_6$ and the outputs are assigned $X_6$. Figure 3-6(a) shows the state of the circuit after initialization. The goal of the simulation is to obtain a value of $D$ or $\overline{D}$ at one or more of the primary outputs, given a target fault.

Now, consider a target fault $s-a-\theta$ at the primary input $N_6$. The algorithm
Figure 3-6: The phases of simulation of D-Algorithm
Figure 3-7: (Continued) The phases of simulation of D-Algorithm
Figure 3-8: (Continued) The phases of simulation of D-Algorithm

begins by exciting the fault $s-a-0$ at the fault site, the primary input $N_6$ of NAND gate $G_1$. The procedure, intersects the current values with the fault excitation cubes of the faulty version of the NAND gate. The primary input is assigned a value $D_6$. The intersection yields only one set of local value assignments, namely $N_6 = 1_6$ and $V(c_1) = 1_6$. The state of the algorithm is saved and the current local assignments are marked as tried. The implication procedure is executed next and the state of the circuit is shown in Figure 3-6(b). The values at the gate $G_1$ are justified after the implication procedure. Also, the D-frontier contains the gates $G_2$ and $G_4$, and J-frontier is empty.

The algorithm proceeds to a second round of execution and identifies fault effect propagation of the fault through each of the gates in the D-
frontier as a test generation sub-task. The fault effect propagation procedure, attempts to propagate the fault through the gate G4 in the D-frontier. The procedure, intersects the current values at the inputs and the output of the gate with the fault propagation cubes of the faulty version of the two-input NOR gate. The only local value assignment found is \( V(c_7) = D_6 \). Since the D-frontier has another gate, an alternative assignment is also considered by the fault effect propagation procedure. The alternative assignment is \( V(c_6) = D_6 \), from intersecting the current input and output of the two-input NAND gate G2 with the corresponding fault propagation cubes. Here, the first test generation subtask, i.e. propagation of the fault through the gate G4, is selected and the local value assignment is performed on the circuit. The current state of the algorithm is saved and the corresponding value assignments are marked as tried. The implication procedure is executed next and the circuit values are as shown in Figure 3-7(a). Both the D-frontier and the J-frontier are empty sets.

In the next round of execution, the algorithm tries to identify subtasks. Since both D-frontier and J-frontier are empty, the primary outputs are checked to see if it contains a \( D_6 \) or \( \overline{D}_6 \). The primary outputs \( N_{23} \) and \( N_{22} \) contain the values \( 0_6 \) and \( 1_6 \), backtracking is initialized and the circuit is restored to the previously saved state. Since there exists an alternative assignment, test generation subtask is not performed and the alternative local value assignments from the stack is used. The restored state of the circuit
is as shown in Figure 3-6(b). The algorithm identifies a subtask, which is propagation of the fault through the NAND gate G₂.

In this round, the algorithm has identified to propagate the error through gate G₂. The only remaining local value assignment is \( V(c₆) = D₆ \). This assignment is marked as tried and the current state of the algorithm is saved onto the stack. The implication procedure is followed. The implication procedure justifies the values at the gates G₂, G₄, G₅ and G₆. The D-frontier updated by the implication procedure contains the gate G₇ as a potential fault propagation gate and the J-frontier is empty. The value assignments after this step is as shown in Figure 3-7(b). The algorithm now identifies another possible round of execution, since the D-frontier is not empty and the fault has not reached the primary outputs.

In the next round the fault is attempted to be propagated through the NAND gate G₇. The fault propagation now identifies the only local value assignment \( N_{22} = D₆ \). The implication procedure is executed, which justifies the gates G₇ and G₃. The D-frontier and J-frontier are empty. The state of the algorithm is saved and the current local value assignments are marked as tried. The algorithm proceeds to the identification of a test generation subtask. The current circuit value assignments are shown in Figure 3-8(a).

In the next round, the algorithm finds that both the D-frontier and the J-frontier are empty. The algorithm also finds that one of the primary outputs of the circuit-under-test \( N_{22} \) has a value of \( D₆ \). The algorithm concludes that
a test vector has been successfully generated to test the fault at the primary input N₆. The algorithm now reports the vector \{1, 1, 1, 1, 1\} as a test for the target fault s-a-0 at the primary input N₆.

Since the number of possible input combinations can be exponential, logic justification can be truncated. This results in a false path not being identified. This is a well known limitation of the D-algorithm. This has been addressed in the following algorithm.

### 3.2 PODEM-X

PODEM (Path Oriented DEcision Making) developed by Goel[6] is an ATPG algorithm where the search space is constituted of values assigned at primary inputs and only forward implication is performed to propagate the signals. This restriction to search space implies that the number of backtracks is $2^n$, for a $n$-input circuit. Also, since the values are assigned only at the primary inputs, there are no unjustified lines during the execution of the test generation. By performing only forward implication, the runtime complexity is greatly reduced as compared to the D-Algorithm. Another consequence is that during execution, the stack only stores the values at the primary inputs and their untried alternative values. When a backtracking is initiated, these untried alternatives are assigned and a forward implication is performed.

Unlike D-Algorithm, no line justification is performed during path sensitization. The only subtasks performed are fault effect excitation and fault
effect propagation. If multiple subtasks are identified during sensitization, the following rules for prioritizing alternatives are performed to reduce complexity.

- Among multiple subtasks, if some are necessary and the others are not, then the subtasks that are necessary are chosen.

- Among the set of subtasks that are identified as necessary and chosen, the one with the highest cost is selected.

- If some of the subtasks can be identified as individually sufficient to accomplish the overall task among a set of subtasks, the one with the lowest cost is chosen.

3.3 FAN Algorithm

FAN algorithm visualizes the circuit in terms of cones. These cones represent the re-convergent fan-outs in the circuit. Both D-Algorithm and PODEM does not handle the case of re-convergent fan-outs. In that sense, FAN is an improvement over both algorithms. The efficiency of the path search is greatly improved.

3.4 Pseudorandom test generation

This technique employs pseudorandom number generator to generate test vectors. This is a relatively simple approach to ATPG. The efficiency of
this method varies with fault characterization. This methodology is used to identify test vectors quickly and those vectors that are not identified by this method can be generated using one of the previously mentioned ATPG techniques.

3.5 VIPER

VIPER technique developed by Abraham et al.[5], is an acronym for VIgorously sensitizable Path ExtractoR. Existing timing verification algorithms take huge computation time to eliminate false paths particularly with large circuits. In this work, the efficiency and speed is achieved by employing a progressive path sensitization criterion.

Existing algorithms disregard the logic behavior to achieve faster computation times. This however, results in conservative timing metrics. Timing verification methods are a two step process involving generation of list of paths with delays greater than a threshold followed by path sensitization. The path list generation process additionally sorts the paths in decreasing order of path delay values. Static sensitization approach, assumes that the side-inputs have non-controlling values. Dynamic sensitization approach utilizes arrival times of side-inputs avoid conservative results. Dynamic sensitization is preferred over static sensitization for this reason, although it has some limitations.

Early static and dynamic approaches suffer from several drawbacks. Firstly,
a huge list of paths with delays greater than a threshold needs to be generated. Also, the number of paths is gradually increased to find the critical path, if it’s not in the initial path list. Secondly, the generated paths are tested for timing conflicts until the critical path can be found. During this process, several partial paths are redundantly checked which requires a lot of system time for large circuits.

Another inefficiency is introduced during the path sensitization check, where D-Algorightm is used to verify the paths. This involves a lot of backtracking at every internal node, which is the cause of this inefficiency. Sensitization checks through multiple paths and association of different delay values corresponding to a logic 0 or 1 is not handled. The following section gives some details of the VIPER technique.

VIPER begins by preprocessing the circuit under test (CUT), transforming it into an acyclic directed graph. The gates and the interconnections are represented as nodes and edges respectively. The nodes and edges are assigned weights corresponding to their associated delays. During this transformation, a pseudo-input or source node and a pseudo-output or sink node are added. All the primary inputs are connected to the pseudo-input and the outputs are connected to the pseudo-output.

The sink delay for a gate is the maximum delay of a path from the node to the pseudo-output and represented as $sID(G)$. It’s expressed as

$$sID(G) = gD(G) + eD(e_c) + \max_i\{sID(g_i)\},$$
where \( i \) is the range over all successor gates. The sink delay for the PO is zero. Similarly, source delay for a gate is the maximum delay of a path from the pseudo-input to that node and represented as \( \text{soD}(G) \). It’s expressed as

\[
\text{soD}(G) = \max_i \{\text{soD}(g_i)\} + \text{eD}(e_c) + \text{gD}(G),
\]

where \( i \) is the range over all input gates. The source delay of the PI is zero. Also, \( \text{eD}(e_c) \) is the edge delays of the interconnect, and \( \text{gD}(G) \) is the propagation delay of gate \( G \).

The preprocessing stage ends here and the structurally longest path can be found by path tracing. All the inputs, connected to the pseudo-input are sorted in non-increasing order of their sink delays. The critical path tracing begins with the first successor gate of the pseudo-input. If a critical path is not found, then the second successor gate is chosen and path tracing is performed. This continues until all the primary inputs are exhausted. The chosen primary input for path tracing is denoted by \( \text{start.PI} \). Also, only the PIs with sink delay greater than or equal to a threshold is chosen for path tracing. This is required to reduce the search space. Logic values used in propagation are 0, 1 and \( X \). To indicate a fault, special signals \( P \) and \( N \) are applied to the \( \text{start.PI} \).

During the critical path tracing, the logic value of all the gates are initialized to \( X \) except the \( \text{start.PI} \). One of the special signal, say \( P \) is applied on the \( \text{start.PI} \) and propagated through to a primary output. If the primary output has a \( P \) or \( N \) signal, then the path is sensitizable. If a critical path is
not found the other special signal $N$ is applied and the tracing is performed again.

During partial path sensitization, those gates that have a $P$ or $N$ as an input and $X$ as output, are termed as candidate gates. Propagation delay is the actual delay of a partial path from the *start.PI* to the gate that has a $P$ or $N$ as input. The propagation delay of a gate $G$ is denoted by $prD(G)$ whose value is stored in the candidate gates. A general expression for gate delay is

$$prD(G) = prD(g_s) + eD(e_c) + gD(G)$$

For different types of gates based on whether the side-inputs are controlling or non-controlling, the expression varies either using minimum propagation delay or maximum propagation delay for that gate for the each of the special signals.

Arrival time for a gate is the time required for the output of a gate to have a stable logic value. For a gate $G$, it is denoted by $AT(G)$. Depending on whether the input to the gate is controlling or non-controlling, arrival time is expressed as follows -

For controlling input, where $i$ ranges over all the inputs of the gate $G$,

$$AG(G) = \min_i \{AT(g_i)\} + eD(e_c) + gD(G)$$

For non-controlling input,

$$AG(G) = \max_i \{AT(g_i)\} + eD(e_c) + gD(G)$$
Implication process attempts to propagate the signal to a primary output. During this stage, vigorous sensitization is performed. Assume that $g_0l$ is the online input for a gate $G_i$ where $i$ ranges from 1 through $m$. Vigorous sensitization rules for candidate gate evaluation are as follows,

- When $g_0l$ has NCV($G_i$), all side inputs have NCV($G_i$) and the signal of $g_0l$ should arrive last.

- When $g_0l$ has CV($G_i$), there should be no side inputs that arrives earlier than CV($G$)

After the implication process, to limit the number of candidate gates, a cost function (CF) is introduced. This pruning is performed during evaluation of propagation delays of candidate gates. The cost function for a gate is expressed as follows,

$$\text{CF}(G) = \text{prD}(G) + \epsilon D(e_c) + \text{siD}(g_{fs})$$

Here, $g_{fs}$ is the first successor gate of $G$ with maximum sink delay. Only candidate gates whose CF is greater than or equal to the threshold delay are chosen since other candidate gates cannot be on a sensitizable path. To improve efficiency, the selected candidate gates are saved in non-decreasing order of their CF values. The gate closest to the pseudo-output has the smallest sum of edge and sink delays.

To detect any obstacles during propagation of a signal, candidate gates need to be verified for a free path to the pseudo-output. A free path is
defined as a path in which all gates in the path has the output value $X$. A gate which has the $X$-path is closest to the primary output is selected for propagating the signal in most ATPG algorithms. To propagate a signal in VIPER, maximum CF is used as an additional criteria. When a free path has been identified, justification is performed and a PI stack is created with the most recently justified PI on top of the stack. This PI is the new $start_{PI}$ and implication is performed on this input. If no candidate gate satisfies the free path check, then implication is performed after a backtrack using the inverted value for the same input.

If a $P$ or $N$ signal reaches the PO, then there exists a vigorously sensitizable path from $start_{PI}$ to that PO. Thus, from the propagation delay of the PO, a critical path can be identified, if the delay is greater than or equal to the threshold and, if the CF’s of all the candidate gates is smaller than the threshold. One obvious drawback of this technique is the large number of backtracks to identify a critical path.
Chapter 4

Algorithm

The critical path of a circuit is the longest sensitizable path in a circuit. It also determines the worst-case execution time of the circuit, i.e. the frequency of operation. The following sections provide detailed descriptions of our algorithm and the heuristic technique.

4.1 Circuit Levelization

Here, the given circuit is levelized to get the delay values for each gate in the circuit. The input circuit in verilog/VHDL description is synthesized using Leonardo Spectrum and TSMC 180nm process library. During synthesis, the circuit is transformed into it’s gate level representation in the form of a net-list. The synthesis tool outputs the net-list as EDIF description and the gate/wire delays as SDF (Standard Delay Format) description. The EDIF and SDF descriptions are input to a wrapper script which then translates the pin/node interconnect information and the delays into a levelized bench
description. An example bench format for the circuit C17 from ISCAS ’85 is presented in figure 4-9.

```
# c17
# 5 inputs
# 2 outputs
# 0 inverter
# 6 gates ( 6 NANDs )
INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)
OUTPUT(22)
OUTPUT(23)
10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

Figure 4-9: Example Bench format

## 4.2 Preprocessing

In the preprocessing stage, the bench description of the circuit is read and the circuit is transformed into an acyclic directed graph. A pseudo-input and pseudo-output node are added to the graph. The pseudo-input is connected to all the primary inputs and pseudo-output is connected to all the primary
outputs. Also, rise and fall delays are assigned to their respective gates. Next, the source delay and sink delay of each of the gates are calculated.

- **Source delay** of a gate is the maximum delay of the partial paths from the pseudo-input to that gate.

- **Sink delay** of a gate is the maximum delays of the partial paths from the pseudo-output to that gate.

The gates in the first level have source delay values equal to their gate delays for rising and falling signal transitions. For sink delay, the gate level is counted in reverse starting from the primary output. Thus, the gates in the first level, have sink delays equal to their respective gate delays. If any gate has more than one fan-in or fan-out, the branch which has the highest delay is chosen as the source or sink delay respectively. Each gate has two source and sink delays corresponding to a falling or rising signal transition. For inverting logic gates like "NAND", "NOR", and "INV", the source and sink delays for rising transition must be accumulated to the falling transition delay of the successor gate.

### 4.2.1 Partitioning

Partitioning is the process which identifies the paths between a primary output and the primary inputs. This process is performed on all the primary outputs until all the partitions are identified. Figure 4-10 shows the partitions in an example circuit. The partitions are identified by marking the
gates between the primary output and the primary inputs in the partition. Later in path sensitization subroutine, we have implemented a heuristic to guide the search within the partition chosen for sensitization. The heuristic is a condition which returns \textit{true} value if there exists a partial path from the output of the gate where is transition/fault is at to the primary output of the partition. It returns \textit{false} value if no such partial path exists.

![Partitions in an example circuit](image)

\textbf{Figure 4-10: Partitions in an example circuit}

Henftling \textit{et al.} [8] have used a partitioning scheme called \textit{Path Hashing} to speed-up path delay fault testing. Kocan and Gunes [11] have also used a partitioning scheme to improve path delay fault coverage calculation.
4.3 Path Sensitization

Here, a modified version of PODEM is used to perform path sensitization, eliminating false paths and identifying the critical path. The alteration to PODEM is with the selection of gates from the D-frontier. Figure 4-11 lists the pseudo-code for our technique.

4.4 Algorithm

The algorithm begins by setting the inputs of all gates of the circuit to X. Like PODEM and D-Algorithm, D-cube is used to represent signal transition. As an example, figure 3-5 is used to exemplify our technique. A transition $D$ is set on the primary input and it’s propagated until it reaches a primary output. In our example circuit, the a transition of $D$ is applied at the primary input $N_6$. This is also called fault effect excitation followed by fault effect propagation. During fault effect excitation, gate $G_1$ had an input $D$ and an output X. The fault excitation cubes for the gate are intersected and backtrace is initiated. Backtrace determines that input $N_3$ must be assigned logic "1" for signal propagation. Implication is then performed which sets the output of gate $G_1$ to $\overline{D}$ and adds the gates $G_2$ and $G_4$ to the D-frontier in the non-increasing order of cost. Before adding the gates to the D-frontier, the heuristic determines that partial paths exist to primary output $N_{23}$ via gates $G_2$ and $G_4$. The cost or difficulty in propagating a signal in our technique is defined as the sum of source and sink delay at a particular gate.
MAIN() {
    compute gate source and sink delay;
    Partition the circuit;
    Mark the gates within the partitions;
    for( Start_PI=0; Start_PI < number of inputs; Start_PI++ ){
        assign signal to Start_PI;
        for( Dest_PO=0; Dest_PO < number of outputs; Dest_PO++ ){
            while ( Critical Path not found ){
                if ( test not possible ){
                    backtrack;
                    implication;
                    update D-frontier;
                    if ( redundant ){
                        return ( no sensitizable path );
                    }
                }else{
                    justify the current objective;
                    backtrace;
                    implication;
                    update D-frontier;
                }
            }
        }
    }
    return ( Sensitization Vector, Gate List, 
              and source and sink delay );
}

Figure 4-11: STA pseudo code.
The state of the circuit after first implication is as shown in figure 4.4. Assuming that the sum of source delay and sink delay of gate $G_2$ is greater than that of gate $G_4$, the objective is now to propagate the fault through gate $G_2$. In order to propagate the fault through gate $G_2$, backtrace identified that the primary input $N_2$ should be logic "1". Implication is then performed and the output of gate $G_2$ is assigned $D$. Also the output of gate $G_5$ is assigned $\overline{D}$. The gates $G_6$ and $G_7$ are added to the D-frontier in non-increasing order of cost. The state of the circuit after second implication is as shown in Figure 4-13(a).

Figure 4-12: **Initial Assignment and First Implication**

Again, fault effect propagation is performed assuming that gate $G_6$ has the highest cost. The backtrace procedure identifies that the primary input
Figure 4-13: The phases of simulation
N_7 must be assigned logic "1" in order for the signal to propagate through gate \(G_6\). The output at gate \(G_6\) also primary output \(N_{23}\) is assigned \(D\). The D-frontier is empty and the algorithm has identified a path that’s sensitizable. The sensitization vectors for the path \([G_1, G_2, G_6]\) is \([1, X, 1, 1, 1]\) and \([1, X, 1, 0, 1]\) corresponding to the primary inputs \([N_2, N_1, N_3, N_6, N_7]\). The final execution state of the circuit is shown in figure 4-13(b).
Chapter 5

Results

The methodology that we have mentioned was implemented in C language. For testing purposes, ISCAS ’85 Combinational and ISCAS ’85 Sequential benchmarks were adopted. There were two sets of tests that were performed on the benchmarks.

One set of tests used the algorithm without applying heuristics. The results of the test procedure on the ISCAS ’85 Combinational benchmarks are presented in Table 5.1 and the ISCAS ’89 Sequential benchmarks are presented in Table 5.3 respectively.

The other set of tests used the algorithm applying heuristics to indicate the efficiency of the proposed idea. The results of the tests on ISCAS ’85 Combinational benchmarks are shown in Table 5.2 and the ISCAS ’89 Sequential benchmarks are shown in Table 5.4.

Performance metrics were captured in terms of CPU time in seconds. Also the critical delay using both techniques were captured.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Threshold</th>
<th>CPU Time in seconds</th>
<th>Worst Case Critical Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c17</td>
<td>0.0</td>
<td>0.000</td>
<td>0.500</td>
</tr>
<tr>
<td>c432</td>
<td>4.2</td>
<td>211.622</td>
<td>6.060</td>
</tr>
<tr>
<td>c499</td>
<td>1.75</td>
<td>2234.890</td>
<td>3.630</td>
</tr>
<tr>
<td>c880a</td>
<td>4.8</td>
<td>668.941</td>
<td>6.500</td>
</tr>
<tr>
<td>c1355</td>
<td>1.5</td>
<td>2265.844</td>
<td>3.880</td>
</tr>
<tr>
<td>c1908</td>
<td>4.4</td>
<td>1364.219</td>
<td>5.680</td>
</tr>
<tr>
<td>c2670</td>
<td>5.9</td>
<td>1118.986</td>
<td>7.520</td>
</tr>
<tr>
<td>c3540a</td>
<td>7.2</td>
<td>663.025</td>
<td>8.610</td>
</tr>
<tr>
<td>c5315</td>
<td>5.4</td>
<td>1316.703</td>
<td>6.050</td>
</tr>
<tr>
<td>c7552</td>
<td>12.15</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.1: ISCAS '85 Results without Heuristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Threshold</th>
<th>CPU Time in seconds</th>
<th>Worst Case Critical Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>c17</td>
<td>0.0</td>
<td>0.007</td>
<td>0.500</td>
</tr>
<tr>
<td>c432</td>
<td>4.2</td>
<td>73.601</td>
<td>6.030</td>
</tr>
<tr>
<td>c499</td>
<td>1.75</td>
<td>2302.704</td>
<td>3.630</td>
</tr>
<tr>
<td>c880a</td>
<td>4.8</td>
<td>78.854</td>
<td>6.500</td>
</tr>
<tr>
<td>c1355</td>
<td>1.5</td>
<td>2311.369</td>
<td>3.880</td>
</tr>
<tr>
<td>c1908</td>
<td>4.4</td>
<td>1062.631</td>
<td>5.739</td>
</tr>
<tr>
<td>c2670</td>
<td>5.9</td>
<td>17.358</td>
<td>7.520</td>
</tr>
<tr>
<td>c3540a</td>
<td>7.2</td>
<td>118.336</td>
<td>8.510</td>
</tr>
<tr>
<td>c5315</td>
<td>5.4</td>
<td>141.020</td>
<td>6.050</td>
</tr>
<tr>
<td>c7552</td>
<td>12.15</td>
<td>3439.522</td>
<td>13.070</td>
</tr>
</tbody>
</table>

Table 5.2: ISCAS '85 Results with Heuristics
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Threshold</th>
<th>CPU Time in seconds</th>
<th>Worst Case Critical Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>s298</td>
<td>1.0</td>
<td>0.327</td>
<td>2.980</td>
</tr>
<tr>
<td>s344</td>
<td>0.5</td>
<td>29.143</td>
<td>4.870</td>
</tr>
<tr>
<td>s641</td>
<td>4.2</td>
<td>1.422</td>
<td>6.319</td>
</tr>
<tr>
<td>s713</td>
<td>4.2</td>
<td>1.427</td>
<td>6.319</td>
</tr>
<tr>
<td>s838</td>
<td>9.2</td>
<td>7.779</td>
<td>11.669</td>
</tr>
<tr>
<td>s1196</td>
<td>3.1</td>
<td>6.023</td>
<td>5.110</td>
</tr>
<tr>
<td>s1423</td>
<td>9.7</td>
<td>1628.258</td>
<td>15.080</td>
</tr>
<tr>
<td>s5378</td>
<td>2.5</td>
<td>5141.981</td>
<td>4.160</td>
</tr>
<tr>
<td>s9234</td>
<td>4.1</td>
<td>3040.821</td>
<td>5.990</td>
</tr>
<tr>
<td>s13207</td>
<td>6.1</td>
<td>61.481</td>
<td>6.749</td>
</tr>
<tr>
<td>s35932</td>
<td>1.5</td>
<td>369009.983</td>
<td>2.98</td>
</tr>
</tbody>
</table>

Table 5.3: ISCAS ’89 Results without Heuristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Threshold</th>
<th>CPU Time in seconds</th>
<th>Worst Case Critical Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>s298</td>
<td>1.0</td>
<td>0.160</td>
<td>2.980</td>
</tr>
<tr>
<td>s344</td>
<td>0.5</td>
<td>1.240</td>
<td>4.870</td>
</tr>
<tr>
<td>s641</td>
<td>4.2</td>
<td>1.774</td>
<td>6.319</td>
</tr>
<tr>
<td>s713</td>
<td>4.2</td>
<td>1.757</td>
<td>6.319</td>
</tr>
<tr>
<td>s838</td>
<td>9.2</td>
<td>2.313</td>
<td>11.669</td>
</tr>
<tr>
<td>s1196</td>
<td>3.1</td>
<td>6.647</td>
<td>5.110</td>
</tr>
<tr>
<td>s1423</td>
<td>9.7</td>
<td>336.577</td>
<td>14.740</td>
</tr>
<tr>
<td>s5378</td>
<td>2.5</td>
<td>124.775</td>
<td>4.160</td>
</tr>
<tr>
<td>s9234</td>
<td>4.1</td>
<td>102.495</td>
<td>5.990</td>
</tr>
<tr>
<td>s13207</td>
<td>6.1</td>
<td>299.281</td>
<td>6.749</td>
</tr>
<tr>
<td>s35932</td>
<td>1.5</td>
<td>14806.746</td>
<td>2.980</td>
</tr>
</tbody>
</table>

Table 5.4: ISCAS ’89 Results with Heuristics
5.1 Conclusion

In this work, we have presented an efficient way to find a synthesizable path with the longest delay. A partitioning scheme was implemented to reduce the search space of the algorithm. The partitioning scheme represents the circuit as cones. A transition is applied at the primary input on the cone and is propagated through the nodes in the cone until it reaches a primary output. The number of paths from a gate in the cone to a primary output is used as a heuristic in the ATPG subroutines to ensure that a synthesizable path with the longest delay is obtained easily. The results of the algorithm for the ISCAS ’85 and ISCAS ’89 benchmarks shows the improvements and practicality over a non-heuristic based technique in a majority of the test circuits.
Bibliography


