Static Timing Analysis of Dynamically Sensitizable Paths *

S. Perremans, L. Claesen, H. De Man †

IMEC, Interuniversity Micro Electronics Center,
VSDM division, Kapeldreef 75, 3030 Leuven, Belgium

Abstract

This paper describes a new method for solving the false path problem in static timing analysis of acyclic, combinational circuits. The conditions under which a path is false are accurately defined. The fact that these conditions explicitly take into account the dynamic behaviour of the circuit, constitutes the main contribution of the paper. An algorithm for computing the longest dynamically sensitizable paths in an acyclic, combinational circuit is presented.

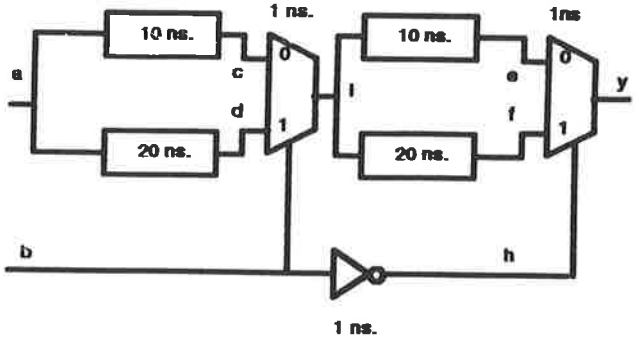

Figure 1: Example of a false path.

1 Introduction

Static timing analysis has become a widely used technique for the verification of digital MOS VLSI circuits. Its ability to locate critical paths, without requiring input excitations, offers a distinct advantage over simulation. Static timing verifiers examine the circuit for a single clock cycle, making an abstraction of the large number of possible states in a sequential machine. By considering only one state, corresponding to the worst case working conditions, all timing constraints can be checked in a single run. Thus, the value-independent approach provides the advantage of complete coverage and fast execution times.

In order to determine the worst case behaviour of the circuit within one clock cycle, the timing properties of the circuit are modelled - explicitly or implicitly - by a directed graph. The vertices in the graph represent events. An edge is placed between two events when one could cause the occurrence of the other. The delay between the two events is represented by a weight on the edge. Finding the latest time of occurrence of all events can then be viewed as a longest path problem in a directed graph. Most existing timing analysis tools have taken either a *path enumeration* or a *block oriented* approach for finding the longest paths. In the path enumeration technique, all paths in the graph are traced. Block oriented methods find only the slowest path to any point in the circuit.

*This research was sponsored by the Belgian IWONL and MIETEC.

†Professor at Katholieke Universiteit Leuven.

Both of these techniques ignore the functional relationships between signals. Thus, they may report critical paths that can never be activated in practice. For instance, in Figure 1, the maximum delay from a to y is calculated to be 42 ns. In practice, both multiplexers cannot select the 1-input at the same time. Consequently, if a and b stabilize at 0 ns., the true worst case arrival time of y is 32 ns.

The presence of false paths results in a loss of accuracy and undermines the user's confidence in the timing analysis tool. Moreover, the false paths hide the real problem areas. Thus, eventual optimisation efforts will be spent on the false paths instead of on the true critical paths, resulting in a waste of silicon area and power consumption.

Two main sources of false paths have been identified: *incorrect signal flow* and *logic dependency* between signals. Timing verifiers that operate at the switch level encounter the problem of incorrect signal flow. Due to the bidirectional nature of MOS transistors, the intended signal flow in structures such as barrel shifters and other pass transistor arrays is not always obvious. The second source of false paths, logic dependency, plagues static timing analysis both at the switch and the logic-block level. In most timing verifiers, the possibility of *case analysis* is included to alleviate this problem. For example, in order to prevent the path *a-d-i-f-y* in Figure 1 from being reported, a separate analysis is required for the case that $b = 1$ and $b = 0$. Case analysis requires that the user provides the specific information needed to eliminate false paths. This can be extremely cumbersome if many cases have to be considered.

This paper describes techniques that automatically account for the logic dependency between signals. In the next section, existing methods are reviewed. It is shown that these methods find the longest *statically* sensitizable

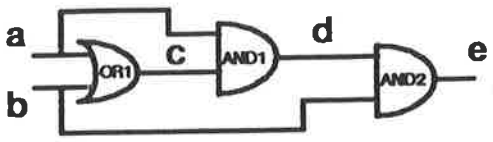

Figure 2: Example of a logic circuit.

path. Because a statically nonsensitizable path still can be *dynamically* sensitizable, the existing methods are not guaranteed to find an upper bound to the true critical path. Section 3 introduces the notation and the model that we use to represent a circuit description. In section 4, conditions for dynamic sensitizability are defined. We present an algorithm for finding the longest dynamically sensitizable paths in an acyclic circuit. Section 5 deals with performance issues. Section 6 summarizes the paper.

2 Previous Approaches

Techniques for false path elimination, using the functional relationships between signals, have been described in [3][1][2]. Basically, these methods are based on tracing a number of paths from an input towards a number of outputs, or vice versa, collecting conditions under which a path is not blocked. If an inconsistent condition is obtained, then the path is always blocked and contribution to the output's arrival time can be ignored. The path is said to be *false*, *nonfunctional*, or *nonsensitizable*.

For example, assume that we are computing the longest path in Figure 2. Suppose that each gate has unit delay. We start at input a and try to proceed through gate $OR1$. In order to sensitize $OR1$ to the transition at node a, we must set $b = 0$. The condition "$b = 0$" is called the *propagation condition* of the OR gate with respect to a transition at its input a. Indeed, if the propagation condition "$b = 0$" is not met, a transition at input a cannot propagate to the output of the gate. From c we proceed through gate $AND1$, for which we must set the propagation condition "$a = 1$". In trying to go from d to e, we must set $b = 1$, which is inconsistent with our previous requirements. The path *a-c-d-e* can therefore be classified as nonsensitizable. An analogous reasoning leads to the rejection of path *b-c-d-e*.

In a previous version of the SLOCOP timing verifier [3], false path elimination was done in a post-processing step. The n longest paths were computed and sorted by decreasing length. Then, those paths were tested one after the other using test pattern generation methods, until a sensitizable path was found. If all n paths proved false, a new run with a larger value of n had to be made.

The method described in [2] eliminates the false paths as part of the longest path search. During the search, whenever a gate is added to a partial path its propagation conditions are checked for compatibility with the propagation conditions of all gates already in the path. If an incompatibility arises, the gate is not added and the search switches to other alternatives.

Time	Signals				
	a	b	c	d	e
-1	0	0	0	0	0
0	1	0	0	0	0
1	1	1	1	0	0
2	1	1	1	1	0
3	1	1	1	1	1

Figure 3: Dynamic behaviour of the circuit shown in Figure 2.

The false path algorithms that we have mentioned above, introduce a number of useful concepts. However, the propagation conditions that they rely on, are largely based on intuitive and implicit assumptions. For instance, the path *a-c-d-e* in Figure 2 is considered to be nonsensitizable. It is argued that any signal starting from input a, will get blocked either at gate $OR1$ or at gate $AND2$, because node b cannot take a high and a low logic value at the same time. The underlying assumption is that the value of b remains constant, during the time it takes for the signal to propagate from a to e. In general however, this condition is not satisfied, because b is free to change value. Figure 3 illustrates such a situation. Node b is 0 at the moment when the signal passes through gate $OR1$, but switches to 1 by the time the signal reaches $AND2$. Hence, the path *a-c-d-e* is activated. If each gate has a unit delay, the arrival time at the output y is 3 units. For the same situation, the methods of [2] [3] [1] predict a worst case arrival time of 2, an underestimate of 1 unit. Notice that this difference can be made arbitrarily large by introducing additional delay elements at the output of $OR1$.

The existing false path algorithms find the longest *statically* sensitizable path, because they consider the propagation conditions along a path as static values. A node that is set by a propagation condition is treated in exactly the same way as if that node was forced permanently to the Vdd or ground. Detecting an inconsistency between propagation conditions can then be viewed as searching for a short circuit situation. In reality however, the propagation condition of a gate must only be satisfied when a signal is moving through the gate. Therefore it is still possible that a statically nonsensitizable path can be *dynamically* sensitized, as shown in the previous example. Consequently, the previous methods are not guaranteed to find an upper bound to the true critical path. In fact, they may underestimate an arrival time by an arbitrarily large amount. In this paper, a new method for finding the longest dynamically sensitizable path is introduced.

3 The event graph

This section introduces the notation and the model that are used to represent a circuit description. The circuit model includes a logic view and a timing view. An interesting feature consists of the fact that both views are integrated into one single graph representation.

A circuit will be represented by a directed graph, called

an *event graph*. The vertices in the graph represent *events*. For each circuit node a, two events are defined: $a0$ and $a1$. These events are designated by the name of the circuit node and a logic level. The logic level takes either a low (0) or a high (1) value, and refers to the steady state value of the node. The vertex $a0$, for example, corresponds to the event: " circuit node a reaches logic level 0 in steady state ". The two vertices $a0$ and $a1$, that represent one circuit node a are called *partner* vertices. This will be denoted as $a0 = P(a1)$ and $a1 = P(a0)$. An edge (v, w) is placed between events v and w, if v can cause the occurrence of w. We say v is a *predecessor* of w, and w is a *successor* of v. The delay $d(v, w)$ between the two events is represented as a weight on the edge.

An event can take two values: TRUE or FALSE, according to the assumption that the event will or will not occur. For instance, if the event $a0$ takes value FALSE, we assume that the steady state logic value of circuit node a is not 0.

The vertices can be partitioned into two categories: AND-vertices and OR-vertices. The AND- or OR-operator define how the value of a vertex w logically depends upon the value of its predecessors $v_1, v_2, \ldots, v_k$. For example, if $w = AND(v_1, v_2)$ then:

AND

		v_1	
		FALSE	TRUE
v_2	FALSE	FALSE	FALSE
	TRUE	FALSE	TRUE

(1)

If $w = OR(v_1, v_2)$ then:

OR

		v_1	
		FALSE	TRUE
v_2	FALSE	FALSE	TRUE
	TRUE	TRUE	TRUE

(2)

Also, partner vertices have complementary values. If $w = P(v)$:

P

v	
FALSE	TRUE
TRUE	FALSE

(3)

Figure 4 shows the event graph for a NAND gate. If both inputs a and b are 1, the output switches to a 0. A low logic level at one of the inputs is sufficient to drive the output to 1. The delays between two events can be gathered from a circuit simulation that demonstrates the causal relation between those events. For instance, to obtain the delay *del1* in Figure 4, a rising input transition has to be applied to the input a, while the input b is kept stable at a high logic value.

In order to model a combinational MOS circuit as a system of AND- and OR- vertices, we use a technique described in [5]. The logic behavior of a circuit element, which realizes the combinational function y, can be concisely represented by the *prime implicants* of y and $\bar{y}$. Consider the CMOS gate, shown in Figure 5a. The prime implicants of $\bar{y}$ are ab and bc, and the prime implicants of

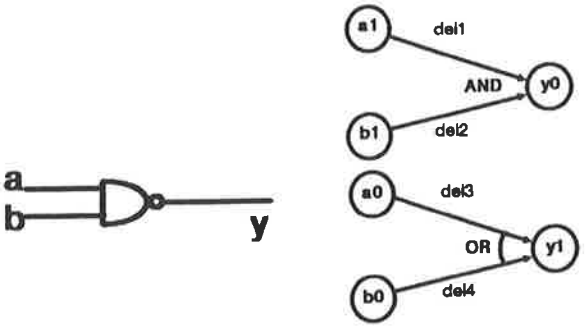

Figure 4: Event graph for a NAND gate.

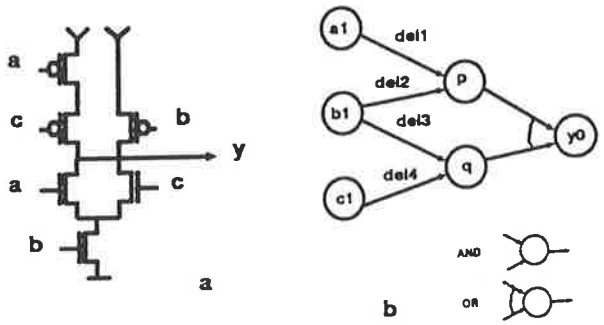

Figure 5: Event graph for a CMOS gate.

y are $\bar{a}\,\bar{c}$ and $\bar{b}$. The output y is driven to 0 if one of the prime implicants of $\bar{y}$ is activated. This occurs when $a = 1$ and $b = 1$, or when $c = 1$ and $b = 1$. Consequently, the event $y0$ can be modelled as shown in Figure 5b. Note that the AND-vertices p and q do not correspond to actual circuit nodes, but represent the prime implicants ab and bc. The delay *del2* can be determined by considering all situations in which a rising transition on input b triggers the prime implicant ab, namely:

- a rising transition on node b, while $a = 1$ and $c = 0$
- a rising transition on node b, while $a = 1$ and $c = 1$.

For our longest path analysis, only the situation with the largest delay is of interest. In this case, the first situation will probably yield the maximum value for *del2*.

4 Path delay analysis

Once an event graph has been constructed, it can be used to determine the maximum delay between a primary input and any node in the circuit that it affects. We propose a new longest path algorithm that provides the ability to eliminate false paths, while still obtaining an upper bound to the true critical paths. Before we discuss this technique, we need a few definitions.

4.1 Definitions

A *worst case arrival time* $T(v)$ of a vertex v is defined as an upper bound to the latest point of time when the vertex can take the value TRUE. For example, the statement

"$T(a1) = 5ns$." means that if the circuit node a reaches a high logic value in steady state, this event cannot occur after 5 ns. A primary input to the design will be called a *start vertex*. Let v_1 be a start vertex. A sequence of vertices $(v_1, v_2, \ldots, v_k)$ is a *path* from vertex v_1 to vertex v_k. Only paths that start from a start vertex will be considered. The *delay* $D(P)$ of a path P is the sum of the delays of its edges and of the worst case arrival time of its start vertex.

4.2 Sensitizability of a path

The worst case arrival time $T(w)$ of a vertex w can be computed as the maximum delay along a sensitizable path that ends in w.

Definition A path is *sensitizable* if the propagation condition of each vertex along the path is satisfied.

Definition The *propagation condition* of a vertex w is defined by the following expressions.

Assume that we are following a path P_j that ends in vertex v_j. Suppose that we want to add an edge (v_j, w) to P_j, such that a path $P = P_j \oplus (v_j, w)$ is obtained.[1] Let $v_1, v_2, \ldots, v_j, \ldots, v_k$ be the predecessors of w. If $w = AND(v_1, v_2, \ldots, v_k)$, then the propagation condition of w with respect to its predecessor v_j is defined by the cube:[2]

v_1	v_2	...	v_j	...	v_k	w
TRUE	TRUE	...	TRUE	...	TRUE	TRUE

(4)

If $w = OR(v_1, v_2, \ldots, v_k)$, then the propagation condition of w with respect to its predecessor v_j is defined by the cube:

v_1	v_2	...	v_j	...	v_k	w
C_1	C_2	...	C_j	...	C_k	TRUE

(5)

Where C_i is given by:

$$
\begin{aligned}
C_i &= \text{TRUE}, \quad \text{if} \quad i = j \\
&= \text{FALSE}, \quad \text{if} \quad (i \neq j) \wedge (T(v_i) + d(v_i, w) < D(P)) \\
&= \text{X (unspecified)}, \quad \text{otherwise}
\end{aligned}
$$

Theorem The application of the propagation conditions (4) and (5) yields a valid value for the worst case arrival time $T(w)$.

Proof. $T(w)$ is defined as an upper bound on the latest point of time when w can become TRUE. Thus only the paths that drive w to TRUE need to be considered. Therefore the entry $(w = \text{TRUE})$ is added to the propagation condition in cube (4) and (5).

Furthermore, if w is an AND-vertex, its predecessor v_j can only drive it to TRUE if all the predecessors of w are TRUE, as stated in (1). This explains why the conditions $(v_i = \text{TRUE})$ for $1 \leq i \leq k$ are incorporated in cube (4). Now consider the case that w is an OR-vertex. Let the vertices v_i and v_j be two different predecessors of w. Suppose that the path P passes through v_j and ends in w. It follows from (2) that the predecessor v_j cannot set w to TRUE unless the vertex v_j itself is TRUE.

Suppose now that the predecessor v_i is also TRUE, and that $T(v_i) + d(v_i, w) < D(P)$. Then path P cannot drive w to TRUE at time $D(P)$, because w was already TRUE since $T(v_i) + d(v_i, w)$, according to (2). The path P is bypassed by a faster path. Any signal that is propagating along P will get blocked at w. Consequently, the entry $(v_i = \text{FALSE})$ must be included in the propagation condition of vertex w, if $T(v_i) + d(v_i, w) < D(P)$.

[1] The symbol $\oplus$ denotes the "append" operator on a path.

[2] A "cube" consists of a sequence of vertex values in positional notation.

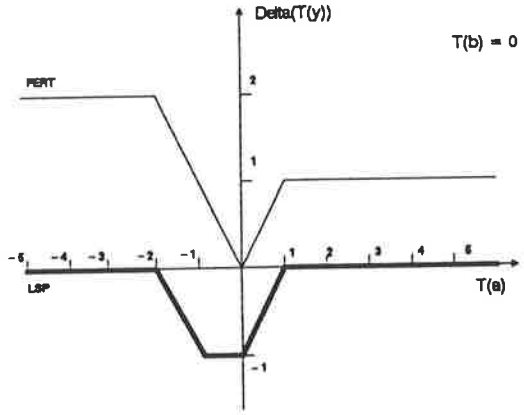

Figure 6: Comparison between existing false path algorithms.

Notice that the propagation condition (5) of an OR-vertex explicitly depends upon the arrival times of its predecessors. Because we take the dynamic behaviour of the circuit into account, our propagation conditions are safe, unlike those proposed in [2] [3] [1]. If a path can be activated in practice, it is never ignored. Therefore, we always obtain an upper bound to the true critical path. Figure 6 illustrates the difference between the new approach and the previous methods of [2] [3] [1], for the circuit of Figure 2. Delta(T(y)) is the deviation from the true worst case arrival time of y for the circuit of Figure 2. Standard longest path algorithms , such as PERT, yield an overestimate. Previous false path algorithms underestimate the delay, as indicated by the LSP-curve. With our method, the exact solution is obtained (same curve as the X-axis).

4.3 The algorithm

An algorithm for computing the longest dynamically sensitizable paths in an event graph is presented below.

First, a special vertex, called the *root*, is connected to the primary inputs of the circuit. The events are ordered by the number of "logic levels" that separate them from the *root*. The *level* of an event is equal to the maximum of the levels of its predecessors plus one. The level of the *root* is zero. We assume that the event graph contains no cycles.

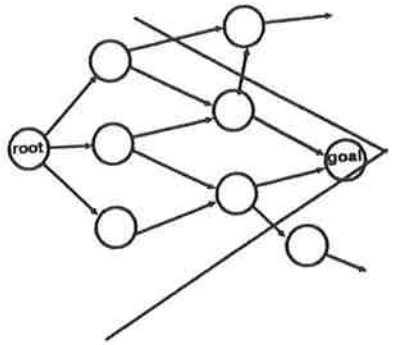

Figure 7: The cone of influence of the *goal* vertex.

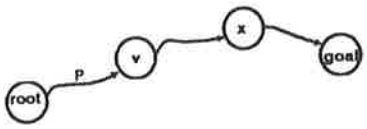

Figure 8: Minmax propagation.

Secondly, events are evaluated in a breadth-first manner from the lowest to the highest level. The evaluation of the worst case arrival time of each event consists of three steps: *back tracing*, *minmax propagation* and *sensitizable path analysis*. The first two steps constitute a preprocessing phase, that is used to speed up the last step. Let *goal* be the event under evaluation.

The back trace procedure consists of a depth-first search from the event *goal* until arriving at the *root*. It isolates the part of the circuit that can affect the arrival time of *goal*. The partial circuit that is reached during the trace is called the *cone of influence* of *goal*. Figure 7 shows an example. The following steps in the algorithm are restricted to the cone of influence.
Furthermore, the back trace procedure computes for each vertex v in the cone of influence the maximum delay $O(v)$ that separates v from *goal*. During the trace, it is assumed that all paths are sensitizable. Hence, the value $O(v)$ constitutes an upper bound on the true delay from v to *goal*.

Minmax propagation is used to propagate information about previously found false paths. Consider the situation in Figure 8. Let P be the path under investigation, and v its last node. Suppose that each path $(v, \ldots, x, \ldots, goal)$ from v to *goal* passes through a vertex $x \neq goal$, and that $D(P \oplus (v, \ldots, x)) > T(x)$. In this case, any path $P \oplus (v, \ldots, x)$ is clearly false. Since vertex x is on a lower level than *goal*, its arrival time has already been evaluated. Consequently, any path to x that yields a larger delay than $T(x)$ must be false. The above rule will be referred to as rule β. In order to apply rule β, we define a value $minmax(v)$ for each vertex v in the cone of influence of *goal*. Let $\mathcal{G}$ denote the set of paths from *root* to *goal*.

- Let:
  $P \in \mathcal{G}$
  v be a vertex on the path P
  Define: $M(v, P)$:
  if $v = goal$: $M(goal, P) = \infty$.
  if $v \neq goal$: $M(v, P) =$
  $min\{T(v), M(w, P) - d(v, w)\}$,
  where w is the successor of v along P.
- Define $minmax(v)$: $minmax(v) = max\{M(v, P) \mid P \in \mathcal{G} \wedge P$ passes through $v\}$

It can be shown that each path P to vertex v, that has a delay larger than $minmax(v)$ is false. The minmax procedure consists of a straightforward depth-first search, starting from *goal* towards the *root*, and generates the $minmax(v)$ value for each v in the cone of influence.

Finally the longest sensitizable path procedure $dsp(root, goal)$ is called. The *dsp*-procedure performs a depth-first search in the cone of influence of *goal*, starting from the *root*. Each time that the search reaches *goal*, the worst case arrival time $T(goal)$ is updated. During the search, $T(goal)$ represents the delay of the longest known sensitizable path to *goal*.

An edge is added to a partial path only if the resulting path remains sensitizable. In order to decide whether a path is sensitizable, the consistency of its propagation conditions must be checked. The consistency operation is based on the D-algorithm [4][5]. The logic implications of the propagation conditions are propagated through the graph, using (1) (2) (3). An inconsistency arises when a vertex is forced to TRUE and FALSE at the same time. The algorithm does not branch into cases. For example, if and AND-node v is set to FALSE, while the value of its predecessors $w_1, w_2, \ldots, w_k$ is still unknown, no value gets propagated towards the inputs. We do not consider each separate case "v_i = FALSE" for $1 \leq i \leq k$. Considering all the possible cases would make our algorithm totally unpractical. Furthermore, it would probably not result in a large reduction of the arrival times, since the incompatibilities that cause important false paths are mostly local.

In order to speed up the *dsp*-procedure, several pruning rules have been incorporated in the algorithm. The efficiency of rule α has already been pointed out by the authors of [2]. It computes the *esperance* of a partial path as the sum of its delay and the maximum delay $O()$ from its last vertex to the *goal*. If the esperance of a partial path is not higher than $T(goal)$, the path can be ignored. The maximum delay $O()$ is also used to indicate the most promising search directions. Rule β has been described above. Rule γ ends the search if no further increase in $T(goal)$ is possible. Rule δ and rule ϵ are not incorporated in the *dsp*-procedure, but are executed before entering it. Rule δ determines which nodes are forced permanently to Vdd or ground, and sets their arrival time to $-\infty$. Finally, the arrival time of a 1-input vertex is determined directly from the arrival time of its predecessor (rule ϵ). More details can be found below.

circuit	# cells	wo	w
24 bit bypass adder	71	91 ns.	52 ns.
8 X 8 RBA multiplier	1050	95 ns.	92 ns.

Table 1: Comparison of the average arrival of the outputs with (w) and without (wo) false path elimination

circuit	exhaustive	+ rule α	+ rule γ	+ rule δ, ϵ	+ rule β
24 bit bypass adder	12268 s.	8476 s.	2907 s.	1927 s.	1364 s.
	-	31%	66 %	34%	29%
8 X 8 RBA multiplier	90963 s.	7837 s.	3917 s.	1234 s.	1192 s.
	-	91%	50%	68%	3.4%
KEY:		x s.	: run time in CPU seconds		
		y %	: pruning gain with respect to the entry at the left.		

Table 2: Run time statistics.

PROCEDURE $dsp(root, goal)$
BEGIN
% Let $g_1, g_2, \ldots, g_l$ be the predecessors of $goal$ %
$upperbound := max\{T(g_i) + d(g_i, goal) \mid 1 \leq i \leq l\}$;
$T(goal) := -\infty$;
% Let P denote the "active" path,
and v the "active" vertex %
$v := root$;
$P := (root)$;
1. Among all the successors w_i of v,
that satisfy the following requirements:
(a) w_i is unexplored
(b) w_i belongs to the cone of influence of $goal$
(c) $P \oplus (v, w_i)$ is a sensitive path
(d) $D(P \oplus (v, w_i)) \leq minmax(w_i)$ % ruleβ %
select the one with the largest $O(w_i)$ and call it u.
If none of the successors satisfies (a)(b)(c)(d),
u is undefined;
2. IF ($u \neq$ undefined) **AND**
$(D(P \oplus (v, u)) + O(u) > T(goal))$ % rule α %
THEN % forward %
mark v as father of u;
mark u as being explored;
erase the explor. marks on all the successors of u;
$P := P \oplus (v, u)$;
$v := u$;
IF ($v = goal$) **AND** ($D(P) > T(v)$)
THEN $T(v) := D(p)$;
IF $T(goal) \geq upperbound$
THEN return; % rule γ %
ELSE % backward %
IF ($v = root$) **THEN** return;
% let f be the father of v %
remove v from P;
$v := f$;
3. GOTO 1.
END

5 Results

The above algorithm has been implemented in C on a VAX 8650 running under VMS. The program has been used to verify several standard cell designs. The results for a 8 X 8 reduced binary adder multiplier and a 24 bit carry bypass adder are presented below. Table 1 shows the average arrival time of the outputs, with and without false path elimination. For the case of the adder, that contains redundant carry bypass circuitry, false path elimination results in a large increase in accuracy. On the multiplier, no significant reduction of arrival time has been obtained with our algorithm.

The processing time statistics in Table 2 show the efficiency of the pruning rules. Each row represents the evolution of the running time, as consecutive pruning techniques are added to the program. The heuristic of rule α works especially well for circuits with few false paths, as can be expected. If no false paths are present, then the back trace procedure is in fact sufficient to find the longest path. Rule β on the other hand, proves useful in the presence of many nonsensitizable paths. The final CPU times in the left column, indicate that the program is fast enough to be of practical use.

6 Summary and Conclusions

Most existing timing verifiers require user input in order to exclude false paths from consideration. Previous attempts to eliminate false paths automatically, yielded unreliable results, because the dynamic behaviour of the circuit was not considered. We have developed accurate conditions for the sensitizability of a path. These conditions explicitly take into account the arrival times of the circuit nodes. An algorithm for computing the longest dynamically sensitizable paths in an acyclic, combinational logic network has been presented. The computation costs are much higher than for standard longest path algorithms, but do not exclude the method from practical use.

References

[1] D. Brand, V. S. Iyengar, "Timing Analysis using Functional Relationships," *Proc. International Conference on Computer-Aided Design*, 1986, pp. 126-129.

[2] J. Benkoski, E. Vanden Meersch, L. Claesen, H. De Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verification," *Proc. International Conference on Computer-Aided Design*, 1987, pp. 44-47.

[3] E. Vanden Meersch, L. Claesen, H. De Man, "SLOCOP, A Timing Verification Tool for Synchronous CMOS Logic," *Proc. ESSCIRC*, 1986, pp. 205-207.

[4] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM J. Res. Develop.*, July 1966, pp. 278-291.

[5] M. A. Breuer, A. D. Friedman, "Diagnosis and Reliable Design of Digital Systems," *Computer Science Press*, 1976, pp. 36-51.