

2

**AD-A231 006**

RADC-TR-90-346  
Final Technical Report  
December 1990



# TEST GENERATION FOR DIGITAL CIRCUITS USING PARALLEL PROCESSING

Syracuse University

Carlos R.P. Hartmann and Akhtar-uz-zaman M. Ali

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*


**DTIC**  
**ELECTE**  
**JAN 11 1991**  
**S B D**

Rome Air Development Center  
Air Force Systems Command  
Griffiss Air Force Base, NY 13441-5700

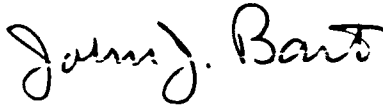
This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-346 has been reviewed and is approved for publication.


APPROVED:

  
WARREN H. DEBANY, JR.  
Project Engineer

APPROVED:

  
JOHN J. BART  
Technical Director  
Directorate of Reliability & Compatibility

FOR THE COMMANDER:

  
BILLY G. OAKS  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (RBRA ) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE December 1990		3. REPORT TYPE AND DATES COVERED Final Jun 89 to Jun 90	
4. TITLE AND SUBTITLE TEST GENERATION FOR DIGITAL CIRCUITS USING PARALLEL PROCESSING				5. FUNDING NUMBERS C - F30602-88-D-0027 PE - 63109F PR - 3003 TA - 00 WU - 01	
6. AUTHOR(S) Carlos R. P. Hartmann, Akhtar-uz-zaman M. Ali					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University School of Computer and Information Science Center for Science and Technology Syracuse NY 13244-4100				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (RBRA) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER  RADC-TR-90-346	
11. SUPPLEMENTARY NOTES RADC Project Engineer: Warren H. Debany, Jr./RBRA/(315) 330-2922					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, distribution unlimited				12b. DISTRIBUTION CODE C	
13. ABSTRACT (Maximum 200 words) The problem of test generation for digital logic circuits is an NP-Hard problem. Recently, the availability of low cost, high performance parallel machines has spurred interest in developing fast parallel algorithms for computer-aided design and test. This report describes a method of applying a 15-valued logic system for digital logic circuit test vector generation in a parallel programming environment. A concept called "fault site testing" allows for test generation, in parallel, that targets more than one fault at a given location. The multi-valued logic system allows results obtained by distinct processors and/or processes to be merged by means of simple set intersections. A machine-independent description is given for the proposed algorithm. ←					
14. SUBJECT TERMS Parallel processors, Multiprocessor, Fault detection, Test generation, Digital logic circuits				15. NUMBER OF PAGES 76	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UL	

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Fault Site Testing</b>	<b>2</b>
<b>3</b>	<b>Deriving Common Requirements for Testing Different Checkpoints</b>	<b>4</b>
<b>4</b>	<b>Parallelism in Sensitizing a Path</b>	<b>5</b>
<b>5</b>	<b>Identifying Independent Subcircuits During Enumeration Phase</b>	<b>9</b>
<b>6</b>	<b>Algorithm Description</b>	<b>13</b>
<b>7</b>	<b>Conclusion</b>	<b>33</b>
<b>A</b>	<b>A 15-Valued Algorithm for Test Pattern Generation</b>	<b>34</b>
A.1	Introduction . . . . .	34
A.2	Pre-processing Phase . . . . .	36
A.2.1	Construction of Dominator Forest . . . . .	36
A.2.2	Selection of pdcf . . . . .	38
A.2.3	Token Assignment . . . . .	39
A.3	Propagation Phase . . . . .	39
A.4	Enumeration Phase . . . . .	42
A.5	Construction of Deterministic Test Cubes . . . . .	44
A.5.1	Forward Implication . . . . .	44
A.5.2	Backward Implication . . . . .	45
A.6	Speed-up Techniques . . . . .	47
A.6.1	Use of the Contrapositive . . . . .	47
A.6.2	Conditional Headlines . . . . .	49
A.6.3	Backward Implication of the Desensitizing Values . . . . .	50

A.7 Examples . . . . . 51

**References** 54

# List of Tables

Table	Page
A1 AND table	35
A2 NOT table	35
A3 XOR table	35
A4 Token assignment for net 3 s - a - 0 in Fig. A1	39
A5 Backward implication for a 2-input AND gate	46
A6 Relationship between 3-VP and 15-VP	48
A7 ( $L_2, G$ ) combinations that yield useful contrapositive assertions	49

<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
<b>Availability Codes</b>	
Dist	Avail and/or Special
A-1	



# List of Figures

Figure		Page
1	Overlapping the testing of several checkpoints	57
2	Conditional basis node with unsatisfiable value	57
3	Example where the subtree corresponding to a single-valued node cannot be satisfied	58
A1	An example circuit	59
A2	Dominator forest for circuit of Fig. A1	60
A3	Introduction of fictitious gate	61
A4	Fictitious gate for net $3\ s - a - 0$ in circuit of Fig. A1	61
A5	Gate decomposition	62
A6	Circuit for Example A2	63
A7	Circuit for Example A3	64

# 1 Introduction

The problem of test generation for combinational circuits is known to be NP-complete [11, 13]. The growing complexity of VLSI circuits has made test generation a more difficult task. Several Automatic Test Pattern Generation (ATPG) algorithms for detecting stuck-at-faults in combinational circuits exist in the literature [5, 8, 10, 12, 14, 17, 18]. In [2] we have proposed a new algorithm based on a 15-valued logic system that introduces some novel approaches to make test generation more efficient. The advent in recent years of low-cost, high-performance parallel machines has spurred further interest in investigating the possibility of developing fast parallel CAD algorithms. In this report we present an approach to efficiently parallelize the ATPG algorithm proposed in [2]. To make this report self-contained a condensed version of [2] is presented in Appendix A.

The task of parallelizing test generation can be approached in various ways. One common approach is to divide the fault set among several processors, a method sometimes referred to as *fault parallelism* [16]. Although communication overhead is low in such an approach, it still provides no improvement over the uniprocessor algorithm with respect to "hard-to-detect" faults or identifying redundant faults.

All test generation algorithms use some heuristic to guide test generation. These usually consist of testability measures in the form of controllability and observability values for all nets of the circuit. Although several such measures exist in the literature, experimental results suggest that no single measure is inherently superior to the others [6]. Consequently, some researchers have suggested the use of *heuristic parallelism* whereby different processors would be used to generate test(s) for the same fault with each of them using a different heuristic to guide the search [7]. The disadvantages of such an approach are that parallelism is limited by the number of useful heuristics available (usually no more than 5) and that no significant improvement is possible if a fault is "hard-to-detect."



Instead of following either of the two approaches discussed above, we concentrate on developing a scheme which exploits the properties of the algorithm presented in [2] to achieve efficient parallelism when generating tests for a fault. Added levels of parallelism can be easily provided by including fault or heuristic parallelism. First, we develop the concept of *fault site testing* in which we utilize the 15-valued logic system in order to derive the common requirements of testing for  $s-a-0$  and  $s-a-1$  faults at the same site (§2). We then present a method whereby testing for different *checkpoint* faults can be efficiently overlapped (§3). Both these speed-up techniques can exploit parallelism during the Enumeration Phase. On the other hand, we can speed-up the Propagation Phase by dividing the work of sensitizing a path, a key feature of [2], among several processors (§4). It is important to note that our 15-valued logic system makes it easy for such a division of tasks because subsequent merging of the results computed by the different processors would involve simple set intersection. In this report we also present a method of identifying several "independent" subcircuits during the Enumeration Phase so that their value justification can be performed in parallel (§5). Finally, we provide a detailed step-by-step description of our proposed parallel ATPG algorithm (§6). To this end we use an algorithm description language that does not cater to any specific existing programming language but uses simple mathematical/logical descriptions of each computational step. This approach was preferred because it provides a much more detailed insight than a flowchart can, without burdening the reader with actual implementational details.

## 2 Fault Site Testing

In this section we discuss how we can exploit the common requirements that are imposed when we sensitize the same path from the fault site to a primary output in order to generate tests for both stuck-at faults at this site. In this report we will use the term *net* to denote the different lines of a circuit; thus the circuit consists of

four different kinds of nets — primary input (PI) nets, primary output (PO) nets, fanout stem (FOS) nets and fanout branch (FOB) nets. In order to perform fault site testing we cannot impose the conditions required to sensitize the fault site until the common requirements are taken into consideration. To do this we introduce a primitive  $d$ -cube of a failure, (*pdf*), different from that in [2], that allows us to take into account both stuck-at faults at a given net:

$$\begin{array}{cc} n & n_f \\ \hline 0/1 & \Delta \end{array}$$

In the *pdf* shown,  $\Delta$  is a variable which contains the information that there is a difference between the normal and faulty circuits without imposing any constraints about the direction of the difference. So  $\Delta = \{(x, \bar{x})\}$  and the corresponding  $\bar{\Delta} = \{(\bar{x}, x)\}$  where  $x \in \{0, 1\}$ . The calculus used in a  $\{0, 1, \Delta, \bar{\Delta}\}$  system is equivalent to that in a  $\{0, 1, D, \bar{D}\}$  system. One way of obtaining this equivalence is to replace  $D$  by  $\Delta$  and  $\bar{D}$  by  $\bar{\Delta}$ . (Another way to do this would be to replace  $D$  by  $\bar{\Delta}$  and  $\bar{D}$  by  $\Delta$ ). Using this system we execute the Propagation Phase of [2] by sensitizing a path  $p_i$ . In the resulting deterministic test cube we set the value of net  $n$  to 0 (1) and find its corresponding deterministic test cube to generate  $T_f(p_i)$  for an  $s$ -a-0 ( $s$ -a-1) fault at net  $n$ . The Enumeration Phase can then be independently executed for both  $T_f(p_i)$ 's in order to generate tests for both the faults.

**Example 1.** We use the same circuit described in Appendix A and shown in Fig. A1 to illustrate the concept of fault site testing. Since net 25 belongs to the set of checkpoint faults for the circuit, we have to generate test for  $s$ -a-0 and  $s$ -a-1 faults at this net. As explained above we start by constructing the *pdf* shown below:

$$\begin{array}{cc} 25 & 25_f \\ \hline 0/1 & \Delta \end{array}$$

Executing the Propagation Phase of [2] with the chosen sensitized path being the one through nets 27, 30, 31, 36, 37, 40 and 45 yields the following deterministic test cube.

4	7	9	10	11	17	18	20	21	25	25;	27	28	29	30
1	0	0	0	0	1	1	0	0	0/1	$\Delta$	$\Delta$	1	0	$\Delta$
31	32	33	34	35	36	37	38	39	40	41	42	43	45	
$\Delta$	0	0	0	0	$\Delta$	$\Delta$	$\Delta$	$\Delta$	$\Delta$	$\Delta$	1	1	1	$\overline{\Delta}$

The value of all nets not indicated above is 0/1. To generate tests for the two stuck-at faults we set the value of net 25, in the above test cube, to 0 and 1 and construct the two corresponding deterministic test cubes. Note that these deterministic test cubes can be constructed independent of each other. The resulting cubes are shown below where only the nets whose values change in the process are indicated.

Stuck-at-0 fault:

24	25	26
1	1	1

Stuck-at-1 fault:

2	5	8	22	23	24	25	26
0	0	1	0	0	0	0	0

In the *s-a-0* case, nets 21 and 24 are the only variant nets whereas nets 19 and 21 are variant for the *s-a-1* situation. The Enumeration Phase for yielding tests using these two test cubes can also be executed independently. □

### 3 Deriving Common Requirements for Testing Different Checkpoints

It is well known that a test set, that detects all single stuck-at faults at the PI nets, FOB nets and the output nets of all XOR/XNOR gates of a circuit, will detect all single stuck-at faults in the circuit [4]. Thus these nets, henceforth referred to as "generalized checkpoints," constitute our initial list of target faults for which tests

have to be generated. However, if any of these target faults is undetectable, additional target faults must be considered [1, 9].

In this section we investigate the possibility of reducing the computation required in testing several checkpoints by first considering their common requirements and performing this computation only once.

Consider a two-input AND gate  $G$ , shown in Fig. 1(a), where both inputs of  $G$  are generalized checkpoints and thus belong to our initial list of target faults. Instead of testing each of the inputs separately, we first impose the constraints that must be satisfied to test the output net of  $G$  as shown in Fig. 1(b). The resulting deterministic cube can then be used to generate tests for the individual faults. Fig. 1(c) shows the additional constraints that must be imposed in order to generate tests for all four faults at the inputs of  $G$ .

The above procedure should be adopted whenever we encounter a gate which has at least two inputs belonging to the set of generalized checkpoints.

## 4 Parallelism in Sensitizing a Path

The Propagation Phase of our algorithm involves the sensitization of a chosen path, say  $p_i$ , from the fault site to a primary output (PO). This sensitization process could be performed on several (say  $k$ ) processors by dividing  $p_i$  into subpaths  $(p_{i1}, p_{i2}, \dots, p_{ik})$ . The division of path  $p_i$  would depend on the availability of processors and the nature of the circuit under consideration. A feasible approach is to divide the path into subpaths such that each processor is allotted the task of performing sensitization between successive FOS nets or between a certain number of FOS nets. In such a scheme a single processor would be used for the Propagation Phase if the circuit under test is a fanout-free one. Furthermore, if we add the provision of indicating node inversion on the dominator forest, then, using the forest, we can determine exactly which of  $\Delta$  or  $\bar{\Delta}$  should be present at the first net of every subpath. If this extra information

is not included, then the test cubes yielded by the different processors must first be compared and accordingly  $\Delta$  must be replaced by  $\overline{\Delta}$  and vice versa in the values of all nets in the test cube yielded by sensitizing the subpath  $p_{i(j+1)}$  if the value of the first net on  $p_{i(j+1)}$  and the last net on  $p_{ij}$  are complementary. The actual sensitization process for a subpath is similar to the Propagation Phase of [2] with the following differences:

1. Instead of sensitizing a path all the way to some PO we now sensitize a subpath  $p_{ij}$ .
2. The list of nets for which forward implication has to be performed will initially contain only the first net of the subpath being sensitized.
3. The list of nets for which backward implication has to be performed will initially contain all the nets, except the first one, that lie on the subpath being sensitized.

(The last two differences need not be implemented if the dominator forest contains information about node inversion.)

Once all the processors have successfully sensitized the subpaths (else an alternate path needs to be chosen) and the  $\Delta$  to  $\overline{\Delta}$  conversion (if necessary) is performed the resulting deterministic test cubes are intersected to yield a new test cube. Two important facts should be noted at this point. The ease with which the computation of several processors is merged by a simple set intersection operation is due to the completeness of the 15-valued logic system. Second, the resultant test cube is not necessarily a deterministic test cube and must consequently be converted into one. If an empty intersection results for any net value or the resulting test cube cannot be converted into a deterministic one, then an alternate path must be chosen. In the situation where we get an empty intersection at a net, it might be useful to investigate how the values of this net in the different test cubes can be used in the selection of an alternate path.

**Example 2.** Once again we will use the circuit shown in Fig. A1 of Appendix A to illustrate the principles of the procedure outlined in this section. Let net 3 be the fault site for which tests have to be generated. Assume that the chosen sensitization path is through nets  $3_f$ , 15, 20, 23, 24, 25, 27, 30, 31, 36, 39, 42, 43 and 45. We divide this into three subpaths as indicated below:

Path  $p_{1,1}$ :  $3_f - 15 - 20 - 23 - 24$

Path  $p_{1,2}$ :  $24 - 25 - 27 - 30 - 31 - 36$

Path  $p_{1,3}$ :  $36 - 39 - 42 - 43 - 45$

Note that this division is based on allotting the path between successive FOS nets to each processor. The initial test cube and resultant deterministic test cubes yielded by the three processors that propagate sensitization along the above subpaths are shown below. Nets which have a value  $0/1$  or  $0/1/\Delta/\bar{\Delta}$  are not indicated in the cubes. Furthermore nets whose value in the deterministic test cube is unchanged from that in the initial test cube are not indicated in the former.

### Processor 1

$(tc_1)_1$ :

$3_f$	15	20	23	24
$\Delta$	$\Delta$	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$

$(d(tc_1))_1$ :

1	2	4	5	12	13	14	16	17	18	19	21
0	0	0	0	0	0	$\Delta$	$\Delta$	0	0	0	$0/\Delta$
22	25	26	27	28	29	30	31	36			
0	$\bar{\Delta}$	$\bar{\Delta}$	$1/\bar{\Delta}$	$1/\bar{\Delta}$	1	$1/\bar{\Delta}$	$1/\bar{\Delta}$	$1/\bar{\Delta}$			
37	38	39	40	41	42	43	45				
$1/\bar{\Delta}$	$1/\bar{\Delta}$	$1/\bar{\Delta}$	$1/\bar{\Delta}$	$0/1/\Delta$	$0/\Delta$	$0/\Delta$	$1/\bar{\Delta}$				

## Processor 2

$(tc_1)_2$ :

24	25	27	30	31	36
$\Delta$	$\Delta$	$\Delta$	$\Delta$	$\Delta$	$\Delta$

$(d(tc_1))_2$ :

7	9	21	26	28	32	33	37	38	39	40	41	42	43
0	0	$0/\Delta$	$\Delta$	$1/\Delta$	0	0	$\Delta$	$\Delta$	$\Delta$	$1/\Delta$	$1/\bar{\Delta}$	$1/\bar{\Delta}$	$1/\bar{\Delta}$

## Processor 3

$(tc_1)_3$ :

36	39	42	43	45
$\Delta$	$\Delta$	$\bar{\Delta}$	$\bar{\Delta}$	$\Delta$

$(d(tc_1))_3$ :

4	10	17	18	29	34	35	37	38	40	41	44
0	1	0	0	1	1	1	$\Delta$	$\Delta$	1	$1/\bar{\Delta}$	1

Since the value of net 24 is  $\bar{\Delta}$  in  $(d(tc_1))_1$  and  $\Delta$  in  $(d(tc_1))_2$ , we complement all  $\Delta$  and  $\bar{\Delta}$  values in the latter. For the same reason the  $\Delta$  and  $\bar{\Delta}$  values of  $(d(tc_1))_3$  also have to be complemented. We then intersect the three test cubes to obtain the following one:

1	2	3 <sub>f</sub>	4	5	7	9	10	12	13	14	15	16	17
0	0	$\Delta$	0	0	0	0	1	0	0	$\Delta$	$\Delta$	$\Delta$	0
18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	$\bar{\Delta}$	0	0	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$	$1/\bar{\Delta}$	1	$\bar{\Delta}$	$\bar{\Delta}$
32	33	34	35	36	37	38	39	40	41	42	43	44	45
0	0	1	1	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$	$\bar{\Delta}$	1	$1/\Delta$	$\Delta$	$\Delta$	1	$\bar{\Delta}$

Note that the above is not a deterministic test cube — converting it to one changes the value of net 6 to 0. The resultant test cube does not have any variant nets and is hence a test for both the stuck-at faults at net 3 can be obtained from it setting the value of this net appropriately.  $\square$

## 5 Identifying Independent Subcircuits During Enumeration Phase

In this section we discuss how we can use the dominator forest to identify “independent” subcircuits whose value justification during the Enumeration Phase can be done in parallel. TOPS [14] introduced the concept of “basis nodes” whereby a net (say  $m$ ) is defined to be a basis node if and only if all FOS nets that influence  $m$  totally reconverge prior to it. Utilizing this property TOPS could postpone the value justification of basis nodes until that of other nodes because they do not interfere with the value justification of nets lying outside its cone of influence. Furthermore, if the circuit does not contain any nets whose value is constant (i.e. independent of the PIs) then the value justification of the basis nodes will not lead to contradictions. Although the use of basis nodes is a generalization of the “headline” concept introduced in FAN [10], it is still a static procedure and does not take into account the constraints imposed by the values of the test cube generated at any stage of the test generation. In [2] we introduced the concept of **conditional headlines** – nets whose value justification could be postponed to the last stage of test generation because they are guaranteed not to cause any contradictions (see §A.6.2). The process of identifying these conditional headlines utilizes both the circuit structure and the values of all the nets in the associated deterministic test cube.

In [2] the discussion of conditional headlines was restricted to nets whose value was either 0 or 1 (see §A.6.2). We now generalize this concept so that nets with any



of the 15 values of our logic system could be investigated for a similar satisfiability property. We will denote a variant net as a **Satisfiable Variant Net (SVN)** if its value justification is guaranteed to succeed and thus can be postponed to the last stage of test generation. In some cases it may be possible to identify nets which are not necessarily SVNs but their value justification depends on a subset of the PIs which do not influence the value justification of some other variant nets. In such a situation the value justification of the two sets of nets in question are independent and could thus be performed in parallel. Thus it would be useful to identify these nets – henceforth denoted as **Independent Variant Nets (IVNs)** – so that their value justification can proceed independently. Note that with every IVN there is associated a subcircuit such that the value justification of the IVN is independent of all nets outside this subcircuit. Hence it is important to identify this subcircuit along with the IVN.

For the remainder of this section we will refer to net as being “single-valued” if the cardinality of the set of values associated with this net, in the deterministic test cube being considered, is unity. Similarly a net will be termed “multi-valued” if the cardinality is greater than unity.

We now present procedures for the identification of SVNs and IVNs using the dominator forest and the values of the circuit nets in the deterministic test cube with respect to which the nets in question are variant.

**(a) SVN Identification:**

(i) Let  $m_v$  be the net under inspection. Consider the subtree  $T$  of the dominator forest that has net  $m_v$  as its root.

(ii) From  $T$  delete all nodes that correspond to FOS nets and also those nodes that are single-valued in the deterministic test cube being considered. Note that the removal of a node implies the removal of the entire subtree which has this node as its root. Let us denote the remaining subtree as  $T'$ .

(iii) If all the leaves of  $T'$  correspond to PI nets then net  $m_v$  is an SVN.

Note that  $T'$  corresponds to a subcircuit (not necessarily "proper") of the largest fanout-free subnetwork that has net  $m_v$  as its output. Moreover the only inputs of this subcircuit at which there is a choice of values are PIs of the overall circuit. Since all values are with respect to a deterministic test cube, the required value at net  $m_v$  can be satisfied and this value justification process will involve assigning values to only the multi-valued PI leaves of  $T'$  and will be independent of all nets that are not in  $T'$ .

**(b) IVN Identification:**

As in the case of SVNs, in order to check whether a net is an IVN we start with the subtree  $T$  of the dominator forest that has net  $m_v$  as its root.

(i) For every node  $m$  of  $T$  which is single-valued in the deterministic test cube in question, consider the subtree  $T_m$  which has net  $m$  as its root.

(ii) If none of the FOB leaves of  $T_m$  are multi-valued then delete the subtree  $T_m$  from  $T$ . Otherwise, consider the FOS nets corresponding to the multi-valued FOB leaves of  $T_m$ . If all these FOS nets are outside  $T$  then delete the subtree  $T_m$  from  $T$ . After all possible deletions let the remaining subtree of  $T$  be denoted as  $T'$ .

(iii) If for every multi-valued FOB leaf of  $T'$ , the corresponding FOS net also belongs to  $T'$  then net  $m_v$  is an IVN.

The value justification of net  $m_v$  can be performed by assigning values to the multi-valued PI leaves of  $T'$  and is independent of all nets that are not in  $T'$ . However unlike the situation for SVNs, this value justification process is not guaranteed to succeed. It is important to note that net  $m_v$  need not be the only variant net in  $T'$ . In such a situation the value justification of all the variant nets in  $T'$  are dependent — however it is independent of all nets lying outside  $T'$ .

As an example of a situation where a net can be ascertained to be a IVN and yet its value in a certain  $d(tc_f(p_i, k))$  cannot be justified, consider the circuit of Fig. 2.

The output of the XOR gate is a IVN with respect to a  $d(tc_f(p_i, k))$  which has the values shown in the figure. However, enumeration will show that this net can only have the value 1 given the values present at the FOB nets shown. The important thing to realize, however, is that even though the value justification of a IVN may not succeed, the justification process is independent of the rest of the circuit.

To understand the procedure for the identification of IVNs note that node  $m_v$  is a basis node if and only if all the FOS nets corresponding to the FOB leaves of  $T$  are contained in  $T$ . In order to generalize the concept of basis nodes we can then relax this condition to allow the single-valued FOB leaves of  $T$  to have their FOS nets outside  $T$ . This is because the value of these nets will not be changed during the value justification of net  $m_v$  and will not affect any nets outside  $T$ . To further weaken the requirement of an IVN we can delete single-valued nodes from  $T$  provided the value justification of net  $m_v$  does not result in an incorrect value at the single-valued node  $m$  that was deleted. The value justification of net  $m_v$  can affect the value of node  $m$  by changing the value of the FOB leaves of the tree  $T_m$  which has node  $m$  as its root. Inspection of the multi-valued FOB leaves of  $T_m$  can result in one of the following three situations:

(i) All the FOS nets corresponding to the multi-valued FOB leaves of  $T_m$  belong to  $T$ .

(ii) All the FOS nets corresponding to the multi-valued FOB leaves of  $T_m$  are outside  $T$ .

(iii) There is at least one multi-valued FOB leaf of  $T_m$  whose FOS net is outside  $T$  and at least one multi-valued FOB leaf of  $T_m$  whose FOS net is in  $T$ .

Situation (i) does not violate the requirement of a basis node but the FOB leaves of  $T_m$  can be affected by the value justification of  $m_v$  and hence  $T_m$  should not be deleted from  $T$ . In situation (ii) the value justification of net  $m_v$  will not affect the value of net  $m$  provided net  $m_v$  is an IVN as per the procedure described earlier.

Hence  $T_m$  can be deleted from  $T$ . We now explain why we cannot delete the subtree  $T_m$  from  $T$  when we have situation (iii). Consider the situation depicted in Fig. 3 where nets  $m_1$  and  $m_2$  are the FOB nets corresponding to the FOS net  $m_{12}$  and nets  $m_3$  and  $m_4$  are the FOB nets corresponding to the FOS net  $m_{34}$ . Let  $m_v$  be the variant net being inspected and let  $m$  be a single-valued node in its tree. Furthermore let nets  $m_{12}$  and  $m_{34}$  be multi-valued in the deterministic test cube being considered. Note that the presence of  $m_1$  and  $m_3$  would prevent us from deleting  $T_m$  from  $T$ . During the value justification of net  $m_v$ , net  $m_{12}$  might be set to a certain value which in turn will assign this new value to nets  $m_1$  and  $m_2$ . This new value of net  $m_1$  might impose certain conditions on the value of net  $m_3$  in order that the required value of net  $m$  be satisfied. Consequently this will affect the value of net  $m_{34}$  and hence value justification of net  $m_v$  will no longer be independent of nets lying outside  $T$ .

The distinction between SVNs and IVNs is of more importance in a sequential implementation of the test generation algorithm because then we can prioritize the value justification of variant nets. In this strategy nets which are neither IVNs nor SVNs will be justified before IVNs which in turn will be justified before SVNs. Thus if any stage results in a contradiction then the subsequent stages need not be performed.

## 6 Algorithm Description

In this section we provide a detailed description of our proposed parallel ATPG algorithm. However, we first discuss the various arrays and data structures that we will be making use of in the actual description.

**Backwardlist:** List of nets for which backward implication needs to be done. If the value of the output of a gate changes we add it to this list so that the corresponding implication is performed.

***Branch\_List:*** For each FOS net this list contains contains the corresponding FOB nets.

***Decisiontree:*** Tree structure to keep track of the decision points that have been tried in terms of the values assigned to the PIs.

***Error:*** Indicates the existence of a contradiction in the test generation process.

***Fault\_List:*** Initial list of target faults (i.e. generalized checkpoints).

***Forwardlist:*** List of nets for which forward implication needs to be done. If the value of any input net of a gate changes we add it to this list so that the corresponding implication is performed.

***Gate\_Predecessor:*** Given a net, its gate\_predecessor is the logic gate for which this net is the output. Note that a FOB net does not have a gate\_predecessor.

***List\_IVN:*** List of Independent Variant Nets.

***List\_SVN:*** List of Satisfiable Variant Nets.

***Maxnet:*** Number of nets in the circuit.

***No\_Test\_Possible:*** Boolean indicating a redundant fault.

***Predecessor:*** Linked list containing predecessor nets of all nets. The inputs to a gate are the predecessors of the output and a FOS net is the predecessor of all its FOB nets.

*Successor*: Linked list containing successor nets of all nets. The output of a gate is the successor of all its input nets and all the FOB nets are the successors of the corresponding FOS net.

*TC*[ $1, \dots, Maxnet$ ]: Test cubes to be used in the test generation process. Entries belong to the 15 valued logic.

*Vnets*: List of Variant nets

In the procedures described in the next few pages we have frequently used terms like dominator forest, node in forest, root etc. which were defined in [2] and are given in Appendix A. Moreover in all the procedures, we have assumed that if a called procedure changes an argument, it changes it for the caller routine also. In our description we have used two kinds of parallelism that should be distinguished. The construct:

```
parbegin
    S1
    S2
    S3
parend
```

has the standard interpretation that statements S1, S2, and S3 can be executed simultaneously or sequentially in any order. On the other hand, the construct used most frequently in this report is:

```
In parallel for (loop control construct) do
begin
    S1
```

S2

S3

end

where the loops are executed simultaneously or sequentially in any order, but within a given execution of the loop statements S1, S2, and S3 are executed sequentially.

We have also used another construct called **Initiate** which initiates a called procedure at the point it is invoked. For every **Initiate** call there is a corresponding **Wait\_for\_completion** where the main routine has to wait till the procedure called by **Initiate** has to finish execution before the statements following the **Wait\_for\_completion** can be executed.

#### Procedure DOMINATOR\_FOREST

/\* This procedure constructs the dominator forest which is then globally accessed (read only) by other subroutines in MAIN \*/

begin1

In Parallel for all nets ( $m$ ) of type *PO* do

TREE ( $m$ )

/\* Procedure TREE ( $m$ ) is performed for every primary output net  $m$  \*/

end1

#### Procedure TREE ( $m_r$ )

/\* This procedure constructs the tree in the forest which has net  $m_r$  as its root \*/

begin1

Create a root node corresponding to net  $m_r$

*Node\_list*  $\leftarrow$  { $m_r$ }

/\* Note that *Node\_list* is local to each parallel processor \*/

```

For all nodes ( $m$ ) in Node_list until Node_list =  $\phi$  do
begin2
  For all  $m_p \in \text{Predecessor}(m)$  do
begin3
   $\text{Child}(m) \leftarrow m_p$ 
  if  $m_p$  is of type FOB then
begin4
  Mark  $m_p$  as a FOB leaf
  Remove  $m_p$  from Branch_list of  $\text{Predecessor}(m_p)$ 
  /* Note that Branch_list is a global variable and all
  the parallel processors have read/write access to it */
  if Branch_list of  $\text{Predecessor}(m_p) = \phi$  then
begin5
  if all nets in  $\text{Successors}(\text{Predecessor}(m_p))$  are in this tree then
begin6
   $m_a \leftarrow$  First common ancestor of  $\text{Successors}(\text{Predecessor}(m_p))$ 
   $\text{Child}(m_a) \leftarrow \text{Predecessor}(m_p)$ 
  Mark  $\text{Predecessor}(m_p)$  as a FOS node
  if  $\text{Predecessor}(m_p)$  is of type PI then mark it as PI leaf
  else add  $\text{Predecessor}(m_p)$  to Node_list
end6
  else if  $\text{Predecessor}(m_p)$  is of type PI then create a single-node
  tree for it and mark it as both a PI and FOS node
  else Initiate1 {TREE ( $\text{Predecessor}(m_p)$ )} on another processor
end5
end4
else

```



```

    begin7
        if  $m_p$  is of type PI then mark  $m_p$  as PI leaf
        else add  $m_p$  to Node_list
    end7
end3
    Remove  $m$  from the Node_list
end2
    Wait_for_completion (Initiate1)
endl

```

#### Procedure RESET\_TC (*TC*)

```

begin1
    For  $i = 1$  to Maxnet
         $TC[i] \leftarrow 0/1$ 
    endl

```

#### Procedure 3VP

/\* This procedure performs all the implications of a 0 and a 1 at FOS nets and stores those whose contrapositive assertions may be useful later \*/

```

begin1
    RESET_TC ( $TC_0$ )
     $TC_1 \leftarrow TC_0$ 
    In Parallel for all nets ( $n$ ) of type FOS do
    begin2
        Forwardlist  $\leftarrow \{n\}$ 
        In Parallel for  $i = 0$  and  $1$  do

```

```

begin3
     $TCi[n] \leftarrow i$ 
    FORWARD ( $TCi, DTCi$ )
    For  $j = 1$  to  $Maxnet$  and  $j \neq n$  do
        begin4
            if  $DTCi[j] = L_2 \neq 0/1$  then
                if  $(DTCi[j], Gate\_Predecessor(j)) \in (L_2, G)$  Table then
                    /* The  $(L_2, G)$  Table is Table A7 of the Appendix and
                    tells us whether this particular implication is worth storing. */
                    Store ( $i$  at net  $n \implies L_2$  at net  $j$ )
            end4
        end3
    end2
end1

```

**Procedure PDCF ( $n, TC_1$ )**

/\* Net  $n$  is the fault site\*/

begin1

Initialize Token for all nets to **False**

RESET\_TC ( $TC_1$ )

$TFLAG[n_f] \leftarrow True$

**Checklist**  $\leftarrow \{n\}$

/\* The following segment identifies all **True** token nets  
and initializes their value to  $c/1/\Delta/\bar{\Delta}$  \*/

For all nets  $m \in Checklist$  until **Checklist** =  $\phi$  do

begin2

```

    In Parallel for all  $m_s \in \text{Successor}(m)$  do
    begin3
         $TFLAG[m_s] \leftarrow \text{True}$ 
        Add  $m_s$  to Checklist
         $TC_1[m_s] \leftarrow 0/1/\Delta/\overline{\Delta}$ 
    end3
    Remove  $m$  from Checklist
end2
 $TC_1[n_f] \leftarrow \Delta$ 
Forwardlist  $\leftarrow \{n_f\}$ 
Backwardlist  $\leftarrow \phi$ 
In Parallel for all  $m_d \in \text{Dominators}(n)$  do
begin4
     $TC_1[m_d] \leftarrow \Delta/\overline{\Delta}$  /* Setting the value of the dominators of  $(n)$  */
    Add  $m_d$  to Forwardlist /* Initial list of nets for which forward and
                                backward implication has to be performed */
    Add  $m_d$  to Backwardlist
end4
end1

```

**Procedure FORWARD** ( $TC, TC'$ )

/\* The calling routine provides  $TC$ , while **FORWARD** returns  $TC'$  to it \*/

begin1

$TC' \leftarrow TC$

For all nets ( $m$ ) in *Forwardlist* until

(*Forwardlist* =  $\phi$ ) or (*Error* = *True*) do

```

begin2
  if m is not of type PO then
    begin3
      if m is of type FOS then
        begin4
          Modify TC' by assigning the value of m to all its FOB nets
          add all the FOB nets of m to Forwardlist
        end4
      else if m is the input of a logic gate then
        begin5
          In TC' forward imply value of net m through gate
          if gate output =  $\phi$  then Error = True
          else if gate output changes then add output net
            to Forwardlist and Backwardlist
        end5
      end3
      Remove m from Forwardlist
    end2
    if Error = True then TC' ← TC
  end1

```

Procedure BACKWARD (*TC, TC'*)

/\* The calling routine provides *TC*, while BACKWARD returns *TC'* to it \*/

```

begin1
  TC' ← TC
  For all nets (m) in Backwardlist until

```

```

(Backwardlist =  $\phi$ ) or (Error = True) do
begin2
  if m is not of type PI then
  begin3
    if m is of type FOB then
    begin4
      In TC' assign the value of m to all nets in
      the set (Successors(Predecessor(m)) - {m})
      add all the nets in this set to Forwardlist and Backwardlist
    end4
    else if m is the output of a logic gate then
    begin5
      In TC' backward imply value of net m through driving gate
      if any input (mi) changes then
        if value of mi =  $\phi$  then Error = True
        else
        begin6
          add mi to Forwardlist and Backwardlist
          forward imply new input values and add m
          to Vnets if appropriate
        end6
      Use 3VP implications to change value of any other net (if
      possible) and add them to Forwardlist and Backwardlist
    end5
  end3
  Remove m from Backwardlist
end2

```

```

    Error = True then TC' ← TC
endl

Procedure DETERMINIZE (TC, DTC)
/* The calling routine provides TC, while DETERMINIZE returns DTC to it */
beginl
    Error ← False
    Repeat
        TC' ← TC
        FORWARD (TC', TC'')
        if Error = False then
            BACKWARD (TC'', TC')
        if Error = False then
            TC ← TC'
        /* Repeated forward and backward implications are performed
        until a deterministic test cube or a contradiction obtained */
    until (Forwardlist = Backwardlist =  $\phi$ ) or (Error = True)
    if Error = False then DTC ← TC
endl

```

```

Procedure PROPAGATE (TC, DTC)
/* This procedure divides the propagation path into subpaths and then computes
the deterministic test cube that considers all the propagation requirements of
the path. This routine is called by providing TC and the resulting cube
is returned as DTC */
beginl

```

Use path heuristic to identify path  $p_i$  and subpaths  $(p_{i1}, p_{i2}, \dots, p_{ik})$  to be sensitized

/\* Notation: First net on subpath  $p_{ij}$  is termed  $m_{ij}$  \*/

**Error**  $\leftarrow$  **False**

In Parallel for  $j = 1, 2, \dots, k$  do

begin2

$TC_{ij} \leftarrow TC$

Use node inversion information from dominator forest to set the value of nets on path  $p_{ij}$  to  $\Delta$  or  $\bar{\Delta}$  or  $\Delta/\bar{\Delta}$  (as appropriate) in  $TC_{ij}$

/\* Note that  $m_{i1}$  is set to  $\Delta$  \*/

**Forwardlist**  $\leftarrow$  {all nets on  $p_{ij}$ } -  $\{m_{i(j+1)}\}$

**Backwardlist**  $\leftarrow$  {all nets on  $p_{ij}$ } -  $\{m_{ij}\}$

/\* The **Forwardlist** and **Backwardlist** are initialized as above in order to avoid the unnecessary duplication of computation that is common to several processors \*/

**DETERMINIZE** ( $TC_{ij}, DTC_{ij}$ )

end2

if **Error** = **True** then

begin3

/\* Sensitizing path  $p_i$  does not yield test \*/

If alternate path available then **PROPAGATE** ( $TC, DTC$ )

else **No\_Test\_Possible**  $\leftarrow$  **True**

end3

else

begin4

$TC \leftarrow \bigcap_{j=1}^k DTC_{ij}$

Set up **Forwardlist** and **Backwardlist** using result of above step

DETERMINIZE ( $TC, DTC$ )

end4

endl

Procedure **FOB\_LEAVES** ( $m$ )

begin1

$List \leftarrow \phi$

From Dominator forest find leaves of subtree that has  $m$  as root

If leaf is of type **FOB** then add it to  $List$

return  $List$

endl

Procedure **PRIORITIZE\_VNETS** ( $DTC$ )

begin1

In Parallel for all nets ( $m_v$ ) in  $Vnets$  do

begin2

Consider the subtree  $T$  of the dominator forest that has  $m_v$  as root

In  $T$  follow multi-valued paths from every child of  $m_v$  towards the

leaves such that no path passes through **FOS** nets

if all paths end at leaves of type **PI** then add  $m_v$  to  $List\_SVN$ ,

delete  $m_v$  from  $Vnets$  and add these **PI** leaves to  $Choicelist(m_v)$

/\*  $List\_SVN$  contains all the Satisfiable Variant Nets

$Choicelist(m_v)$  contains all the **PI** nets to be used in the

value justification of a net  $m_v$  which is either a SVN or an IVN \*/

else

begin3



```

Checklist  $\leftarrow$  {All nodes that are children of  $m_v$  in dominator forest}
For all  $m \in$  Checklist until Checklist =  $\phi$  do
begin4
  if  $m$  is single-valued then
begin5
  add all children of  $m$  to Checklist
  remove  $m$  from Checklist
end5
else
begin6
  List1  $\leftarrow$  FOB_LEAVES ( $m$ )
  In_Stem  $\leftarrow$  False
  Out_Stem  $\leftarrow$  False
  Abort  $\leftarrow$  False
  /* In_Stem and Out_Stem keep track of whether any net
  in List1 has its FOS net inside or outside  $T$  */
  For all  $m_b$  in List1 until (List1 =  $\phi$ ) or (Abort = True) do
begin7
  if  $m_v \in$  Dominator(Predecessor( $m_b$ )) then
    In_Stem  $\leftarrow$  True
  if  $m_v \notin$  Dominator(Predecessor( $m_b$ )) then
    Out_Stem  $\leftarrow$  True
  if In_Stem = Out_Stem = True then Abort  $\leftarrow$  True
  else remove  $m_b$  from List1
end7
  if In_Stem = False and Out_Stem = True then add  $m$ 
  to Delete_Nodes

```

```

        end6
    end4
    if Abort = False then
    begin8
        From T construct another tree T' by deleting all the
            nodes (and their subtrees) that belong to Delete_Nodes
        List2 ← {All the multi-valued FOB leaves of T'}
        IVN ← True
        For all mi in List2 until (List2 =  $\phi$ ) or (IVN = False) do
        begin9
            if Predecessor(mi) belongs to tree T' then remove mi from List2
            else IVN ← False
        end9
        if IVN = True then add mv to List_IVN, delete mv from Vnets
            and add all multi-valued PI leaves of T' to Choicelist(mv)
            /* List_IVN contains all the Independent Variant Nets */
            else Choicelist(mv) ←  $\phi$ 
        end8
        else Choicelist(mv) ←  $\phi$ 
    end3
end2
end1

```

**Procedure ENUMERATION (*DTC*)**

```

begin1
    TC ← DTC

```

```

For all nets ( $m_v$ ) in  $Vnets$  until ( $Vnets = \phi$ ) or ( $Decisiontree = \phi$ ) do
  repeat
  begin2
    Use controllability measures to identify a candidate  $PI(m_i)$  from  $m_v$ 
     $Forwardlist \leftarrow \{m_i\}$ 
     $TC[m_i] \leftarrow Value$ 
    /* Assign a value to a  $PI$  for value justification of  $m_v$  */
    FORWARD ( $TC, TC'$ )
    Store  $PI$  and  $Value$  in  $Decisiontree$ 
     $TC \leftarrow TC'$ 
  end2
  until (objective at  $m_v$  met) or ( $Decisiontree = \phi$ )
if  $Decisiontree = \phi$  then
begin3
   $No\_Test\_Possible$ 
  Exit ENUMERATION
end3
else
begin4
   $Backwardlist \leftarrow \phi$ 
  In Parallel (indexed by  $i$ ) for all nets ( $m_{ivn}$ ) in  $List\_IVN$  do
  begin5
    repeat
      Use controllability measures to choose a net ( $m$ ) from  $Choicelist(m_{ivn})$ 
       $Forwardlist \leftarrow \{m\}$ 
       $TC_i[m] \leftarrow Value$ 
      FORWARD ( $TC_i, TC'_i$ )
    repeat

```

```

    if  $m$  is a PI then store PI and Value in Decisiontree
     $TC_i \leftarrow TC'_i$ 
    Remove  $m$  from Choicelist( $m_{i,vn}$ )
    if  $m$  is a FOS net then add  $m$  to Backwardlist
until (Choicelist( $m_{i,vn}$ ) =  $\phi$ ) or (Decisiontree =  $\phi$ ) or (objective at  $m_{i,vn}$  met)
end5
if (Choicelist( $m_{i,vn}$ ) =  $\phi$ ) or (Decisiontree =  $\phi$ ) then
begin6
    No_Test_Possible
    Exit ENUMERATION
end6
else
begin7
     $TC \leftarrow \cup_i TC_i$ 
    DETERMINIZE (TC, DTC)
    if Vnets  $\neq \phi$  then
begin8
        ENUMERATION (DTC)
         $TC \leftarrow DTC$ 
end8
    In Parallel (indexed by  $i$ ) for all nets ( $m_{svn}$ ) in List_SVN do
begin9
    repeat
        Use controllability measures to choose a PI from Choicelist( $m_{svn}$ )
        Forwardlist  $\leftarrow PI$ 
         $TC_i[PI] \leftarrow Value$ 
        FORWARD ( $TC_i$ ,  $TC'_i$ )

```

```

         $TC_i \leftarrow TC'_i$ 
        Store PI and Value in Decisiontree
        until objective at  $m_{s,vn}$  met
    end9
     $TC \leftarrow \cup_i TC_i$ 
    DETERMINIZE (TC, DTC)
end7
end4
end

```

#### Procedure MAIN

```

begin1
    READ_DATA
    Construct Fault_List, Predecessor, Successor
    parbegin2
        DOMINATOR_FOREST
        3VP
    parend2
    For all n in Fault_List until Fault_List =  $\phi$  do
    begin3
        if n drives gate G such that any other input(s) of G are in Fault_List then
        begin4
            Remove these input nets from Fault_List and add them to Faultset
            /* Faultset contains the different checkpoints to be tested */
            PDCF ( $m_g$ , TC) /*  $m_g$  is the output of G */
        end4
    end3
end

```

```

else PDCF ( $n, TC$ ) /* In this case  $Faultset = \phi$  */
Remove  $n$  from  $Fault\_List$ 
PROPAGATE ( $TC, DTC$ )
if  $No\_Test\_Possible = False$  then
begin5
    PRIORITIZE_VNETS ( $DTC$ )
    Using  $Faultset$  and  $G$  create list  $L_c$  of ordered pairs (net  $m_c$ , value  $l_c$ )
        corresponding to all checkpoint faults that can be enumerated in parallel
    /* If  $Faultset = \phi$  then  $L_c = \{(n,0), (n,1)\}$  */
    In Parallel for all  $(m_c, l_c) \in L_c$  do
begin6
     $TC \leftarrow DTC$ 
     $TC[m_c] \leftarrow l_c$ 
    DETERMINIZE ( $TC, DTC'$ )
    ENUMERATE ( $DTC'$ )
    if  $No\_Test\_Possible = False$  then return ( $DTC', Test$ )
    else ADDITIONAL_FAULTS ( $m_c, l_c$ )
    /* If any of the generalized checkpoint faults are determined to be redundant
        then the above step will determine the additional faults to be tested */
end6
end5
else
begin7
    ADDITIONAL_FAULTS( $m_{pdcf}, 0$ )
    ADDITIONAL_FAULTS( $m_{pdcf}, 1$ )
    /* Net  $m_{pdcf}$  is the net used in the construction
        of the PDCF i.e. either net  $n$  or net  $m_g$  */

```

end7

end3

end1

## 7 Conclusion

In this report we have presented some new techniques for efficiently parallelizing the 15-Valued Test Pattern Generation algorithm introduced in [2]. This was accomplished by dividing the path to be sensitized into several subpaths and using a separate processor to perform the sensitization of each subpath. This is possible because of the strength of the 15-valued logic system used. To make the algorithm even more efficient we overlap the testing of several checkpoints and introduce the concept of fault-site testing where the propagation phase for testing both the stuck-at faults at any net is executed simultaneously. We have also presented a procedure to identify "independent" subcircuits whose value justification during the Enumeration Phase can be performed independently. It is important to note that in each of the parallelization techniques proposed the communication overhead is low because it involves only the intersection of test cubes whose entries are elements of the logic system used. The analysis of the different ideas introduced in this report suggest that an implementation in an MIMD environment can prove to be a significant improvement in the area of testing.



# Appendix

## A A 15-Valued Algorithm for Test Pattern Generation

### A.1 Introduction

In this appendix we present an ATPG algorithm, for detecting single stuck-at-faults in combinational circuits that contain NOT, AND, NAND, OR, NOR, XOR and XNOR gates. This algorithm is based on a 15-valued logic system and introduces some novel approaches to make test pattern generation more efficient.

Test generation involves considering the value of a net in the good and the faulty circuit. This can be done by representing the value of a net as an ordered pair  $(b_g, b_f)$  where  $b_g(b_f)$  is the value of the net in the good (faulty) circuit [15]. Thus the value of a net can be one of the elements of the set  $U = \{(0,0), (0,1), (1,0), (1,1)\}$ . In the process of generating tests it might not be possible to uniquely specify the value of a net as one of the elements of  $U$ . However, we may already know that a net cannot assume one or more of these values. We incorporate this information by defining the value of a net as one of the 15 nonempty subsets of  $U$ . We denote these 15 sets as  $0, 1, D, \bar{D}, 0/1, 0/D, 1/D, 0/\bar{D}, 1/\bar{D}, D/\bar{D}, 0/1/D, 0/1/\bar{D}, 0/D/\bar{D}, 1/D/\bar{D}$ , and  $0/1/D/\bar{D}$  where  $0 = \{(0,0)\}$ ,  $1 = \{(1,1)\}$ ,  $D = \{(1,0)\}$ ,  $\bar{D} = \{(0,1)\}$  and “/” denotes set union. Note that  $U = 0/1/D/\bar{D}$ . These 15 values are equivalent to the elements of the logic system developed by Akers [3] to provide a tool for test generation. Tables A1, A2 and A3 represent the AND, NOT, and XOR functions in our 15-valued system for the values  $0, 1, D$ , and  $\bar{D}$ . The complete table for all 15 values can be easily constructed from the given tables by using the set union operation. The tables for all other logic functions can be obtained from these three tables.

AND	0	1	$D$	$\bar{D}$
0	0	0	0	0
1	0	1	$D$	$\bar{D}$
$D$	0	$D$	$D$	0
$\bar{D}$	0	$\bar{D}$	0	$\bar{D}$

Table A1. AND table

Variable	0	1	$D$	$\bar{D}$
Complement	1	0	$\bar{D}$	$D$

Table A2. NOT table

XOR	0	1	$D$	$\bar{D}$
0	0	1	$D$	$\bar{D}$
1	1	0	$\bar{D}$	$D$
$D$	$D$	$\bar{D}$	0	1
$\bar{D}$	$\bar{D}$	$D$	1	0

Table A3. XOR

table

Using this notation we will define a sensitized net as one whose value is either  $D$ ,  $\bar{D}$ , or  $D/\bar{D}$ . Furthermore, if all the nets along a path in the circuit are sensitized, then the path is said to be sensitized. As will be seen later on, this 15-valued system exploits the linearity of XOR/XNOR gates during test generation. It also allows us to characterize all restrictions that are imposed by a fault and the particular circuit path chosen in order to propagate its effect.

There are three distinct phases in the algorithm presented here:

(i) Pre-processing phase (§A.2). In this phase we construct a set of trees based on the interdependence of circuit nets. Among other things this forest will be used to easily identify which circuit nets *must* be sensitized to derive a test.

(ii) Propagation phase (§A.3). In this phase we deliberately sensitize a single path from the fault site to a PO and find all the resulting deterministic forward and backward implications. In the process other paths may get sensitized. Path selection is the only choice made in this phase—implications are based on all the constraints that *must* be satisfied in order to sensitize the chosen path. This is possible because of the completeness of the 15-valued system and the use of deterministic implication rules.

(iii) Enumeration phase (§A.4). In general, the test cube constructed by the Propagation Phase will not yield a test—particularly because no arbitrary choices were made. Thus there may be gates whose input net values contain combinations

capable of desensitizing the chosen path. In this phase we use an enumeration procedure to choose values for the PIs so that such combinations can never occur.

To illustrate the above phases of our algorithm we will consider the fault net  $3s - a - 0$  in the circuit of Fig. A1.

In order to make the last two phases more efficient we have developed some speed-up techniques (§A.6). One is the extension of the contrapositive procedure presented in SOCRATES [18] for backward implying 0 and 1 values. However, our procedure not only generates the contrapositive assertions for all 15 values of our system, but also requires less computation and storage than SOCRATES. We will also present a procedure that not only takes into account the circuit structure but also the constraints imposed by the values of a test cube in order to identify nets whose value justification can be postponed until the end. Furthermore, we will show how backward implication of the values that desensitize the chosen path can help in the selection of PI values during the Enumeration Phase.

## A.2 Pre-processing Phase

### A.2.1 Construction of Dominator Forest

The importance of identifying nets that *must* be sensitized for a fault to be detected was first highlighted by Akers [3] and later by Fujiwara and Shimono [10]. As pointed out in TOPS [14], the concept of graph dominators [19] can be used to identify the nets which *must* be sensitized to detect a fault. In the context of test generation we term the set of dominators of a net  $m$  as the set of all nets in the circuit which lie on every path from net  $m$  to any PO. By definition, net  $m$  is a dominator of itself; however, for ease of notation we define  $D(m)$  as the set of all dominators of  $m$  except  $m$  itself. To account for multiple output circuits the concept of dominator tree can be extended to that of a forest. We present here a procedure to construct this forest for a given circuit. This forest will not only be used to compute the dominators for a

particular fault site; but also for the sensitization of subpaths, selection of PIs in the enumeration phase and generation of the initial list of target faults.

We construct a set of trees such that every net of the circuit corresponds to a node in one of the trees in the forest. We start by creating as many trees as there are POs such that each PO corresponds to a root of a tree. However, new trees may be created during the procedure. Thereafter, each node which has not been marked as a leaf is inspected and the tree construction is continued as follows:

(i) If the node  $m_i$  being considered corresponds to the output net of a logic gate  $G_i$  in the circuit, then every input net of  $G_i$  becomes a child of this node  $m_i$ . Furthermore, if the input net is a PI it is marked as a PI leaf. If the input net is a FOB, then it is marked as a FOB leaf.

(ii) If the node  $m_i$  being inspected is a fanout stem (FOS), then wait until all the FOBs corresponding to this FOS have been marked as FOB leaves. Then find the immediate ancestor of all these FOB leaves. If such an ancestor exists, then make  $m_i$  a child of this ancestor node. If it does not, then start a new tree with  $m_i$  as a root. In either case, mark  $m_i$  as an FOS node—if it is also a PI, then it must be marked as a PI leaf also.

The above procedure is continued until every net of the circuit becomes a node in some tree of the forest.

Note that the leaves of the trees in this forest correspond to the *checkpoints*, i.e., the PIs and the FOBs. Thus our initial list of target faults consists of all leaves of the trees of the dominator forest and the output of all XOR/XNOR gates [4]. However, in case any of these target faults are undetectable additional target faults must be considered [1, 9].

The root of any tree in the constructed forest is either a PO or a FOS. If any tree has a single node, then this node must correspond to a PI which is also a FOS. The set  $D(m)$  contains all the nodes encountered when traversing the tree (in which  $m$  is

a node) from  $m$  to the root.

The "basis nodes," as defined in TOPS [14], can also be identified easily from the dominator forest. However, keeping in mind that a node cannot be a basis node unless all FOS nets that influence it have completely reconverged prior to it, we adopt a simpler approach of identifying which nodes are NOT basis nodes. Thus, instead of inspecting each node to verify whether it is a basis node or not, we pick one FOS net at a time to generate the set of nodes which are NOT basis nodes. Let there be  $k$  FOS nets denoted by  $m_i, i = 1, 2, \dots, k$ . Furthermore, let the FOS net  $m_i$  have  $n_i$  FOB nets denoted by  $m_{i1}, m_{i2}, \dots, m_{in_i}$ . It can be shown [2] that the set of nodes which are NOT basis nodes is given by

$$\bigcup_{i=1}^k \left[ \bigcup_{j=1}^{n_i} [D(m_{ij}) \cup \{m_{ij}\}] - D(m_i) \right].$$

Consequently, all nodes not belonging to the above set are basis nodes.

The dominator forest for the circuit in Fig. A1 is shown in Fig. A2. Note that the only basis nodes for this circuit are the PIs.

### A.2.2 Selection of *pdcf*

The selection of the primitive  $D$ -cube of the failure (*pdcf*) in DALG [17] may involve arbitrary choices which can result in mistaken decisions causing costly backtracking. We avoid this problem by introducing a fictitious gate  $G_f$  at the site of the fault. If the fault is at net  $n$  we introduce  $G_f$  between net  $n$  and a newly created net  $n_f$  as shown in Fig. A3. We now connect  $n_f$  to all nets which were previously connected to  $n$ . Accordingly, the unique *pdcf* depends only on the kind of stuck-at fault.

		$n$	$n_f$
$n$	$s-a-0$	1	$D$
$n$	$s-a-1$	0	$\overline{D}$

Thus in our example we will modify Fig. A1 to include the gate shown in Fig. A4.

### A.2.3 Token Assignment

The goal of this stage is to identify which circuit nets can or cannot be affected by the fault. In order to convey this information we associate with every net a Boolean token. This token will be TRUE if and only if there exists a path from  $n_f$  to any PO which passes through this net. These tokens can be computed by a single forward pass through the circuit. Table A4 shows the Boolean token assignment for our example.

Nets with TRUE Token	3 <sub>f</sub> , 14, 15, 16, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 36, 37, 38, 39, 40, 41, 42, 43, 45
Nets with FALSE Token	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 17, 18, 29, 32, 33, 34, 35, 44

Table A4. Token assignment for net 3  $s - a - 0$  in Fig. A1

### A.3 Propagation Phase

In this phase we will sensitize a single path from net  $n_f$  to a PO, however, other paths may also get sensitized. In a manner analogous to DALG [17] we will use test cubes whose entries reflect the current values of all nets during any stage of test generation. The entries of any test cube,  $tc_k$ , are elements of our 15-valued system.

We initialize this phase by constructing  $tc_1$  in the following manner:

1. Set nets  $n$  and  $n_f$  to the values specified by the *pdf*.
2. Assign  $D/\overline{D}$  to all nets belonging to the set  $D(n)$ .
3. Set all nets with FALSE tokens, except net  $n$ , to 0/1.
4. Assign  $0/1/D/\overline{D}$  to all unassigned nets of the test cube.

In our example  $D(3) = \{31, 36, 45\}$ , and the resulting  $tc_1$  is given below where only nets whose entries are different from 0/1 and  $0/1/D/\overline{D}$  are shown.

3	$3_f$	31	36	45
1	$D$	$D/\overline{D}$	$D/\overline{D}$	$D/\overline{D}$

For each test cube  $tc_k$  generated at any stage of our algorithm we find its corresponding "deterministic" test cube,  $d(tc_k)$ . We define a  $d(tc_k)$  as one in which no entry can be changed without making some arbitrary choice(s) in one or more net values. That is, all unique implications of the net values must be considered. Rules for forward and backward implication procedures to be used in constructing  $d(tc_k)$  from  $tc_k$  are given in §A.5. If in any  $d(tc_j)$  we have a sensitized path  $p_i$  from the fault site to any PO, then the Enumeration Phase is invoked. This test cube,  $d(tc_j)$ , is denoted as  $T_f(p_i)$ . The  $d(tc_1)$  for our example is shown below. Only the entries for nets whose values are different from those in  $tc_1$  are listed. In fact, for each cube that we construct only the entries whose values are different from those in the preceding one will be explicitly shown.

9	14	15	16	19	20	21	22	23	30
0	$D$	$D$	$D$	$0/D$	$0/\overline{D}$	$0/D$	$0/1/D$	$0/1/\overline{D}$	$0/D/\overline{D}$
32	33	37	38	39	40	41	42	43	
0	0	$D/\overline{D}$	$D/\overline{D}$	$D/\overline{D}$	$1/D/\overline{D}$	$1/D/\overline{D}$	$1/D/\overline{D}$	$1/D/\overline{D}$	$1/D/\overline{D}$

If  $d(tc_1)$  cannot be constructed because contradictions were encountered, then there exists no test for the fault. Otherwise we have a sensitized path from  $n_f$  to all the FOB nets corresponding to the first FOS node (could be  $n$  itself!) encountered in traversing the appropriate tree of the dominator forest from  $n$  to the root. If there is no FOS encountered, then we have a sensitized path from  $n_f$  to the PO corresponding to the root of the tree. In our example, since net 3 is an FOS we have sensitized paths only until its FOB nets, i.e., 14, 15, and 16.

At this point we have to select one of the FOB nets, say  $m_1$ , to extend the sensitized path. To obtain  $tc_2$  we should sensitize all nets belonging to the set  $D(m_1) - D(n)$  by intersecting their values in  $d(tc_1)$  with  $D/\overline{D}$ . If any empty intersection

results, then the sensitized path cannot be extended through  $m_1$  and alternate paths should be investigated. Note that this step is implicitly performing the equivalent of the X-path check [12] while setting up which gate outputs should be sensitized. As stated earlier, we would then construct  $d(tc_2)$ . If contradictions occur while constructing  $d(tc_2)$ , then an alternate path must be selected. Otherwise we have a sensitized path from  $n_f$  to at least the FCB nets corresponding to the next FOS net or some PO. Assume that we extend the sensitized path in our example through net 16. We use  $D(16) - D(3) = \{21\}$  so that net 21 has the value  $D$  in  $tc_2$ . In the resulting  $d(tc_2)$  shown below we have sensitized paths till the FOB nets 37, 38 and 39.

$$\begin{array}{cccccccccccc} 6 & 30 & 31 & 36 & 37 & 38 & 39 & 40 & 41 & 42 & 43 \\ \hline 1 & 0/D & D & D & D & D & D & 1/D & 1/\overline{D} & 1/\overline{D} & 1/\overline{D} \end{array}$$

The process of extending the sensitized path by selecting a FOB net, constructing a  $tc_k$  and its corresponding  $d(tc_k)$  is continued until we reach some PO and have constructed  $T_f(p_i)$ . If contradictions occur, then alternate paths should be investigated. If all possible paths give contradiction, then no test exists. Note that all possible single paths need not be explicitly investigated to arrive at this conclusion. Proceeding with our example, let us extend the sensitized path through net 39. Since  $D(39) - D(36) = \{42, 43\}$ , the  $tc_3$  shown below results.

$$\begin{array}{cc} 42 & 43 \\ \hline \overline{D} & \overline{D} \end{array}$$

However, the attempt to construct  $d(tc_3)$  fails as shown below.

Steps in  $d(tc_3)$  construction:

$$\begin{array}{l} (i) \frac{40}{1} \longrightarrow \frac{35 \ 10 \ 34}{1 \ 1 \ 1} \\ (ii) \frac{29}{1} \longrightarrow \frac{4 \ 17 \ 18}{0 \ 0 \ 0} \longrightarrow \frac{20}{\overline{D}} \longrightarrow \frac{23 \ 24}{1/\overline{D} \ 1/\overline{D}} \longrightarrow \\ \longrightarrow \frac{25 \ 26 \ 27 \ 28}{1/\overline{D} \ 1/\overline{D} \ 1/\overline{D} \ 1/\overline{D}} \longrightarrow \frac{30}{(1/\overline{D}) \cap (0/D) = \emptyset} \quad (\text{Contradiction}) \end{array}$$



Thus we go back to  $d(tc_2)$  and choose another path—say through net 37. The resulting  $tc_4$  sets the value of net 40 to  $D$  and the  $d(tc_4)$  constructed from it is shown below:

4	10	11	17	18	20	23	24	25		
1	0	0	1	1	0	0/1	0/1/D	0/1/D		
26	27	28	29	34	35	41	42	43	45	
0/1/D	0/1/D	0/1/D	0	0	0	1	1	1	D	

We now have a sensitized path (say  $p_1$ ) from  $3_f$  to a PO, and thus  $d(tc_4)$  is  $T_f(p_1)$ .

Note that  $T_f(p_i)$  represents all the constraints that *must* be imposed to sensitize path  $p_i$ . Since the backward implication rule does not make any arbitrary choices, there may be gates where the output value is a proper subset of the value implied by the input values, i.e., the input values include combination(s) that will desensitize path  $p_i$ . We define the output nets of such gates as **variant nets**. If a net is not variant it is defined to be **invariant**. In our example the only variant net w.r.t.  $T_f(p_1)$  is net 30.

If there are no variant nets in  $T_f(p_i)$ , then we have already obtained a test for the fault. Otherwise the Enumeration Phase must be invoked to determine a test.

## A.4 Enumeration Phase

The goal of this phase is to obtain a test by specifying the unassigned PIs in  $T_f(p_i)$  such that all nets are invariant and have values that are subsets of their corresponding values in  $T_f(p_i)$ .

We choose an unspecified PI  $I_{i_1}$  in  $T_f(p_i)$  and assign a logic value (0 or 1) to it, thereby creating a new test cube which we denote by  $tc_f(p_i, 1)$ . Now we find its corresponding deterministic test cube  $d(tc_f(p_i, 1))$  and update its list of variant nets (note that new variant nets may be created). However if  $d(tc_f(p_i, 1))$  cannot be obtained due to some contradiction, then we complement the entry for  $I_{i_1}$  in  $tc_f(p_i, 1)$

and construct its corresponding  $d(tc_f(p_i, 1))$ . If this also leads to a contradiction, then there exists no test corresponding to  $T_f(p_i)$ . If we are successful in constructing  $d(tc_f(p_i, 1))$  we now assign a logic value to some other unspecified PI  $I_{i_2}$ , thereby creating  $tc_f(p_i, 2)$ . As before we must construct  $d(tc_f(p_i, 2))$  and update its list of variant nets. This procedure is continued and we traverse the decision tree, in a manner analogous to PODEM [12], until one of the following two conditions occur:

- The list of variant nets corresponding to some  $d(tc_f(p_i, j))$  becomes empty. This indicates the values of the PIs in  $d(tc_f(p_i, j))$  represent test(s) for the fault.
- The decision tree is exhausted, i.e. no test exists.

For sake of completeness we denote  $T_f(p_i)$  as  $d(tc_f(p_i, 0))$ .

We now continue with our example for the fault net 3  $s-a-0$  in the circuit of Fig. A1. As stated earlier, net 30 is the only variant net w.r.t.  $T_f(p_1)$ . By inspecting the dominator forest we notice that nets 7 and 8 are the PIs which are "closest" to net 30. We thus start by setting net 7 to 0—however, this does not change the value of any other net. We continue by setting net 8 to 0—once again no new changes result. We now use the dominator forest to reach the FOS net 24 and thus determine that nets 2 and 5 are the next "closest" PIs. We could, for example, set net 2 to 0—the only resulting change is a  $0/D$  at net 22. Net 30 is still the only variant net, so we now set net 5 to 0. This changes the value of net 23 to 0 and that of nets 24, 25, 26, 27, and 28 to  $0/D$ . Also, all nets are verified to be invariant, thus a test has been generated.

The algorithm described so far can be substantially improved by the introduction of several speed-up techniques which we discuss in §A.6.

## A.5 Construction of Deterministic Test Cubes

In a  $d(tc_k)$  all deterministic implications (no arbitrary choice) of all entries of the test cube  $tc_k$  are fully considered. To construct  $d(tc_1)$  from  $tc_1$ , we perform backward and forward implications of all nets whose values in  $tc_1$  are different from 0/1 and 0/1/D/ $\overline{D}$  and all other nets whose values change during this implication process. In the general case, when we are constructing  $d(tc_k)$  from  $tc_k$ , we start by considering the forward and backward implications of the nets whose values in  $tc_k$  are different from those in the last successfully constructed deterministic test cube and that of all other nets whose values change during this implication process. During the construction of  $d(tc_k)$  from  $tc_k$ , if a backward or forward implication request results in a new value  $L'_j$  for any net  $m_j$  of the circuit, then we should update the corresponding net entry  $L_j$  by setting it to  $L_j \cap L'_j$ . If this intersection yields the empty set then  $d(tc_k)$  cannot be constructed.

In order to obtain  $d(tc_k)$  the process of forward and backward implications should be continued until no more changes occur in the values associated with any net. Note that this process will terminate in a finite number of steps because we are performing set intersection on finite sets.

The rules for constructing deterministic test cubes must include the provision for appropriately handling the values of nets associated with fanout points and should also take into account the information provided by the token vectors.

### A.5.1 Forward Implication

The process of forward implications of the values associated with every net is done with the help of Tables A1, A2 and A3. These tables are a generalization of the truth tables of the respective gates. For gates with more than two inputs the method adopted is similar to that used by Akers [3]. We view every gate as being constructed out of 2 input gates and use the existing values at the inputs of a gate to generate a

new value for the output. Depending on the gate in question, appropriate tables are used.

Suppose we are performing forward implications due to change(s) in input(s) of a gate  $G$  whose output is net  $m$ . Let  $L_O$  be the set of values associated with net  $m$  in the test cube prior to forward implication being performed. Also let  $L_N$  be the value obtained at net  $m$  by using the new values of the inputs of  $G$ . Net  $m$  will then be set to  $L_O \cap L_N$  unless  $L_O \cap L_N = \emptyset$  which implies a contradiction. Four other situations are possible:

1.  $L_O = L_N$ . No further action is needed for this forward implication.
2.  $L_N \subset L_O$  (proper subset). We now have to consider the forward implication of the value of  $L_N$  at net  $m$  on all gates driven by  $G$ .
3.  $L_O \subset L_N$ . We now have to perform a backward implication of the value  $L_O$  at net  $m$ . This may result in further changes in the inputs of gate  $G$ .
4.  $L_O \not\subset L_N$  and  $L_N \not\subset L_O$ . Both forward and backward implications should be performed.

### A.5.2 Backward Implication

The process of backward implication involves determining the changes required at the inputs of a gate in order to satisfy a requested change at the output. A change in the value of a net will mean that one or more possible values associated with the net has been deleted. In that sense an input change can be made only if the deleted value can never be used with the existing values at the other inputs to generate any of the requested output value(s).

The backward implications rules for a two-input AND gate is shown in Table A5. Note that the element  $\emptyset$  has been included in this table to detect an unsatisfiable backward implication request.

* **	0	1	$D$	$\bar{D}$
0	0/1/ $D/\bar{D}$	$\emptyset$	$\emptyset$	$\emptyset$
1	0	1	$D$	$\bar{D}$
$D$	0/ $\bar{D}$	$\emptyset$	1/ $D$	$\emptyset$
$\bar{D}$	0/ $D$	$\emptyset$	$\emptyset$	1/ $\bar{D}$

\* Requested Output

\*\* Existing value at one input

Table A5. Backward implication for a 2-input AND gate

The complete table for all 15 values is obtained by the set union operation. The resulting table is equivalent to that proposed by Akers [3]. To perform backward implication for a two-input AND gate we reference the table using the requested value at the output and the existing value at one input to generate the value of the other input. Since the XOR gate is linear, Table A3 can be used for backward implication also. Thus Tables A2, A3 and A5 can be used to perform backward implication for any two-input gate. Irrespective of the gate in question, the value generated by the appropriate table must be intersected with the existing value of the input to generate the new value of the input. Analogously, the new value of the input and the requested value of the output must now be used to generate the new value of the other input.

As before, any gate with more than two inputs will be represented as a cascade of two-input gates. Consider an  $n$ -input gate  $G$  represented as a cascade of  $(n - 1)$  two-input gates  $G_1, G_2, \dots, G_{n-2}$  and  $G_{n-1}$ , with net numbers as shown in Fig. A5. Assume that the values at nets  $1, 2, \dots, n$  are  $X_1, X_2, \dots, X_n$  respectively. We first use forward implication of these values to compute  $Y_1, Y_2, \dots, Y_{n-2}$ , the values of nets  $n+1, n+2, \dots, n+(n-2)$  respectively. Then using the value  $Z$ , which is the required value at the output of gate  $G$ , we apply the backward implication rules for gate  $G_{n-1}$

to obtain  $Z_{n-2}$  and  $X'_n$ , the new values of nets  $n + (n - 2)$  and  $n$  respectively. Having done that, we proceed backwards and apply the backward implication rules for all the gates, one at a time, ending with gate  $G_1$ .

It has been shown in [2] that the above procedure will stabilize in a single pass, unlike the approach followed in [3] which may require several passes.

## A.6 Speed-up Techniques

### A.6.1 Use of the Contrapositive

The use of the contrapositive to reduce the search space was first suggested by Schulz, et al., in SOCRATES [18]. However, the procedure presented in SOCRATES can only be used to backward imply the value 0 or 1.

In our 15-valued system, assume that the forward implication of a value  $L_1$  at net  $m_1$  with  $0/1/D/\bar{D}$  at all other nets yields the value  $L_2$  at net  $m_2$ . Thus when we require a value  $L'_2 \subseteq ((0/1/D/\bar{D}) - L_2)$  at net  $m_2$ , then the value of net  $m_1$  cannot contain any element of  $L_1$ . However, in some cases the backward implication may yield the same information. Hence it is useful to identify the conditions under which a backward implication cannot yield the information provided by a contrapositive assertion. In such cases we may store this information for possible future use. To obtain the implications for all possible values of  $L_1$  we only need to perform implications for each individual element of  $0/1/D/\bar{D}$ . Thus the procedure to obtain the implications for the 15-valued system, henceforth referred to as **15-VP**, would be to set the value of net  $m_1$  to each of the values 0, 1,  $D$  and  $\bar{D}$ , one at a time and with  $0/1/D/\bar{D}$  at all other nets, and observe the implied value at net  $m_2$ . It can be shown that the information yielded by **15-VP** can be obtained from a simpler procedure that utilizes a 3-valued (0, 1, 0/1) logic system [2]. In this procedure, which we denote as **3-VP**, we set the value of net  $m_1$  to each of the values 0 and 1, one at a time and with 0/1 at all other nets, and observe the implied value at net  $m_2$ . Table A6 shows how the

information yielded by 15-VP can be obtained by the results of 3-VP.

	Value applied at net $m_1$	Implied value at net $m_2$								
		(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	(ix)
3-VP	0	0/1	0	1	0	1	0	1	0/1	0/1
	1	0/1	0	1	1	0	0/1	0/1	0	1
15-VP	0	0/1/D/ $\bar{D}$	0	1	0	1	0	1	0/1/D/ $\bar{D}$	0/1/D/ $\bar{D}$
	1	0/1/D/ $\bar{D}$	0	1	1	0	0/1/D/ $\bar{D}$	0/1/D/ $\bar{D}$	0	1
	D	0/1/D/ $\bar{D}$	0	1	D	$\bar{D}$	0/D	1/ $\bar{D}$	0/ $\bar{D}$	1/D
	$\bar{D}$	0/1/D/ $\bar{D}$	0	1	$\bar{D}$	D	0/ $\bar{D}$	1/D	0/D	1/ $\bar{D}$

Table A6. Relationship between 3-VP and 15-VP

We now present a procedure which, when incorporated into the pre-processing phase, can derive all the contrapositive assertions for our 15-valued system. For ease of explanation we define the values 0 and 1 as "singleton" values.

1. Construct two test cubes  $tc_{00}$  and  $tc_{01}$  in which the values of all nets of the circuit are set to 0/1.
2. In  $tc_{00}$  ( $tc_{01}$ ) change the value of net  $m_1$ , where  $m_1$  is a FOS net, to the singleton value  $L_1(\bar{L}_1)$  and perform a forward implication of this value.

Let  $L_2$  ( $L_3$ ) be the implied value at the output net  $m_2$  of gate  $G$ .

3. If both  $L_2$  and  $L_3$  are singleton values, then both these implications ( $L_1$  at  $m_1 \implies L_2$  at  $m_2$  and  $\bar{L}_1$  at  $m_1 \implies L_3$  at  $m_2$ ) need to be stored.
4. If only one of the values (say  $L_2$ ) is singleton and this value  $L_2$  and the gate  $G$  happen to be one of the combinations listed in Table A7, then this implication ( $L_1$  at  $m_1 \implies L_2$  at  $m_2$ ) should be stored.
5. Repeat steps 1-4 for all FOS nets.

$L_2$	$G$			
0	OR	NAND	XOR	XNOR
1	NOR	AND	XOR	XNOR

Table A7. ( $L_2, G$ ) combinations that yield useful contrapositive assertions

The “learning procedure” presented in SOCRATES [18] performs the 0 and 1 implications for all nets of the circuit while we need to do this for only FOS nets. It is easy to show that the information for all other nets can be derived from this because of the deterministic nature of our backward implication procedure [2]. Hence our procedure generates the contrapositive assertions in the 15-valued system and yet requires less computation and storage than the method proposed in [18].

Note the contrapositive assertions in the 15-valued system corresponding to the implications stored by the above procedure can be generated using Table A6. It has been shown in [2] that if any implication was not stored by the above procedure, then either its corresponding contrapositive assertions yield no information or the information yielded can be derived by using the stored contrapositive assertions and the backward implication rules.

### A.6.2 Conditional Headlines

TOPS [14] extended the concept of headlines introduced in FAN [10] by using circuit topology to identify more nodes whose value justification could be postponed until the last stage of test generation. However, none of these schemes take advantage of the additional restrictions imposed by a particular fault. These restrictions might identify a potentially larger set of circuit nets whose value justification may be postponed.

Let the output net  $m_1$  of a gate  $G$  be a variant net with a singleton value in  $T_f(p_i)$ . Consider the tree  $T$  which is a subgraph of the dominator forest and whose root is net  $m_1$ . Furthermore consider the subtree  $T_1$  of  $T$  which does not contain any of the subtrees of  $T$  whose roots  $m_k, m_k \neq m_1$ , correspond to FOS nets. Note that  $T_1$  corresponds to the largest fanout-free subcircuit whose output is net  $m_1$  and whose



inputs are FOBs and/or PIs. Net  $m_1$  is defined to be a **conditional headline** if and only if all the nets corresponding to the FOB nodes in  $T_1$  have singleton values in  $\mathbf{T}_f(\mathbf{p}_i)$ .

We now show that if net  $m_1$  is a conditional headline, then it can be set to either of the singleton values 0 or 1, subject to the condition that the values of all the FOB nodes in  $T_1$  can be satisfied. Note that if  $m_1$  is a conditional headline, then  $T_1$  satisfies the following properties:

(i) The values in  $\mathbf{T}_f(\mathbf{p}_i)$  of the nodes in  $T_1$  can only be 0, 1, or 0/1. This is because if the value of any node includes either  $D$  or  $\overline{D}$ , then this must be due to a fault at node  $m_l$  which belongs to  $T_1$  since all FOB nodes have singleton values. But  $m_1 \in D(m_l)$ , and hence  $m_1$  would have a sensitized value.

(ii) At least one leaf of  $T_1$  is a PI net whose value in  $\mathbf{T}_f(\mathbf{p}_i)$  is 0/1 because  $m_1$  is a variant net with a singleton value in  $\mathbf{T}_f(\mathbf{p}_i)$ .

Consider a node  $m_j$  in  $T_1$  which has a singleton value and whose parent node has the value 0/1 in  $\mathbf{T}_f(\mathbf{p}_i)$ . Note that the value of  $m_j$  is a non-controlling input value for the gate it drives since  $\mathbf{T}_f(\mathbf{p}_i)$  is a deterministic test cube. If we delete from  $T_1$  all the subtrees which have any such  $m_j$  as a root, then the remaining tree corresponds to a fanout-free circuit whose output is net  $m_1$  and whose inputs have the value 0/1 in  $\mathbf{T}_f(\mathbf{p}_i)$ . Thus any required singleton value of net  $m_1$  can be satisfied by specifying the unassigned PIs in  $T_1$  subject to the condition that the values of the FOB nets in  $T_1$  can be satisfied. Note that this assignment does not interfere with the requirement of other variant nets since  $m_1$  is a dominator for all these PIs.

### A.6.3 Backward Implication of the Desensitizing Values

In this section we discuss how backward implication of the desensitizing value from variant nets may help speed-up the enumeration process. Consider the output net  $m_1$  of a gate  $G_1$  which is variant w.r.t.  $\mathbf{T}_f(\mathbf{p}_i)$  and has the value  $L_1$ . Let  $L'_1$  be the value

implied at net  $m_1$  by the values in  $T_f(\mathbf{p}_i)$  of the inputs of  $G_1$ . We construct a new test cube  $T'_f(\mathbf{p}_i)$  which is identical to  $T_f(\mathbf{p}_i)$  except that net  $m_1$  has the value  $L'_1 - L_1$ . Note that the value  $L'_1 - L_1$  at net  $m_1$  desensitizes path  $\mathbf{p}_i$ . Using  $T'_f(\mathbf{p}_i)$  we backward imply the value  $L'_1 - L_1$  at net  $m_1$  by applying only the backward implication rules and the stored contrapositive assertions and observe the nets whose values change in the process. Let  $m_j, 2 \leq j \leq J$ , be the nets where this backward implication terminates. Note that  $m_j$  is either a PI or the output of a gate whose input values do not change during this process. Also, let  $L'_j, 2 \leq j \leq J$ , be the new value obtained at net  $m_j$  by the above procedure.

Since the value  $L'_1 - L_1$  at net  $m_1$  implies that the value of net  $m_j$  is  $L'_j$ , we know from the contrapositive principle that, for *any*  $j, 2 \leq j \leq J$ , if the value of net  $m_j$  does not contain any of the values in the set  $L'_j$ , then the value at net  $m_1$  will not contain any of the values in the set  $L'_1 - L_1$  and hence  $m_1$  will become an invariant net w.r.t.  $T_f(\mathbf{p}_i)$ . A sufficient condition to make  $m_1$  an invariant net without interfering with the requirements of other variant nets is that there exists some  $m_j$  such that  $m_1 \in D(m_j)$  and  $m_j$  is a basis node. If  $m_j$  is not a basis node but is a conditional headline w.r.t.  $T_f(\mathbf{p}_i)$ , then net  $m_1$  can still be made invariant by removing the value  $L'_j$  from net  $m_j$ , provided the conditions that make net  $m_j$  a conditional headline are satisfied.

## A.7 Examples

**Example A1.** Let us reconsider the circuit of Fig. A1 with the fault net 3  $s - a - 0$  to highlight the improvements obtained by the speed-up techniques. Note that  $tc_1$  will be identical to that discussed earlier. However, the new  $d(tc_1)$ , shown below, is different because the use of the contrapositive assertion and the value of net 30 drops the value 1 from net 24 which has further deterministic implications.

2	5	9	14	15	16	19	20	21	22	23	24	25	26
0	0	0	D	D	D	0/D	0/ $\bar{D}$	0/D	0/D	0/ $\bar{D}$	0/D/ $\bar{D}$	0/D/ $\bar{D}$	0/D/ $\bar{D}$
30	32	33	37	38	39	40	41	42	43				
0/D/ $\bar{D}$	0	0	D/ $\bar{D}$	D/ $\bar{D}$	D/ $\bar{D}$	1/D/ $\bar{D}$	1/D/ $\bar{D}$	1/D/ $\bar{D}$	1/D/ $\bar{D}$				

As before we continue by setting the value of net 21 to  $D$  in  $tc_2$ . When we construct  $d(tc_2)$ , further use of the contrapositive drops  $\bar{D}$  from the value of net 24 and more deterministic changes occur as shown below:

4	6	17	18	20	23	24	25	26	27				
1	1	1	1	0	0	0/D	0/D	0/D	0/1/D				
28	29	30	31	36	37	38	39	40	41	42	43		
0/1/D	0	0/D	D	D	D	D	D	D	1/D	1/ $\bar{D}$	1	1/ $\bar{D}$	

Since net 42 has the value 1 in  $d(tc_2)$  the attempt to sensitize the path through net 39 leads to a contradiction in the construction of  $tc_3$  and the computation for the construction of  $d(tc_3)$  is avoided.

Steps in  $tc_3$  construction:

$$\frac{42}{1 \cap (D/\bar{D}) = \emptyset \quad (\text{Contradiction})}$$

As before, we now extend the sensitized path through net 37 by setting the value of net 40 to  $D$  in  $tc_4$  and obtain  $d(tc_4)$  as shown below:

10	11	34	35	41	43	45						
0	0	0	0	1	1	D						

The only variant net is net 30 and its desensitizing value is 1. Using the procedure explained in §A.6.3 we backward imply the value 1 at net 30 to get the value 1 at nets 7 and 8. Since both nets 7 and 8 are basis nodes and  $30 \in (D(7) \cap D(8))$ , then removing the value 1 from either net 7 or 8 would give a test for the fault.  $\square$

**Example A2.** Consider the class of circuits shown in Fig. A6 with the fault net  $3s - a - 0$ . Note that the ECAT circuit considered by Goel to illustrate the efficiency of PODEM [12] is an element of this class. Using  $D(3) = \{5, 7\}$  we construct  $tc_1$  as shown below where all other nets have the value 0/1.

1	2	3	$3_f$	4	5	6	7
0/1	0/1	1	$D$	0/1	$D/\bar{D}$	0/1	$D/\bar{D}$

The only changes that occur when  $d(tc_1)$  is constructed is that the value of nets 1 and 2 become 1. Since we have a sensitized path from  $3_f$  to the PO and there are no variant nets, a test has been generated. Note that the algorithm specifies only the value of PI nets 1 and 2 because it takes full advantage of the linearity of XOR gates.  $\square$

**Example A3.** In this example we illustrate the use of conditional headlines. Consider the circuit in Fig. A7 whose only basis nodes, other than the PO, are the PIs. The only possible  $T_f(p_1)$  for the fault net  $2s - a - 0$  is shown below:

1	2	$2_f$	3	11	12	23	32	34	35
1	1	$D$	1	1	1	$\bar{D}$	$D$	0	$D$

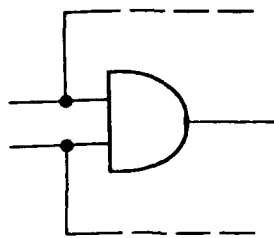
Net 34 is the only variant net and its desensitizing value is 1. Thus we backward imply the value 1 from net 34 which sets nets 31 and 33 to the value 1. The use of the contrapositive sets the value of nets 20 and 24 to 1. It can now be verified, using the procedure of §A.6.2, that net 24 is a conditional headline and net 20 is not. Furthermore, the only condition required for net 24 to have the same independence property as a headline is that the value 1 at net 12 be satisfied. This condition is already met because net 3 is a PI. Thus the required value 0 at 24, which makes net 34 an invariant net with the value 0, can be met by specifying the PI nets 4, 5 and 6.  $\square$

## References

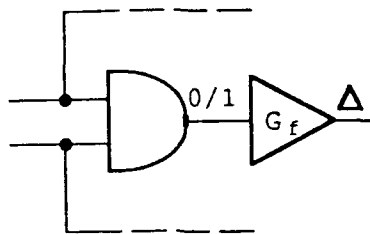
- [1] M. Abramovici, P. R. Menon and D. T. Miller, "Checkpoint Faults are not sufficient Faults for Test Generation," *IEEE Transactions on Computers*, Vol. C-35, pp 770-771, August 1986.
- [2] A. M. Ali and C. R. P. Hartmann, "A 15-Valued Algorithm for Test Pattern Generation," Technical Report No. CIS 89-2, School of Computer & Information Science, Syracuse University, Syracuse NY 13244.
- [3] Sheldon B. Akers, "A Logic System for Fault Test Generation," Presented at Symposium on Fault-Tolerant Computing, Paris, France, June 1975. Also *IEEE Transactions on Computers*, Vol. C-25, pp 620-630, June 1976.
- [4] M. A. Breuer and A. D. Friedman, *Diagnosis & Reliable Design of Digital Systems*. Computer Science Press, 1976.
- [5] Charles W. Cha, William E. Donath and Füsün Özgüner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Transactions on Computers*, Vol. C-27, pp 193-209, March 1978.
- [6] S. J. Chandra and J. H. Patel. "Experimental Evaluation of Testability Measures for Test Generation." *IEEE Trans. on CAD*, Vol. 8, pp 93-98, January 1989.
- [7] S. J. Chandra and J. H. Patel. "Test Generation in a Parallel Processing Environment." *Proceedings of the IEEE International Conference on Computer Design (ICCD-88)*, pp 11-14, October 1988.
- [8] Wu-Teng Cheng, "Split Circuit Model for Test Generation," in *Proceedings of the 25<sup>th</sup> ACM/IEEE Design Automation Conference*, pp 96-101, 1988.

- [9] Warren Debany, "On Using the Fanout-Free Substructure of General Combinational Networks," Ph.D. Dissertation, Syracuse University, December 1985.
- [10] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, Vol. C-32, pp 1137-1144, December 1983.
- [11] H. Fujiwara and S. Toida, "The complexity of fault detection: An approach to design for testability," in *Proceedings of the 12<sup>th</sup> International Symposium on Fault Tolerant Computing*, pp 101-108, June 1982.
- [12] Prabhakar Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, Vol. C-30, pp.215-222, March 1981.
- [13] O. H. Ibarra and S. K. Sahni, "Polynomially Complete fault detection problems," *IEEE Transactions on Computers*, Vol. C-24, pp 242-259, March 1975.
- [14] Tom Kirkland and M. Ray Mercer, "A Topological Search Algorithm for ATPG," in *Proceedings of the 24<sup>th</sup> ACM/IEEE Design Automation Conference*, pp 502-508, 1987.
- [15] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, vol. C-25, pp 550-555, June 1976.
- [16] Srinivas Patil and Prith Banerjee. "A Parallel Branch and Bound Algorithm for Test Generation," *Proceedings of the 26<sup>th</sup> ACM/IEEE Design Automation Conference*, pp 339-344, 1989.
- [17] J. P. Roth, W. G. Bouricius and P. R. Schneider, "Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits," *IEEE Transactions on Computers*, Vol. C-16, pp 567-579, October 1967.

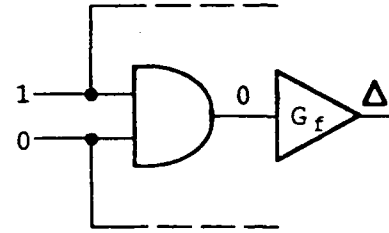
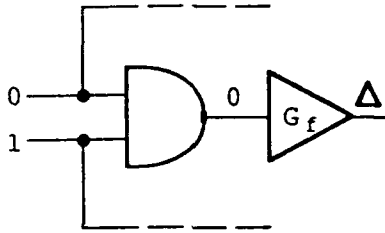
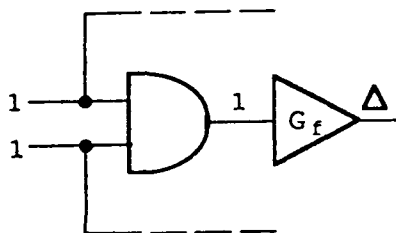
- [18] Michael H. Schulz, Erwin Trischler and Thomas M. Sarfert, "Socrates—A Highly Efficient Automatic Test Pattern Generation System," in *Proceedings of the 1987 International Test Conference*, pp 1016–1026, 1987.
- [19] R. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal of Computing*, Vol. 3, pp 62–89, 1974.



(a)



(b)



(c)

Fig.1 Overlapping the testing of several checkpoints

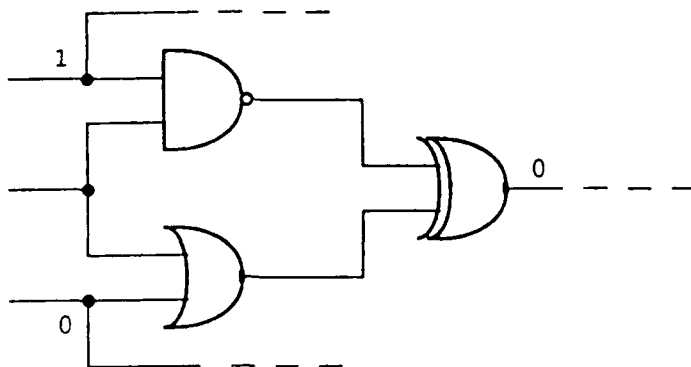


Fig.2 Conditional basis node with unsatisfiable value.



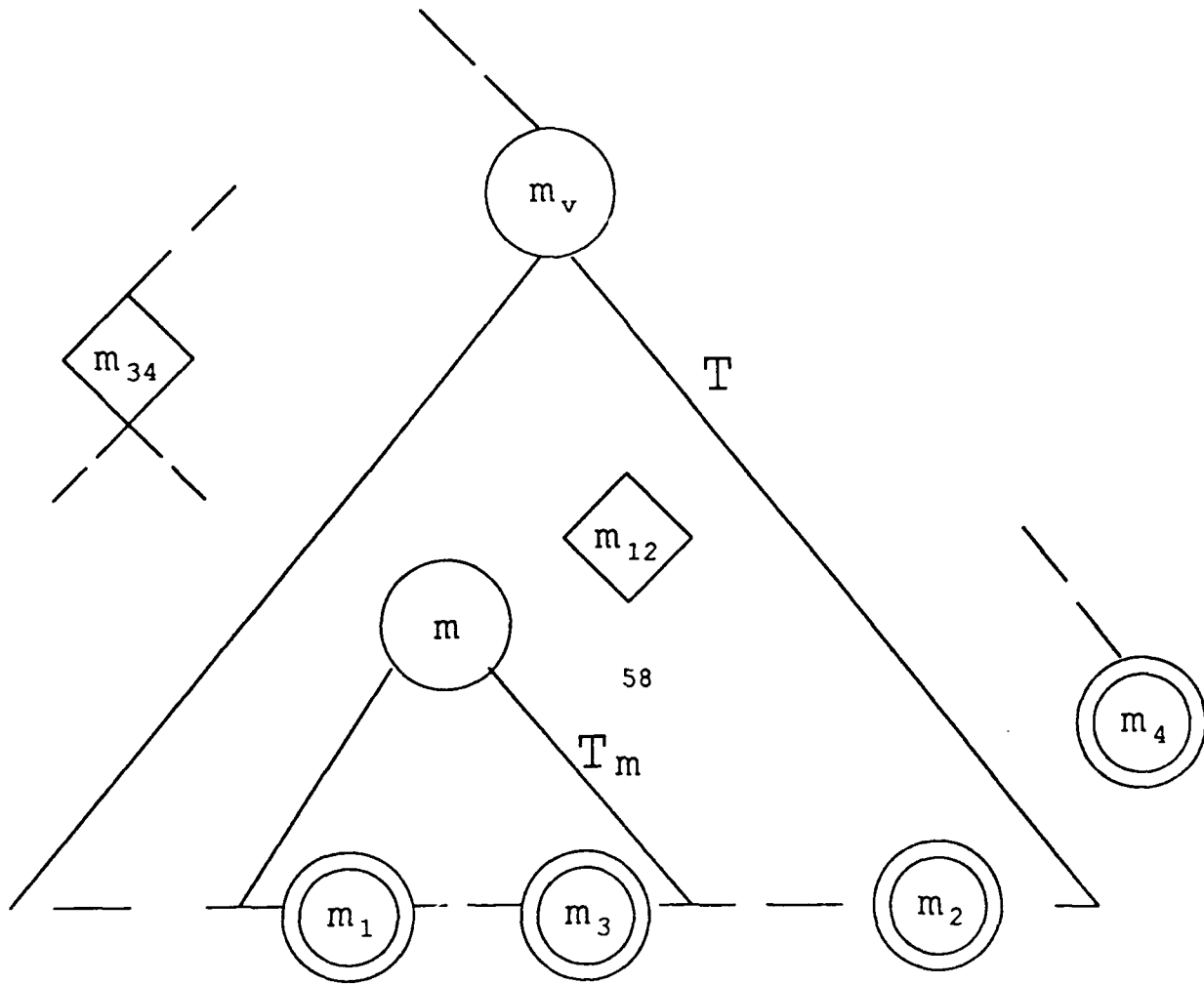


Fig. 3 Example where the subtree corresponding to a unit-valued node cannot be deleted.

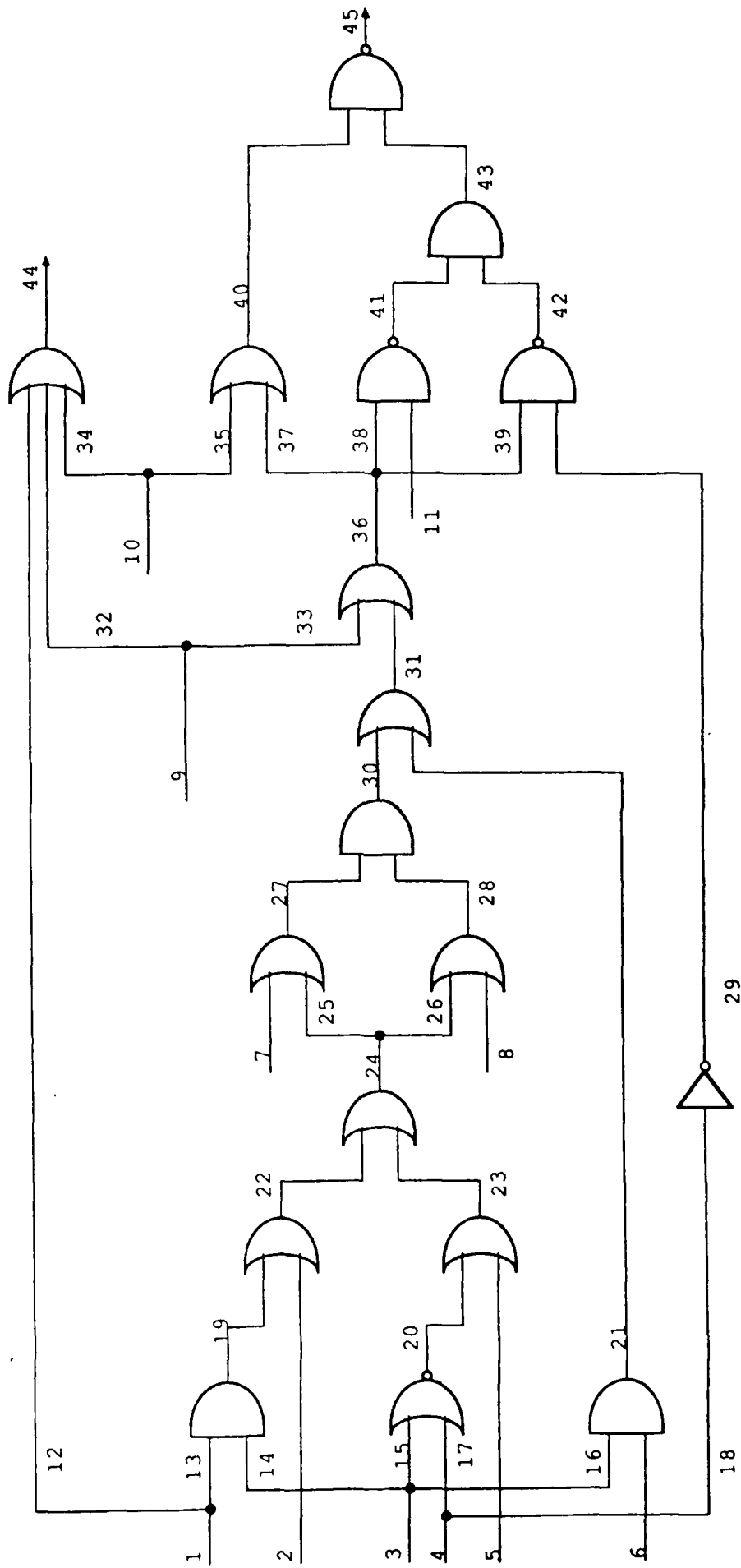
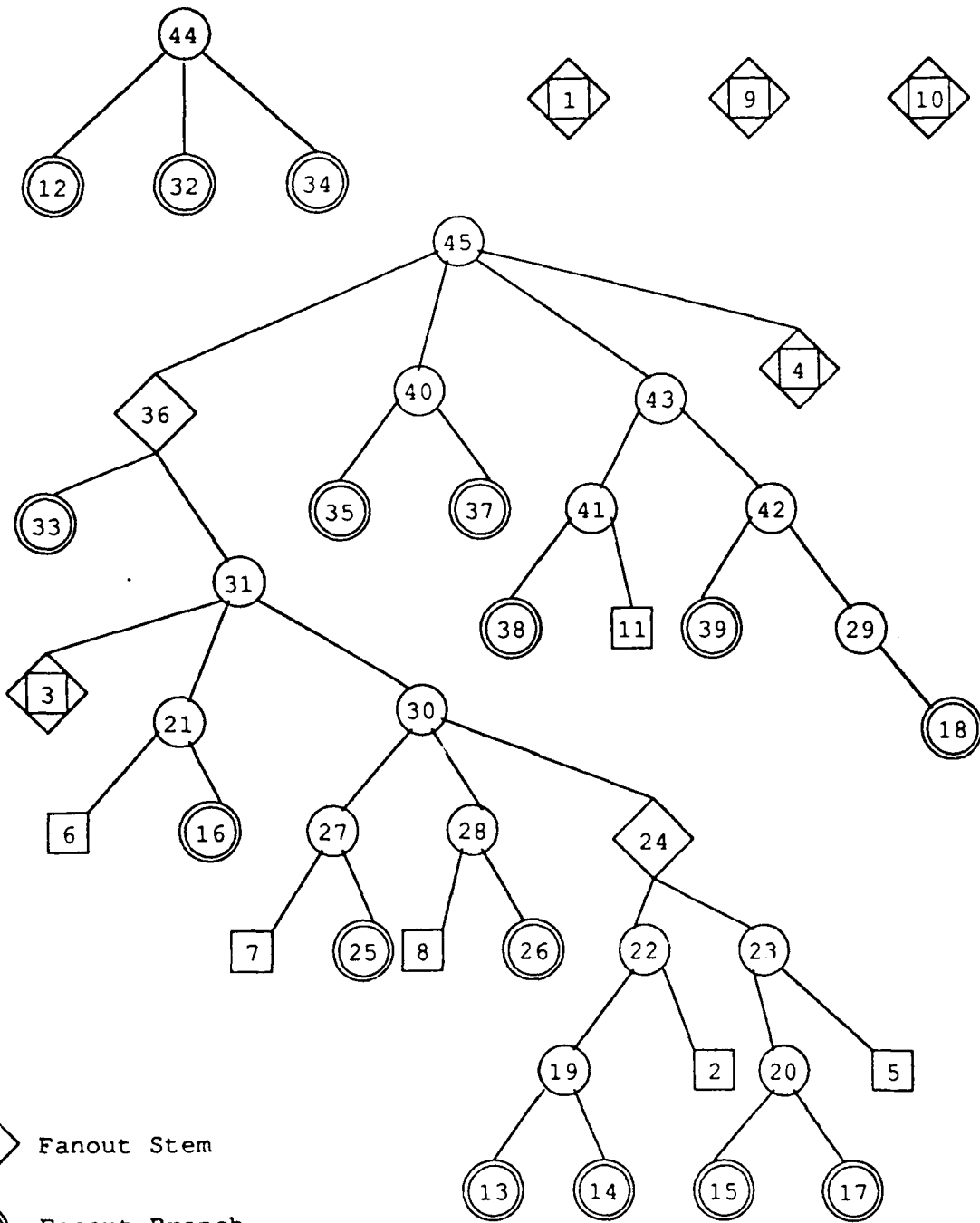


Fig.A1 An example circuit



KEY:  
 ◊ Fanout Stem  
 ○ Fanout Branch  
 □ Primary Input

Fig.A2 Dominator forest for circuit of Fig.A1



Fig.A3 Introduction of fictitious gate

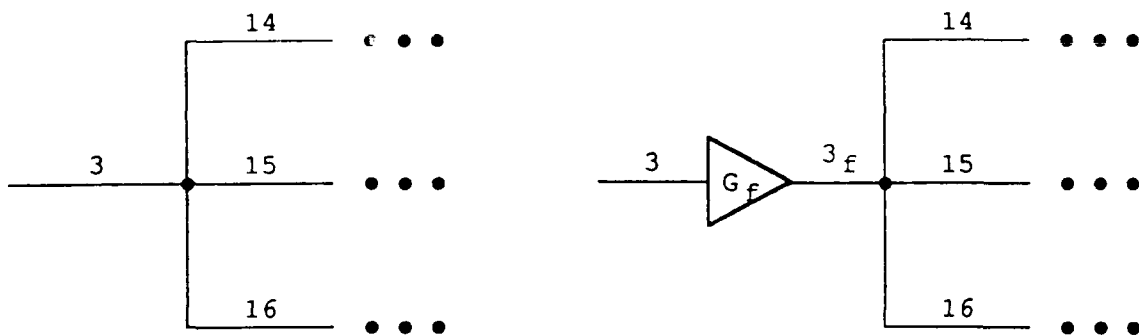


Fig.A4 Fictitious gate for net 3 s-r-0 in circuit of Fig.A1

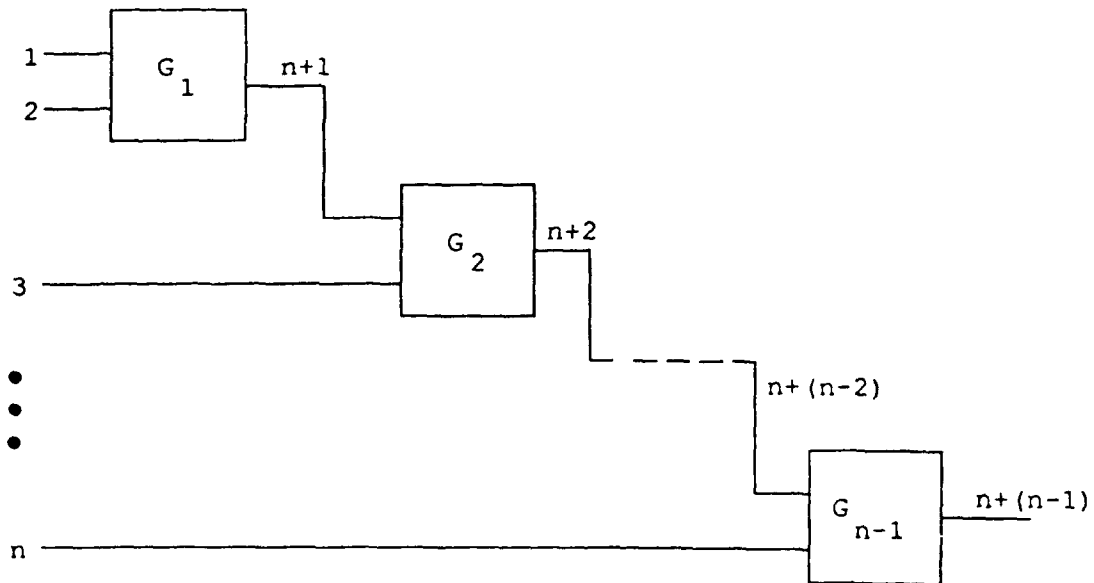
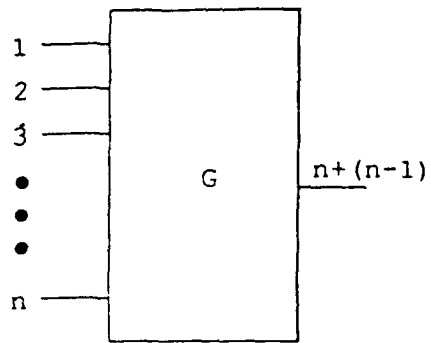


Fig.A5 Gate decomposition

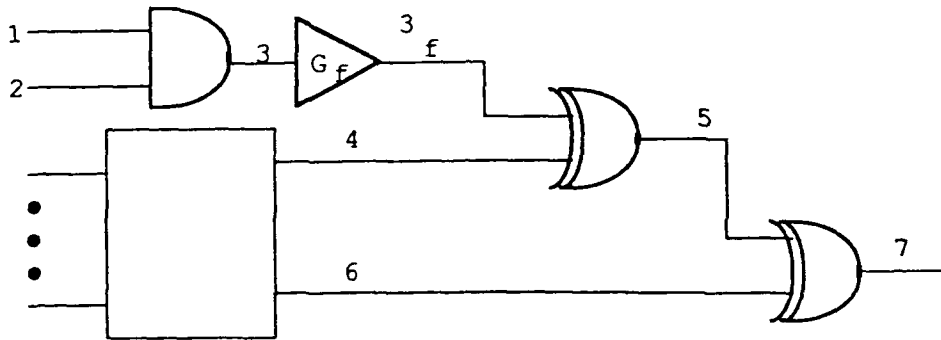


Fig.A6 Circuit for Example A2

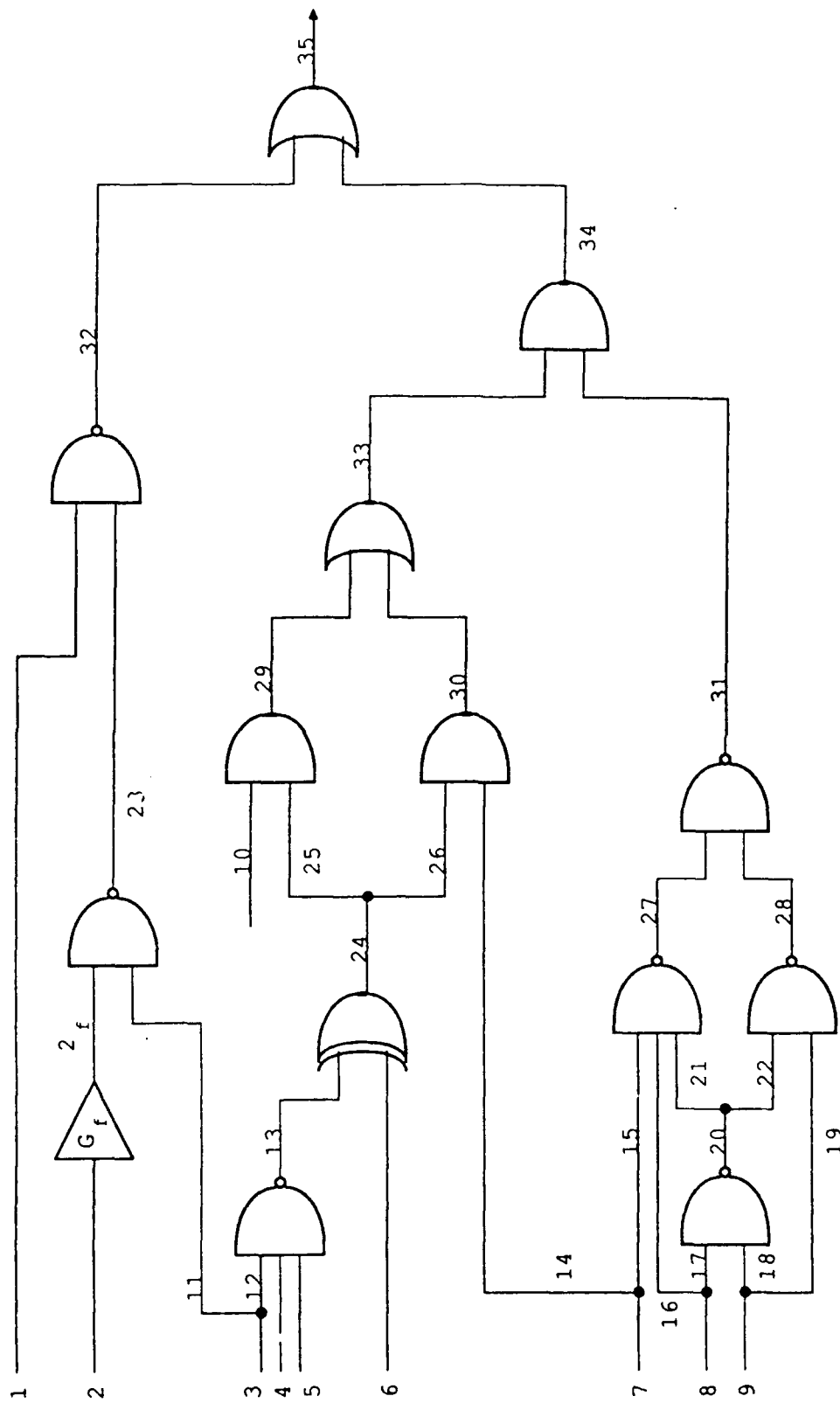


Fig.A7 Circuit for Example A3