## INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International

300 N. Zeeb Road
Ann Arbor, MI 48106

Order Number 1333602

# The evaluation of the PODEM algorithm as a mechanism to generate goal states for a sequential circuit test search

Lee, Hoi-Ming Bonny, M.S.

The University of Arizona, 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# THE EVALUATION OF THE PODEM ALGORITHM AS A MECHANISM TO GENERATE GOAL STATES FOR A SEQUENTIAL CIRCUIT TEST SEARCH

by

Hoi-Ming Bonny Lee

---

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 8

## STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be grant by head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Hoi-Ming Lee

## APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Fredrick J. Hill — 4 May 1988

Fredrick J. Hill
Professor of
Electrical and Computer Engineering

Date

## ACKNOWLEDGMENTS

I would like to express my gratitude to professor Fredrick Hill for his patient guidance and motivation throughout this research. Many of the ideas embodied in this thesis arose from his suggestions.

Special thanks are due Y. H. Chen for his assistance and helpful comments.

Indeed, I am indebted to my mother, Kan Tsun, for her constant support and encouragement.

# TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

# LIST OF ILLUSTRATIONS

## LIST OF ILLUSTRATIONS (CONTINUED)

## LIST OF TABLES

## LIST OF TABLES (CONTINUED)

## ABSTRACT

In a VLSI design environment, a more efficient test generation algorithm is definetly needed. This thesis evaluates a test generation algorithm, PODEM, as mechanism to generate the goal states in a sequential circuit test search system, SCIRTSS. First, a hardware description language, AHPL, is used to describe the behavior of a sequential circuit. Next, SCIRTSS is used to generate the test vectors. Several circuits are evaluated and experimental results are compared with data from a previous version of SCIRTSS which was implemented with the D-Algorithm. Depending on the number of reconvergent fanouts in a circuit, it is found that PODEM is 1 to 23 times faster than the D-Algorithm.

# CHAPTER 1

## INTRODUCTION

As the complexity of VLSI technology increases, the role of CAD tools becomes much more important. During the SSI age, the location of a logic gate malfunction could be spotted with a logic probe. But, this method can only be applied to a circuit with a small number of gates. Nowadays, with more than 100,000 gates in a VLSI circuit, manual testing is no longer applicable to such circuit size. Instead of using a logic probe to test each gate, a set of test vectors are applied to input pins of a VLSI circuit and output vectors from output pins are examined. A fault in a circuit is to be detected and located by examining those output vectors.

### 1 . 1 A Brief Introduction To SCIRTSS

SCIRTSS (Sequential Circuit Test Search System) [1] is an automatic functional test search system which bases on U-AHPL (Universal A Hardware Programming Language) [2] functional description of digital sequential or combinational circuits. U-AHPL is a register transfer type hardware description language which describes synchronous and asynchronous sequential circuits. Fig. 1.1 shows the system configuration of U-AHPL and SCIRTSS.

```
                    U-AHPL
                      |
                      v
Stage 1 compiler   --->-----
     |                     |
     v                     |
Stage 2 Compiler           v
     |                     |
     v                     |
Stage 3 compiler           |
     |                     |
     v                     |
  SCIRTSS  <---------------
ATPG(automatic test pattern generator)
```

Figure 1.1 U-AHPL and SCIRTSS System Configuration

The Stage 3 Compiler [1] produces a gate wiring list which serves as an input data file to SCIRTSS. Both U-AHPL and SCIRTSS can also handle combinational circuits. After SCIRTSS obtains the network list from the Stage 3 Compiler, it begins a Homing Experiment [3]. Then it goes to a fault list to select a fault that is going to be tested. The fault model used in SCRITSS is the stuck-at fault model, i.e. stuck-at-one or stuck-at-zero [4,5,6,7].

There are two heuristic search trees [8] in SCIRTSS. These are the propagation search and sensitization search. If there are faults stored in any storage devices, i.e. flip-flops, SCIRTSS will step into propagation search routine to find a sequence of input vectors which can propagate the stored faults to primary outputs. Otherwise, SCIRTSS applies the D-Algorithm [4,9] to find out what has to be sensitized and then enters a sensitization routine. Both intelligent search routines end up to a parallel fault simulator which performs fault list reduction. Afterward, if the fault list is not empty, SCIRTSS will continue finding a sequence of test vectors for the next fault in the fault list. It is possible that some faults are undetectable, in this case, SCIRTSS sets a time limit of finding test vectors for such fault. If no test is found within the time limit, SCIRTSS will treat the fault as undetectable and will delete it from the fault list.

At the end, SCIRTSS concatenates all test vectors found and gives a summary of fault coverage. Fig. 1.2 shows the structure of SCIRTSS.

```
                              AHPL Design
                                  |
                                  v
                          Homing Experiment
                                  |
--------------->--------------- v
|                             Select Faults
|                                 |
|                                 v
|                             Faults Stored ?
|                                 |
|                                 v
|                       ----------------------
|                   yes |                    | no
|                       v                    v
|                 Propagation      D-Algorithm (or PODEM)
|                   Search
|                       |                    |
|                       |                    v
|                       |              Sensitization
|                       |                 Search
|                       |                    |
|                       v                    v
^                       ------->-------<------
|                                 |
|                                 v
|                     Parallel Fault Simulation
|                        Fault List Reduction
|                                 |
|                                 v
|                        All Faults Detected ?
|                                 |
|                             no  v
|---------------------<---------
                                  | yes
                                  v
                                 End
```

Figure 1.2  SCIRTSS Structure

## 1.2 Other Test Generation Systems

Test generation process includes fault modeling and reduction, test pattern generation, fault simulation, fault coverage evaluation, and the production of a fault dictionary [5].

Test generation systems can be characterized to be deterministic and probabilistic [10]. Deterministic systems employ test generation algorithms, e.g. the D-Algorithm and PODEM [5,11], etc., to find test vectors. This kind of system tells the fault coverage afterward. SCIRTSS is a deterministic system. It tells the percentage of fault coverage and lists undetectable faults. A good candidate, other than SCIRTSS, is the HITEST system [4]. HITEST is based on a knowledged-based interactive test generation and simulation system. One of the subsystems in HITEST is the test generator problem solver and PODEM is the major algorithm in this system. The HITEST system employs CIRRUS Circuit Language(CCL) [4] to describe circuits and uses CIRRUS Waveform Language (CWL) [4] to control the test environment.

Probabilistic systems [10] apply probabilty to estimate the probabilty for detecting faults throughout the entire design. Because they run just one simulation and do not have to go through huge fault tables, they significantly reduce the amount of simulation time. This type of system makes assumptions about the testability [5] of certain structures that make the analysis much more simpler and speed up the overall process. These assumptions do not fully use the potential of those structures, and a risk exists that some faults will be missed. Such system does not tell the exact percentage of fault coverage.

Recently, the application of information theory [12] to faults testing draws very much attention. The concept is that more information known at outputs of a circuit will reduce the number of test vectors required to test all faults.

Many test generation systems are implemented in software. However, some systems use hardware to implement some of test generation algorithms in order to accelerate the whole process. Of course, hardware implemented systems run much faster than software inplementations. There are trade offs between these two kinds of implementations. Using software to implement a system is more fexible and less expensive. Hardware implementation raises the cost but achieves for higher speed.

## 1.3 Review Previous Work On SCIRTSS

SCIRTSS has existed for many years [9]. During this time, much effort has been applied to updating and improving the system. The original SCIRTSS was run in batch mode and could not handle VLSI circuits. Thanks to the contributions of many researchers in the Department of Electrical Engineering at the University of Arizona, the current version of SCIRTSS [1] is able to run in batch mode and interactive mode. The interactive mode system gives users the ability to change the system parameters, heuristic weights in both search trees and to observe fault behavior along the course of processing. SCIRTSS applies D-Algorithm [9,13] to generate goal states for sensitization search.

The D-Algorithm starts from a faulty net and drives the fault forward to a primary output [9]. Then a consistency drive is performed from the primary output backward to primary inputs for line justification. In order to generate a test vector efficiently, it is important to minimize the number of backtrackings [5] and to shorten the processing time between backtrackings. Recently, a new version of NDALG, called DF3 [13], was sucessfully interfaced to SCIRTSS. DF3 features the depth-first strategy.

Backtracking in the D-Algorithm consumes a significant amount of time to generate a test vector for a fault in a circuit with many reconvergent fanouts. The more reconvergent fanouts in a circuit, the more number of backtrackings may be required in performimg line justification [4]. The objective of this thesis is to evaluate a new test generation algorithm, called PODEM

[4,5], which does not result in as much backtrackings as in the D-Algorithm.

# CHAPTER 2

## PODEM ALGORITHM

Although the D-Algorithm [4] is one of the most popular test generation algorithms, its application in circuits ,that have many reconvergent fan-out [5], such as parity checking circuits and error detection and correction (EDAC) circuits, is unsatisfactory. EDAC-type circuits are characterized by consisting of some number of XOR trees with reconvergence. In this type of circuit, the D-Algorithm has to solve many signal conflicts arising from its line justification process. When the consistency drive traces backward from a primary output to a primary input, two or more paths with different signal values may reconverge at the same node. This type of signal conflict must not be allowed to exist. The D-Algorithm must go back to one of the previous nodes where an arbitrary decision was made and then choose another alternative assignment. Depending on how many signal conflicts have to be solved, this may be a very time consuming process.

In 1981, P. Goel [11] published a paper on a new algorithm to generate tests for combinational logic circuits. It is known as the PODEM Algorithm (Path Oriented Decision Making Algorithm). This new algorithm tackles the problems originated in the D-Algorithm.

## 2.1 Basic Concepts of PODEM Algorithm

PODEM attempts to reduce the number of backtrackings by working from primary inputs to primary outputs to generate a test vector. It begins by assigning X's (DON'T CARE) to all inputs and internal lines and then assigns a binary value to an unassigned primary input. Implications of each input assignment are determined. If either of the following two propositions [4] is true, the input assignment is rejected.

(1) The signal net for the stuck fault being tested has the same logic level as the stuck level.

(2) There is no signal path from an internal signal net to a primary output such that the internal signal net has value D or ^D and all other nets on the signal path are at X.

The first proposition excludes input combinations which produce the same signal for both the good machine and faulty machine at the fault site. The second proposition excludes input combinations which block all signal paths to propagate the fault from fault site to an output.

The assignment of a value to an input employs "branch and bound" method. For example, if the assignment of a '0' to an input is rejected, a '1' will become the next value to be assigned. Whenever an assignment is rejected, the node is flagged. A node is defined as a test vector or a vector of input values. The PODEM algorithm can be implemented by using the "last-in-first-out" technique [4]. When an input assignment is rejected and the corresponding node is also flaggged, PODEM will eliminate this node and go back to the previous node. Since every previous node has tried one of the binary values, there is only one binary value for the input associated with a previous node to try. If an input assignment is rejected but the corresponding node is not flagged, and the input will be assigned another value. However, the rejection of an input assignment accords to the above two propositions. If an input assignment is not rejected and no fault is propagated to outputs, PODEM goes to the next node, that means PODEM assigns a binary value to an unassigned input according to the above scheme. When a fault is propagated to any output, a test vector for a fault is announced to be found and PODEM stops. If the next unassigned input is not available, PODEM will go back to a previous node that has not been flagged. When a root node is reached, PODEM stops and announces no test vector is found. Fig. 2.1 shows the flowchart of PODEM algorithm.

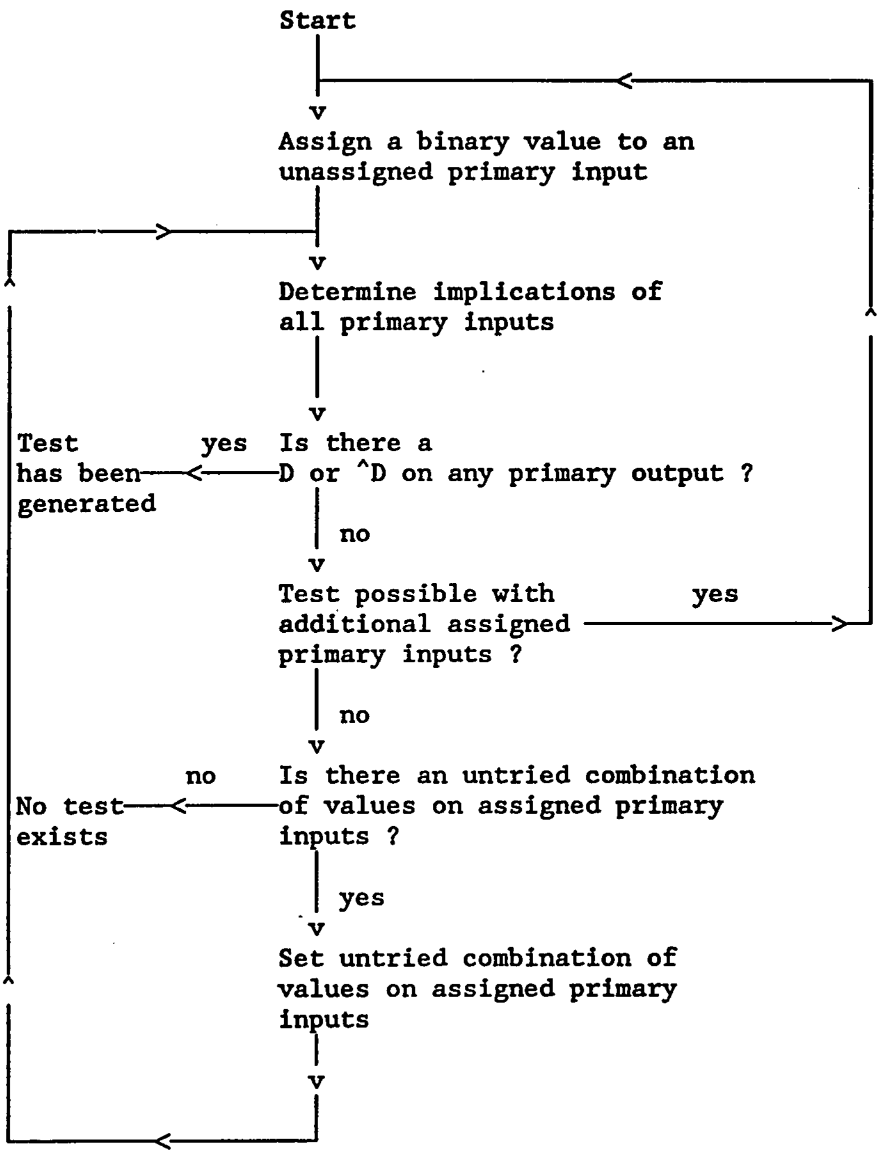


**Figure 2.1 Flowchart of PODEM Algorithm**

Fig. 2.2 shows an ECAT circuit with its major elements consisting of XOR's. For the fault H s-a-0 (stuck at zero), the D-Algorithm may go through the following steps. First, the primitive D-cube is chosen by assigning A=1, B=1 and H=D. And then the faulty signal is propagated to the output R by performing a series of D-intersections of the test cube and propagation D-cubes. After the D forward drive, N=1, P=^D, Q=1, and R=D. The next step is to enter the line jusification process which is a backward process. It is found that the assignment of N=1 and Q=1 is not possible to realize the complementary functions. However, the D-algorithm must enumerate input values exhaustively until the absence of justification is confirmed. And then it goes to other decision nodes to explore other alternatives, but each time before it explores another alternative, it has to backtrace exhaustively to confirm the justification that is non available. Fig. 2.3 shows the decision tree in D-Algorithm.

For the same circuit in Fig. 2.2, considering H s-a-0, PODEM firstly assigns X's(DON'T CARE) to all lines. Then, it assigns A=0, but A=0 produces the same signal level as the faulty signal at H, PODEM assigns A=1. Whenever an input assignment is tried, a forward implication drive should be followed. And then B=0 is chosen, but this meets proposition 1, so B=1 is chosen again. Then C=0 is chosen which is followed by choosing E=0, F=0, and G=0; now the goal node is reached. The faulty signal is finally propagated to output R. The sucessful test vector is (110000). It is noticed that much work has been saved with the elimination of the input combinations consisting of either A=0 or B=0.

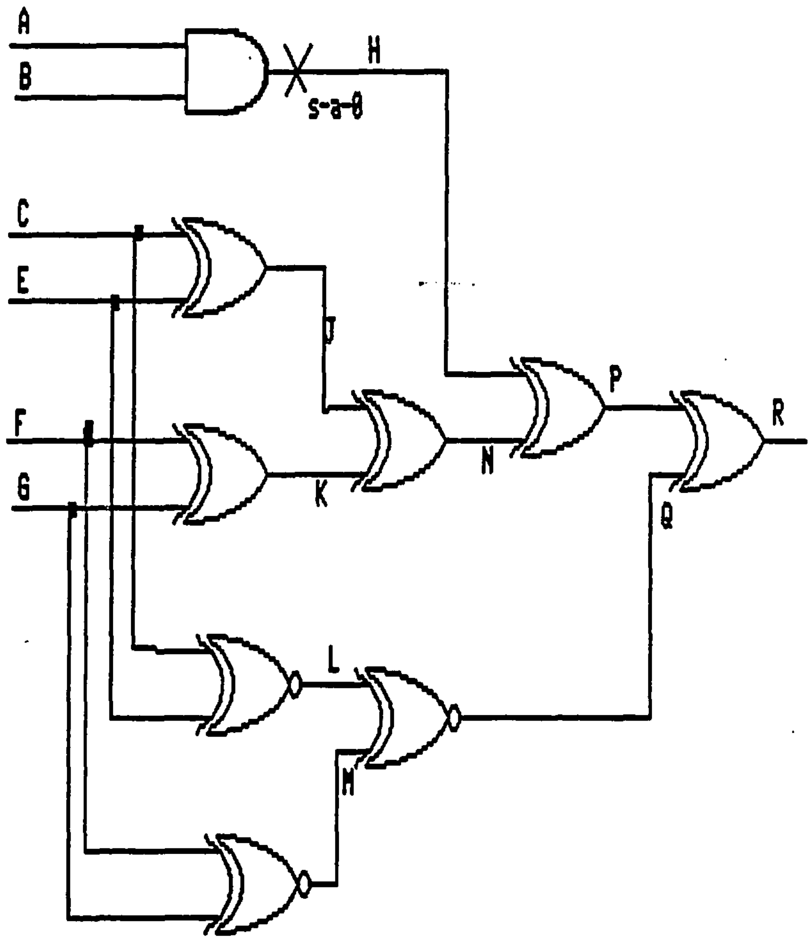


Figure 2.2 ECAT Circuit

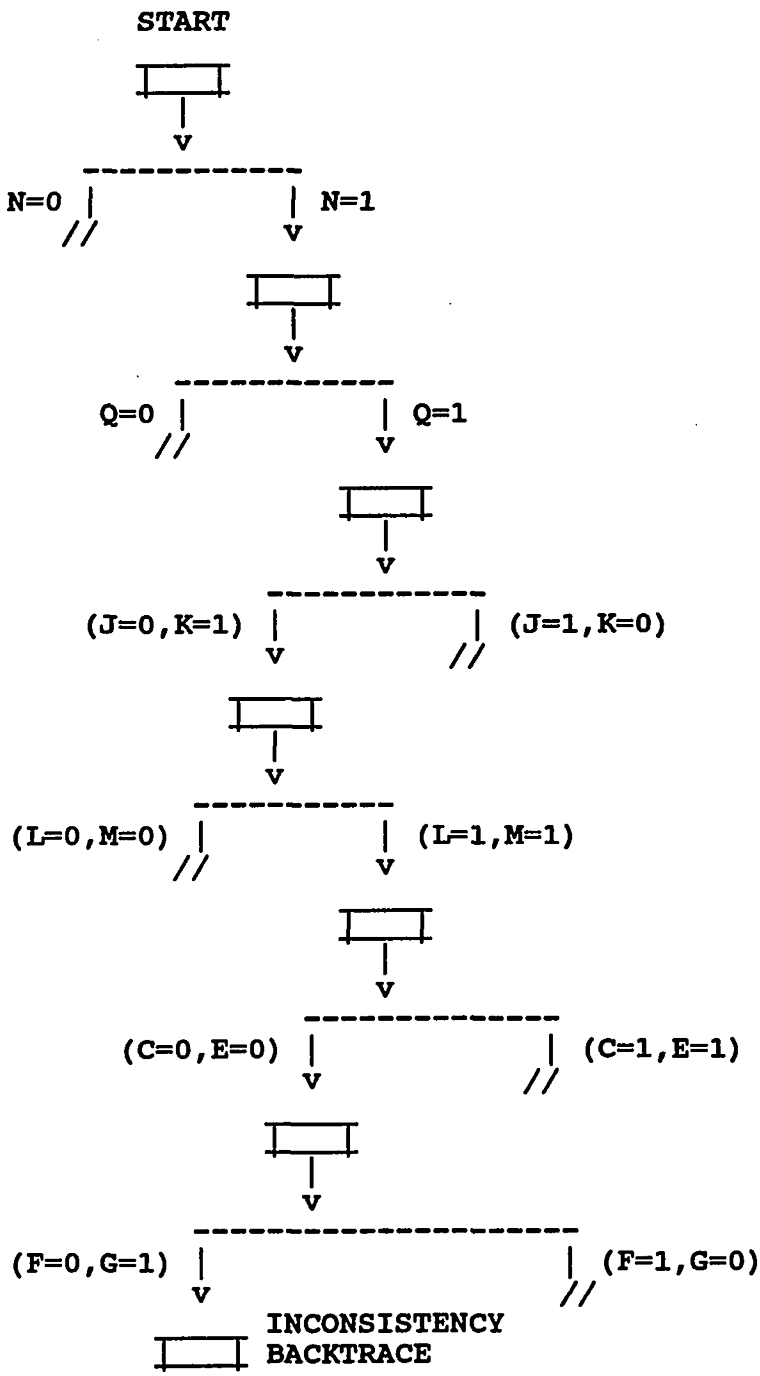


Figure 2.3 Decision Tree of D-algorithm

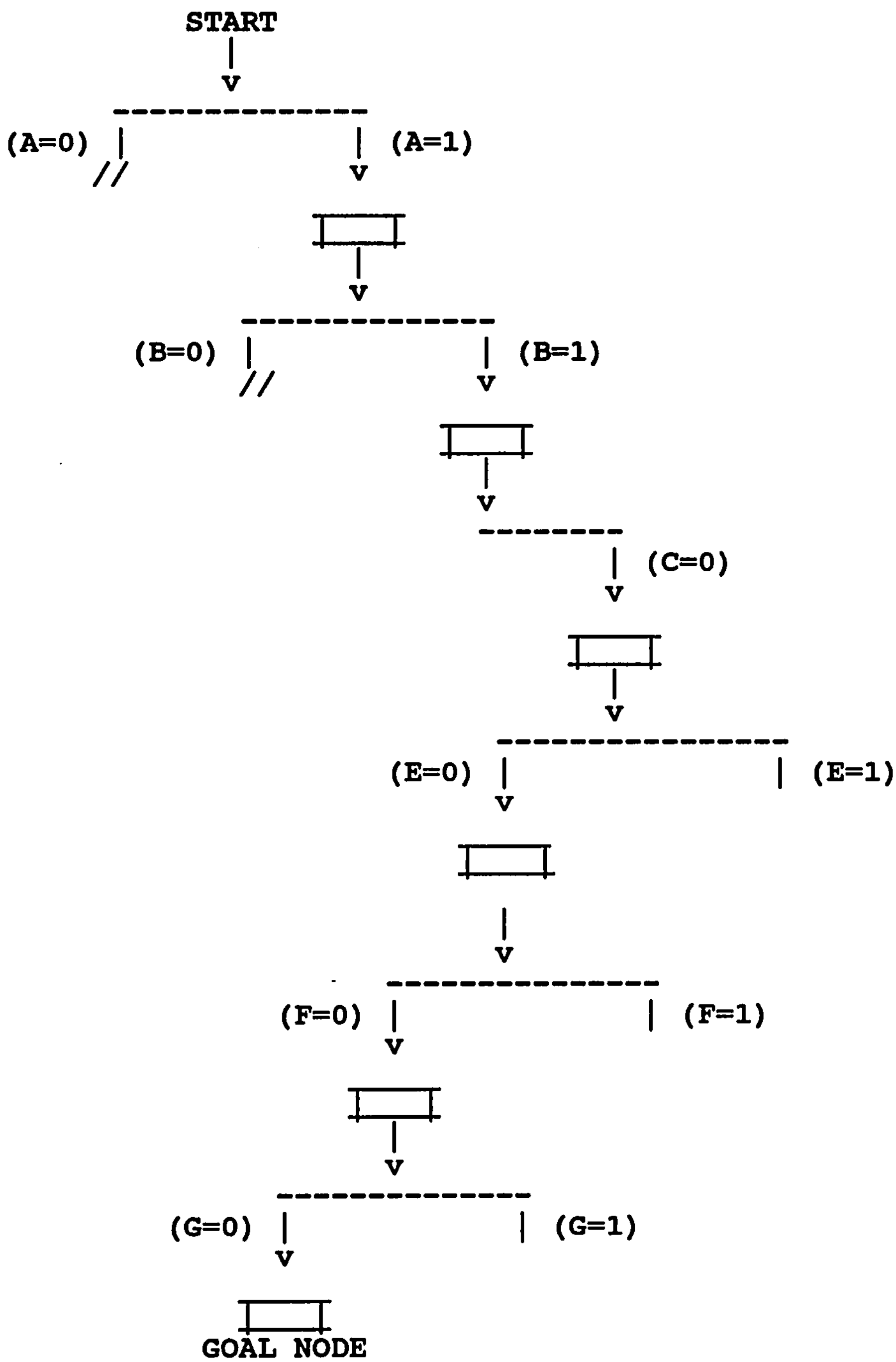


Figure 2.4 Decision Tree of PODEM Algorithm

One interpretation of the PODEM algorithm calls for assignings values to input lines in alphabetical order and always trys the value 0 before the value 1. Fig. 2.5 shows another logic circuit and Fig. 2.6 shows the PODEM decision tree of the logic circuit with P s-a-0. When a node meets one of the propositions [4], the node is bounded, its associated input assignment must be reassigned to X (DON'T CARE) and its previous node is then flagged. PODEM begins by setting all lines to be X's (DON'T CARE) and then assigns A=0. Since the fault signal has not been initiated, PODEM continues to assign B=0. Again, the fault signal is still not generated. PODEM continues to assign C=0. But C=0 causes the stuck fault network to have the same value as the good network at P, so the node is bounded and the previous node is flagged. PODEM branches to the next node by assigning C=1. Again, the assignment of C=1 causes a stuck fault the same as a good network, and the node is bounded. Since there is no other branch node available, PODEM goes back to the previous node. Because the previous node is flagged, PODEM has to go back to a further node. Now, the node is unflagged, PODEM goes to the next available branch node by assigning B=1. Again, the fault signal is not activated. PODEM continues to assign C=0. Now the assignment C=0 causes stuck fault same as a good networ, and PODEM goes to the next available branch node by assigning C=1. Next the assignment D=0 is chosen, but this assignment causes all paths blocked from the fault site to the output. PODEM abandons this node and goes to the next branch node by assigning D=1. Now the fault signal is propagated to the output. PODEM announces that the goal node has been reached. The sucessful test vector is (0111).

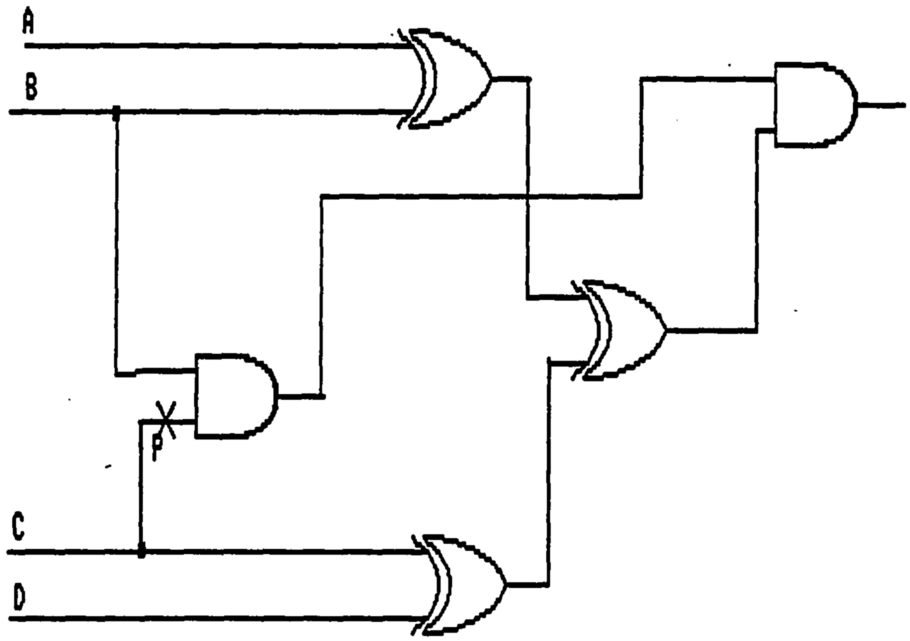


**Figure 2.5 Circuit II**

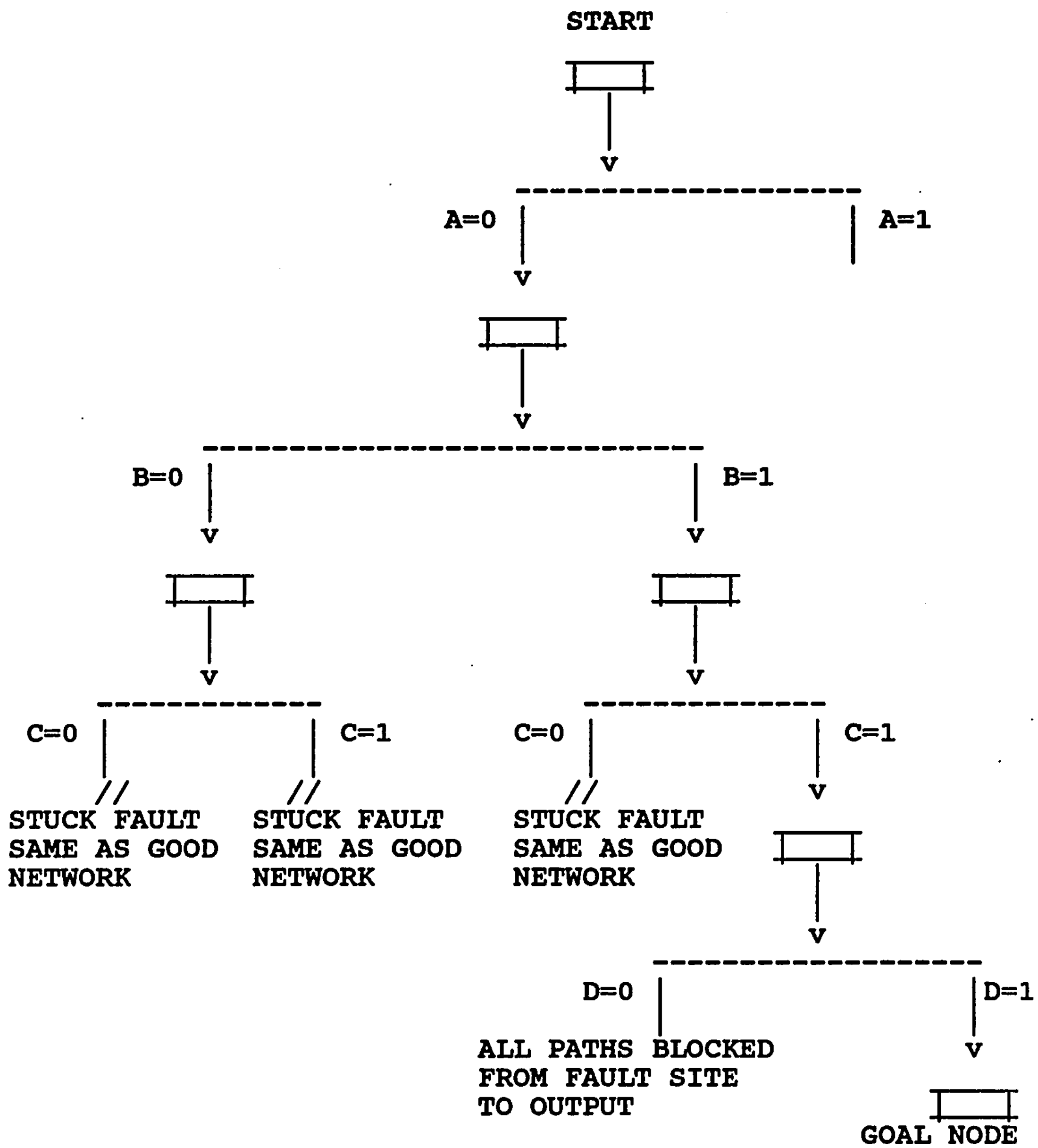


Figure 2.6 PODEM's Decision Tree of Circuit II

## 2.2 Advantages and Disadvantages

One of the advantages of using PODEM is that its implementation is much easier than the D-algorithm because PODEM does not require the consistency drive or line justification procedure as in D-algorithm.

For an n-input circuit, if PODEM rejects the jth input assignment, it will not have to explore the rest of 2 to the power of (n-j) input combinations [4]. If the first input assignment is rejected, this will in effect cut the number of combinations to be considered in half which in turn will cut the simulation time in half.

It is known that the backtracking process takes a lot of time. In order to shorten the simulation time, one of the ways is to reduce the number of backtrackings and the time between backtrackings. PODEM aims at this purpose by assigning values only to primary inputs. Thus, the backtrackings can only occur at the primary inputs. In fact, by this means, PODEM has already reduced the number of nodes. For the D-algorithm, backtrackings can occur at each gate, because values are assigned to internal lines.

Unlike D-Algorithm, PODEM does not need to keep track of the activity vectors and the D-frontier [4], which effectively lowers the memory requirement to run the software.

One drawback of PODEM is that it may leave many X's

(DON'T CARE) in a circuit under test. Since PODEM does not exercise all internal lines of a circuit, many lines are left with X's. As a matter of fact, the more X's are left in a circuit, the less information is known about the circuit. For a single fault detection, this drawback is not very important. For multiple faults detection, a good test vector is the one which detects as many faults as possible, then, the total number of test vectors required to test all faults in a circuit will be reduced. This is especially important in searching test sequences for a sequential circuit. For example, SCIRTSS transforms a sequential circuit to a combinational circuit and then requests either the D-algorithm or PODEM to provide a test vector for a given fault. The input vector to the combinational circuit together with the values at defined primary outputs of the combinational circuit represents a particular state or node of a sequential circuit. SCIRTSS sensitizes the node and then enters propagation search [14]. If SCIRTSS succeeds in both searches, there may be other faults which are trapped after that. If so, SCIRTSS will next skip the time consuming sensitization search and PODEM/D-alogorithm, and go directly to propagate the rest of the trapped faults.

As mentioned above, SCIRTSS or similar type of simulators requests a test vector from PODEM or the D-Algorithm. Actually, SCIRTSS may request up to 5 test vectors from PODEM for a particular node. Under this situation, PODEM will continue to find more test vectors for the same fault with no repeat vector until the specified number of test vectors are found or all input combinations are exhausted. What is so special? The D-Algorithm also has the capability of returning more than one test vector for a fault to SCIRTSS but the D-Algorithm may end up in spending time to find the same test vector repeatly. A test minimization process is required to reduce repeated test vectors in the D-Algorithm. This shows one more time how PODEM is potentially time saving.

# CHAPTER 3

# DATA STRUCTURE

The PODEM routine is called by the BRIDGE routine in SCIRTSS [14], and uses five FORTRAN COMMON blocks to pass information and results to and from SCIRTSS. These five COMMON blocks are COMMON/F/, COMMOM/S/, COMMON/A/, COMMOM/ARRAY/, and COMMON/LOGIC/ [9]. The elements of these five COMMON's are variables of integer type, boolean type or binary type, and arrays of these types. After receiving information from SCIRTSS through the five COMMON's, PODEM processes data in two large arrays. They are the NRY array which is an element of COMMON/ARRAY/ and CNR array which is an element of COMMON/LOGI/. Each element of NRY array is 32-bit wide and is used to store integer type data. The circuit description table and the pointers to the table are stored in the NRY array. Each element of CNR array is 8-bit wide and is used to store binary type data. Any found test vector is stored in the CNR arrays.

## 3.1 COMMON Blocks In Fortran

NRY array is used to implement a link-list data structure that contains information about a circuit under test. The following is a brief explanation of each varible in a COMMON block:

### 3.1.1 COMMON/F/

The pointers of a circuit description table are associated with COMMON/F/.

```
COMMON/F/ xxx(81), NCNT, ITYP, IFAN, ILINK, XXX(2), IOUT,
          IORDER, XXX(4), IFANOUT, XXX(3), NMAX, XXX(25),
          NROUT
```

NCNT total number of elements in a full circuit.

ITYP pointer to element type list. ITYP is firstly initialized to an integer number.

| if element J= | NRY(ITYP+J) |
|---|---|
| AND | 1 |
| OR | 2 |
| NAND | 3 |
| NOR | 4 |
| XOR | 5 |
| not used | 6 - 9 |
| DFF | 10 |
| SCFF | 11 |
| JKFF | 12 |

IFAN pointer to list giving fan-in of each element.

ILINK pointer to list of pointer to lists of inputs of each element.

IOUT pointer to list of primary outputs.

IORDER pointer to list indicating the simulation order of each element.

IFANOUT pointer to list of pointers to fanouts of each element. For element J, NRY(IFANOUT+J) is the pointer to the fanout list. If K=NRY(IFANOUT+J), NRY(K)=number of elements in the fanout list of element J, and the Ith member of the fanout list is located in NRY(K+I).

NMAX maximum number of inputs to an element.

NROUT number of elements in the primary output list.

3.1.2 COMMON/S/

SCIRTSS reserves both ends of NRY array for permanent tables. Only the space between NRY(MARKA) and NRY(MARKB) is allowed to be used by PODEM.

COMMON/S/ XXX(10), MARKA, MARKB, XXX(4), IPRINT

MARKA pointer to the beginning of a segment in NRY array such that the segment is available for PODEM to use.

MARKB pointer to the end of a segment in NRY array such that the segment is available for PODEM to use.

IPRINT SCIRTSS print mode. If IPRINT=2, full debugging output is requested. If IPRINT=1, PODEM prints a circuit partition listing. If IPRINT=0, PODEM only prints founded test vectors if found. Debug mode is helpful to analyze results and to find bugs in a circuits but a debug session for a large circuit may use up an enormous amount of memory storage.

### 3.1.3 COMMON/A/

Information about a fault, that a test vector is going to be found for the fault, is passed from the BRIDGE routine through COMMON/A/ to the PODEM routine. All information returned from PODEM to BRIDGE is also passed through COMMON/A/, or in lists in COMMON/ARRAY/ with pointers in COMMON/A/.

COMMON/A/ JNTER, INTER, FTYPE, JTESTS, NTESTS, NL, LISTIN,
LTESTOR, LOOPL

JNTER — gate number specified by BRIDGE on which a fault occurs.

INTER — the input_number of a gate (not the gate number) on which a fault occurs if the fault is an input stuck fault. Output stuck fault is designated by INTER=0.

FTYPE — fault type. FTYPE=0 if stuck_at_0, FTYPE=1 if stuck_at_1.

JTESTS — number of tests returned by PODEM.

NTESTS — number of tests requested by BRIDGE.

NL — number of elements in a circuit partition (includes primary inputs).

LISTIN — pointer to list of element_numbers of the original circuit in COMMON/ARRAY/ corresponding to the element_numbers of the partitioned circuit. For example, element J of a partitioned circuit corresponds to element NRY(LISTIN+J) of the original circuit.

LTESTOR pointer to list of test vectors in COMMON/LOGI/ found by PODEM. Test 1 is stored in CNR(LTESTOR+1) through CNR(LTESTOR+NL). The location of the next test vectors begins at CNR(LTESTOR+NL+1).

LOOPL time limit set on PODEM by SCIRTSS to avoid very long PODEM execution time in determining an undetectable fault or in finding a test vector for a fault. In the D-Algorithm [9], LOOPL also functions as a timer to limit the D-algorithm in repeatly finding the same test.

### 3.1.4 COMMON/ARRAY/

The dimension of arrays in COMMON/ARRAY/ is up to the users to declare but each word is 4 bytes long in a VAX-11 machine. However, the array must be large enough for PODEM and SCIRTSS to manipulate a circuit; otherwise, the computer system will give a message of access violation error. COMMON/A/ only holds arrays of integer type.

COMMON/ARRAY/ NRY

NRY a one dimension array of integr type (32-bits wide) mainly used to store circuit description table.

3.1.5 COMMON/LOGI/

COMMON/LOGI/ consists of an array of binary type and each word is 1 byte long. The dimension of the array is up to the users to declare but it must be large enough to prevent system access violation.

COMMON/LOGI/ CNR

CNR a one dimension logical array (8-bits wide) mainly used to store test vectors.

## 3.2 NRY Array

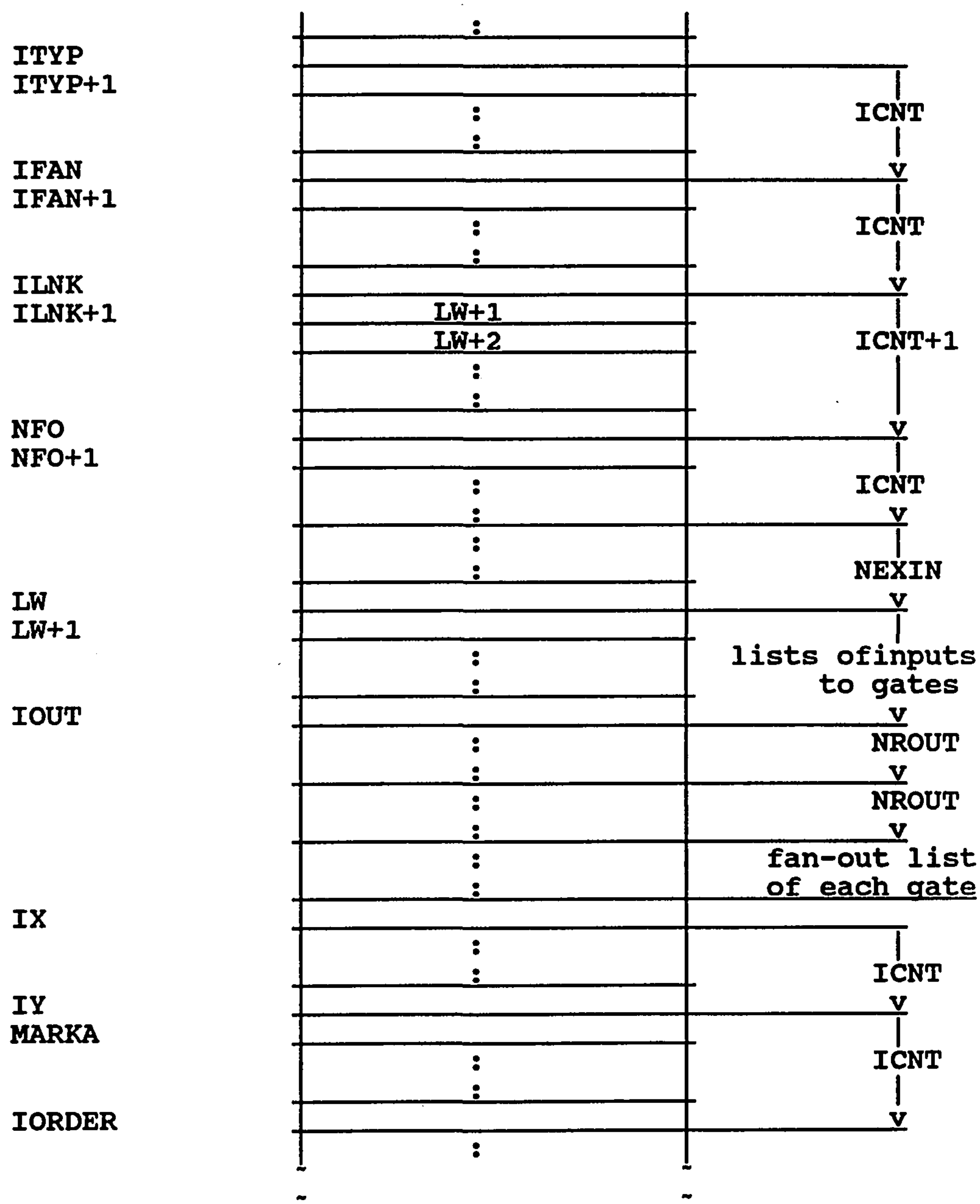


Figure 3.1 Data Structure of NRY Array (Part I)

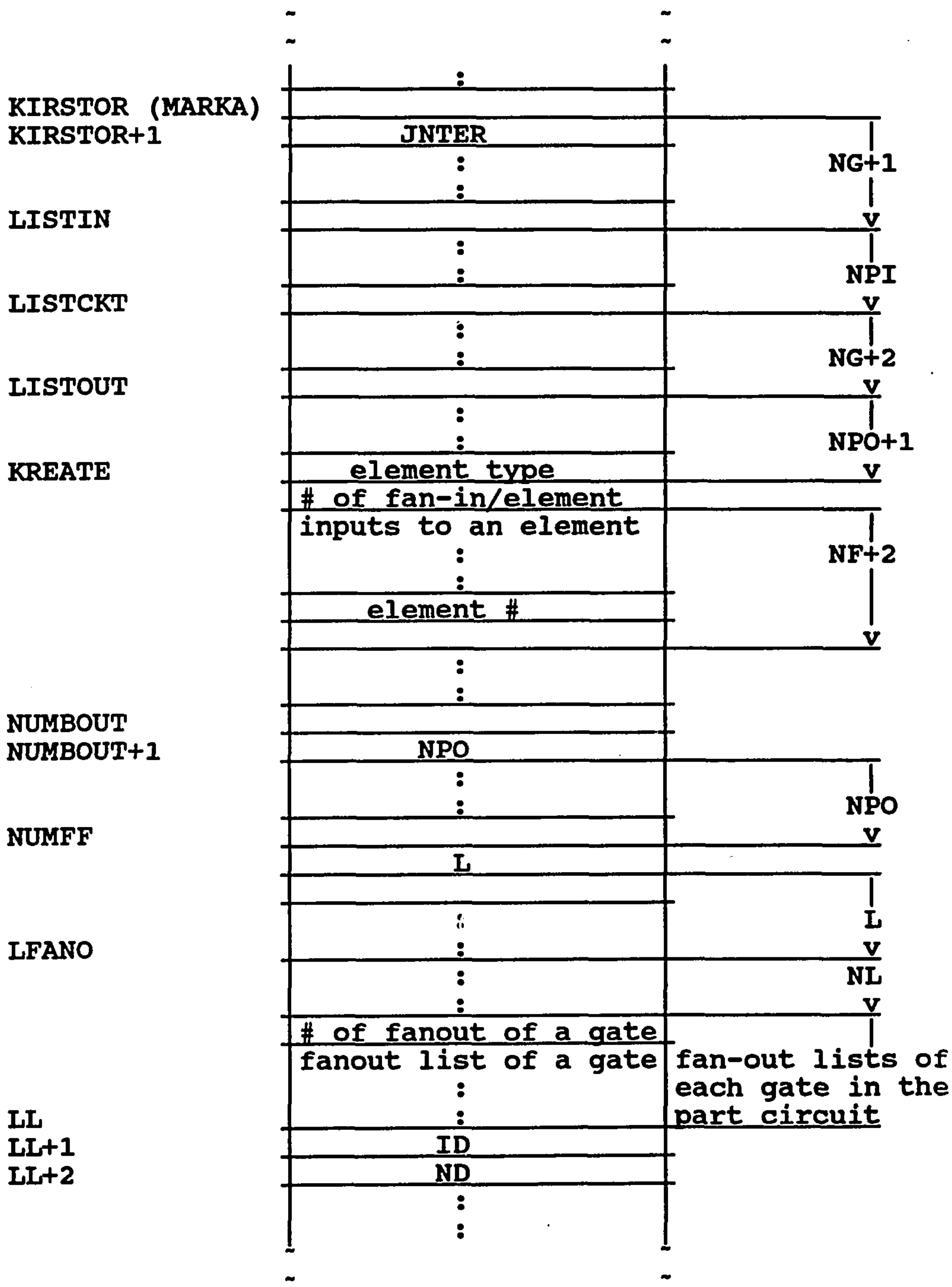


Figure 3.2 Data Structure of NRY Array (part II)

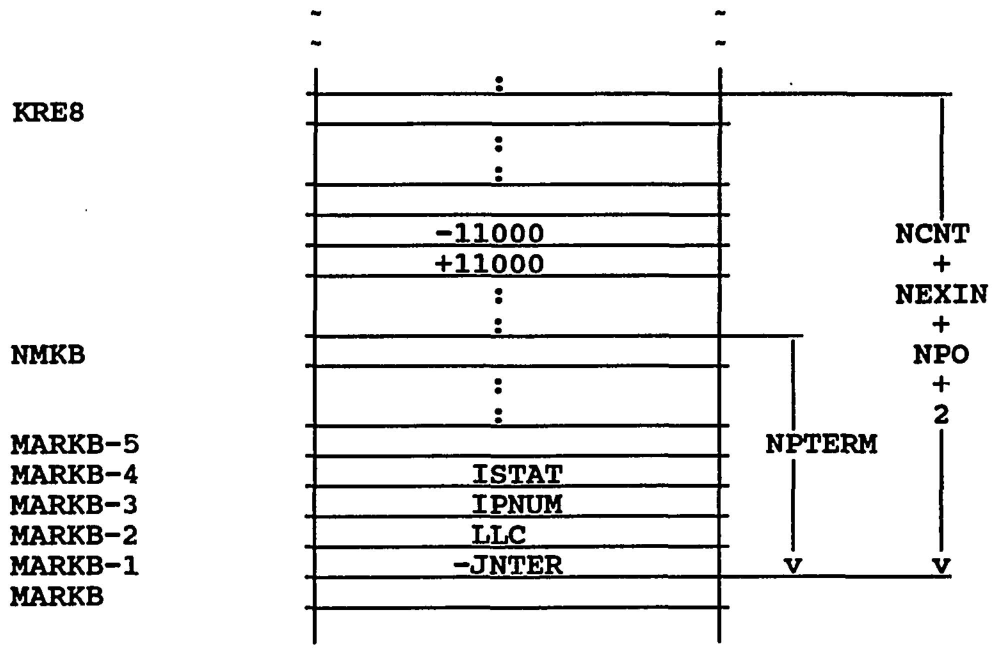


Figure 3.3 Data Structure of NRY Array (part III)

## 3.3 CNR Array

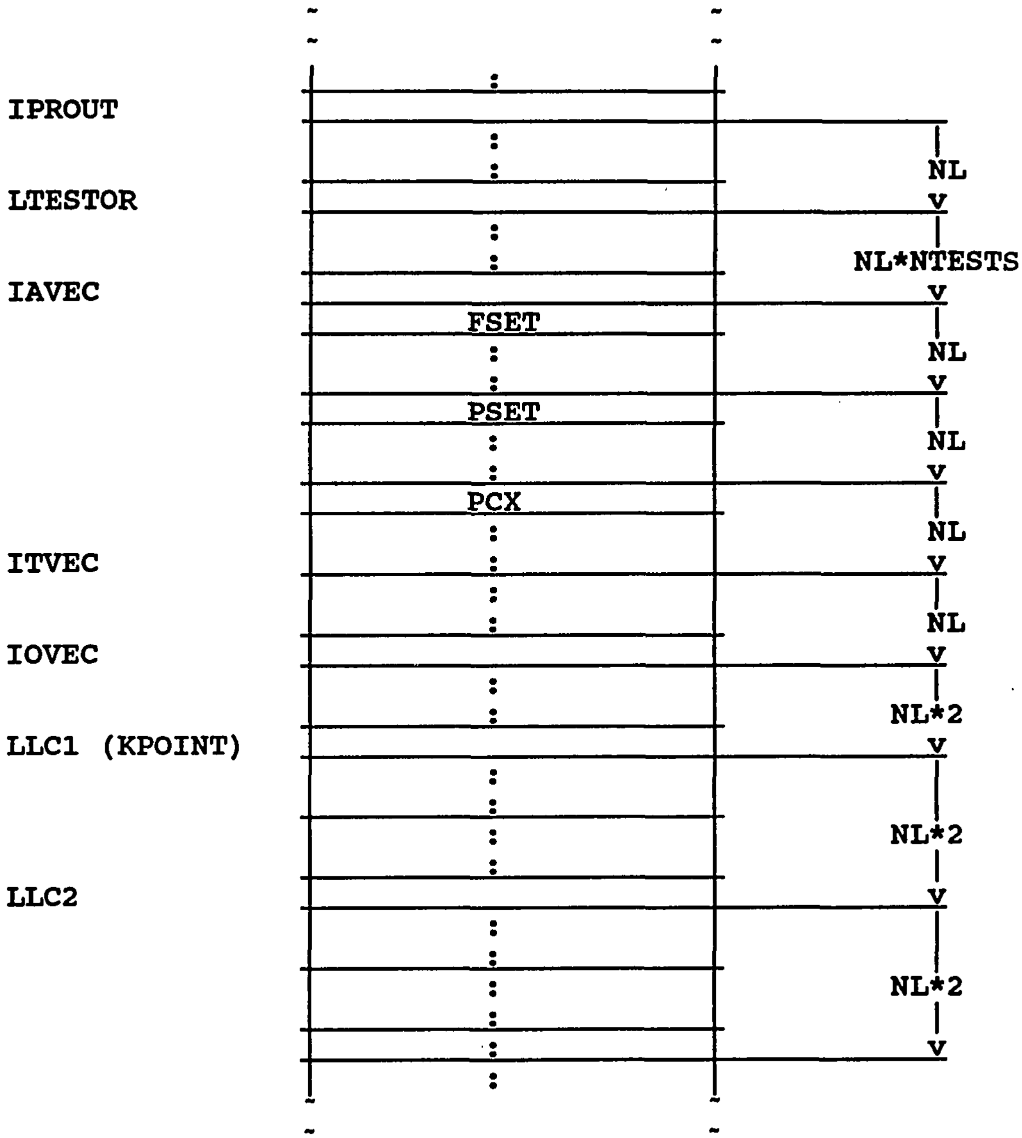


Figure 3.4 Data Structure of CNR Array

## 3.4 Data Structure Differences With Previous Work

An effort is made to use the data structure of the D-algorithm in PODEM. The major reason for doing this is that it ensures PODEM will be compatible to SCIRTSS. It is known that the data structure of the SCIRTSS's D-algorithm [9] is implemented in a large array. PODEM can use the data structure of the D-algorithm with little modification, so PODEM can interface to SCIRTSS exactly the way the D-algorithm does. This eliminates much software interfacing problems. Another reason for doing this is that it makes SCIRTSS have a consistent and unique data structure. At the present time, either PODEM or the D-algorithm, but not both of them at the same time, are available for SCIRTSS to use. If both algorithms, at the same time, are to be made available for SCIRTSS, it will be necessary for the D-algorithm, PODEM, and SCIRTSS to have a common data structure. That is why PODEM trys to use the exact data structure of the D-algorithm in SCIRTSS. PODEM inherits the pattern of the D-algorithm's data structure but may not use it the same way and PODEM may only use some parts of the data structure. That makes the data structure of PODEM a subset of the data structure for D-Algorithm.

PODEM does not have D-frontier and activity vector, it does not use NRY(LL+1) and NRY(LL+2), in Fig. 3.2, the way as D-algorithm uses them. D-Algorithm uses them to indicate the length of a D-frontier list and to index a line in the D-fanout being driven in the forward drive respectively but PODEM uses NRY(LL+1) and NRY(LL+2) to indicate the length of fanout list and the location of the next fanout element. The spaces following NRY(LL+2) store a list of fanout elements in the order from lower priority to higher priority of simulation.

The D-algorithm considers the case of an input line of a gate connecting to a positive power supply, +11000, but it ignores the case of an input line connnecting to a ground. The D-algorithm fails to refer the grounded primary input, -11000, of a partitioned circuit to the corresponding element number of an original circuit in SCIRTSS. PODEM heals this problem by adding one slot for ground primary inputs in the circuit conversion table of NRY array. Fig. 3.3 shows the idea. This additional capability can be added to the D-algorithm by modifying a few programming statements as the way PODEM does. A unified data structure has the advantage that one can modify an algorithm conveniently.

There are two major problems in the implementation of PODEM. The first problem is how PODEM chooses the next input which is going to be assigned a binary value. PODEM solves this problem by using NRY(MARKB-3) to store the value of a variable, IPNUM, that tells which is the next input line to be considered by PODEM. IPNUM is not a variable in the D-algorithm. The second problem is how PODEM knows that it reaches a goal state or a termination state or a keep going state. A goal state is that a fault appears at a primary output; a termination state is that a termination condition of PODEM is satisfied; a keep going state is that PODEM can continue to function. PODEM is composed of several subroutines, some subroutines report the state or status of PODEM. Because this is critical information to PODEM, PODEM assigns the value of a state or status to a variable, ISTAT, and use location NRY(MASKB-4) to store the value of ISTAT. ISTAT is not a variable in the D-algorithm. It is notified that both IPNUM and ISTAT are assigned to fixed locations in NRY array, therefore, each PODEM subroutine can access to this information easily. Although some locations of the NRY array function differently from those in the D-algorithm, the pattern of the data structure is preserved.

It would be best if both IPNUM and ISTAT are stored somewhere in NRY so that they won't alter the original functions of locations NRY(MASKB-3) and NRY(MASKB-4). However, at this phase of implementation, the less optimum way is more convenient.

The above tells the functional and structural differences between the NRY arrays of PODEM and D-algorithm, there are also functional and structural differences between the CNR arrays of these two algorithms. It has been emphasized so far that keeping the pattern of data structure is important. This is also true in CNR array. Like the NRY array, PODEM preserves the pattern of CNR array but uses it in a different way. PODEM does not rely on activity vector or test cube, so it uses IAVEC, ITVAC, IOVAC,and LLC's vectors differently

An important step in PODEM is the activation of a fault site. If a fault site is not activated, there is no need to spend time in the status checking process. When a fault site is activated, PODEM sets FSET, a logical variable, to be true. Since each state of input combination has a FSET value, PODEM makes use of IAVEC to store FSET for each state. The states referred here are the different combinations of input values. There are two other pieces of information which give a more detail description of a state, they are the PSET and PCX. PSET tells whether a sensitized path starts to grow in a state and PCX tells whether the simulation produces CX (DON'T CARE) in any one of the elements in a circuit. Both PSET and PCX are also stored in IAVEC, the idea is shown in Fig. 3.4. Originally, the size of IAVEC is NL. Because of the addition of PSET and PCX, the size of IAVEC is expanded from NL to NL*3. The size of IOVEC is also expanded from NL to NL*2, ITVEC contains the current state of a partition circuit that includes the current line values in the circuit. Current primary output line values are stored in IOVEC. LLC vectors are used to store old states of a partition circuit. Actually, only IAVEC is used differently from the D-Algorithm, the other sections of CNR array are still carrying the same functions as those in the D-algorithm.

# CHAPTER 4

# IMPLEMENTATION

This chapter discusses the implementation of PODEM as a part of an automatic test pattern generator. The simulation technique, storage allocation, logic development, information sharing, modular structure, software interfacing, and limitations in SCIRTSS are the major issues to be addressed in this chapter.

## 4.1 Methodology

A first attempt of using a flow chart approach has been tried but it is found that flow chart is not suitable for a system of size such as PODEM. If PODEM is implemented in a single program, it will result in a very confused data flow system and will always end up with many logical errors. Although a flow chart logic can be refined infinitely, there will still be difficulty for others to understand the implementation and it would also be difficult to make changes or to modify the program. Moreover, the size of a complete flow chart of PODEM would be incredibly large. Nobody can write a completely correct program in the first time, debugging work has to be done before a job is fully finished. Obviously, the flow chart is a bad choice of methodology if heavy debugging work is required. In fact, the more complex an algorithm, the more debugging work is expected. To summarize the disadvantages of choosing a flow chart as a methodology of implementing PODEM, it is decided that another alternative must be explored.

A commonly known approach called top-down design methodology is used in this thesis research. The reasons to use this methodology is that top-down design overcomes the troubles caused by a flow chart as mentioned above. Fig. 4.1 shows the top-down design of PODEM.

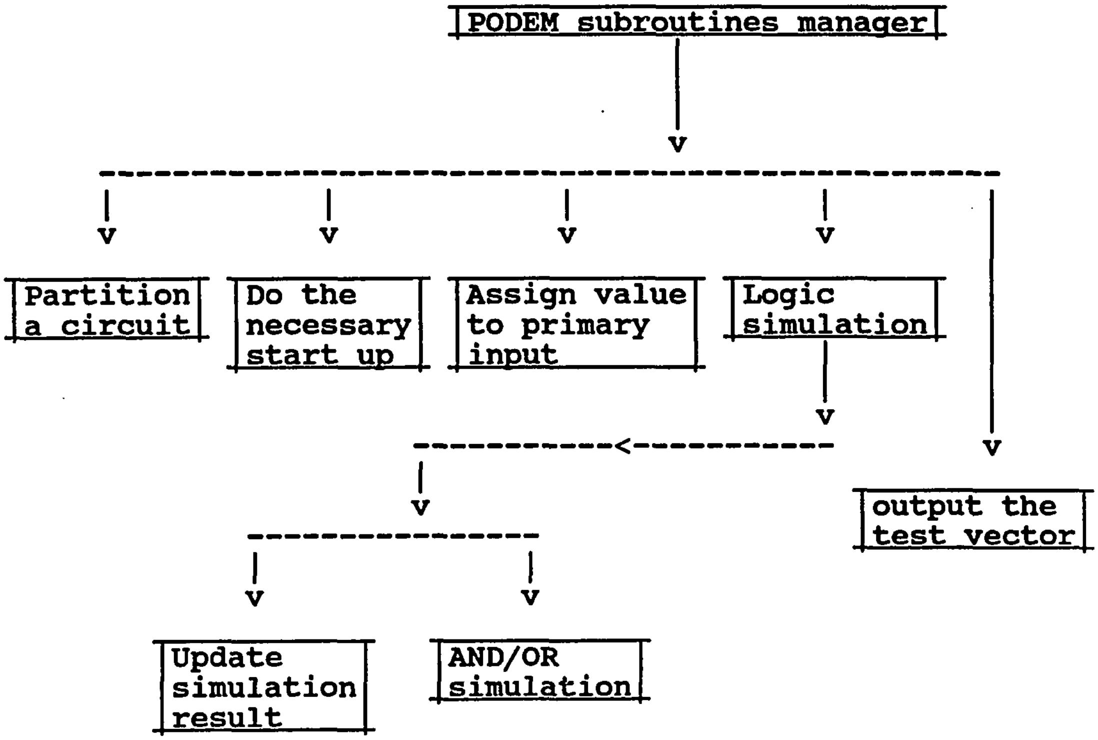


Figure 4.1 Top-down design for PODEM

Although the cost of simulation is cheap, simulation time grows tremendously in a VLSI circuit as the circuit size increases. It is impractical to let a simulation job finish without taking the time factor into consideration. In the case of PODEM implementation, if the program simulates the whole circuit, this will waste much time in simulating same circuit elements but produces the same results as before the simulation. An event driven simulation technique is used in this thesis research to reduce the simulation time. The event driven technique is that an action is taken only when there is a change. PODEM absorbs this technique in its logic simulation routine which only simulates those circuit elements when their present input line values differ from their past input line values. The changes of an input line value is an event and the simulation of a circuit element is a driven. For example, Fig. 4.2 shows the past input line values of each element in a circuit. If the second input (lower) of AND7 is assigned a binary value 0, there is a change to this input so AND7 will be simulated. Now the output of AND7 is 0. Because the present output value of AND7 is the same as its past value, that means there is no change for the input line values of AND10. No further simulation is required for AND8, AND9, AND10, AND11, and AND12. This will save 83% of the simulation time.

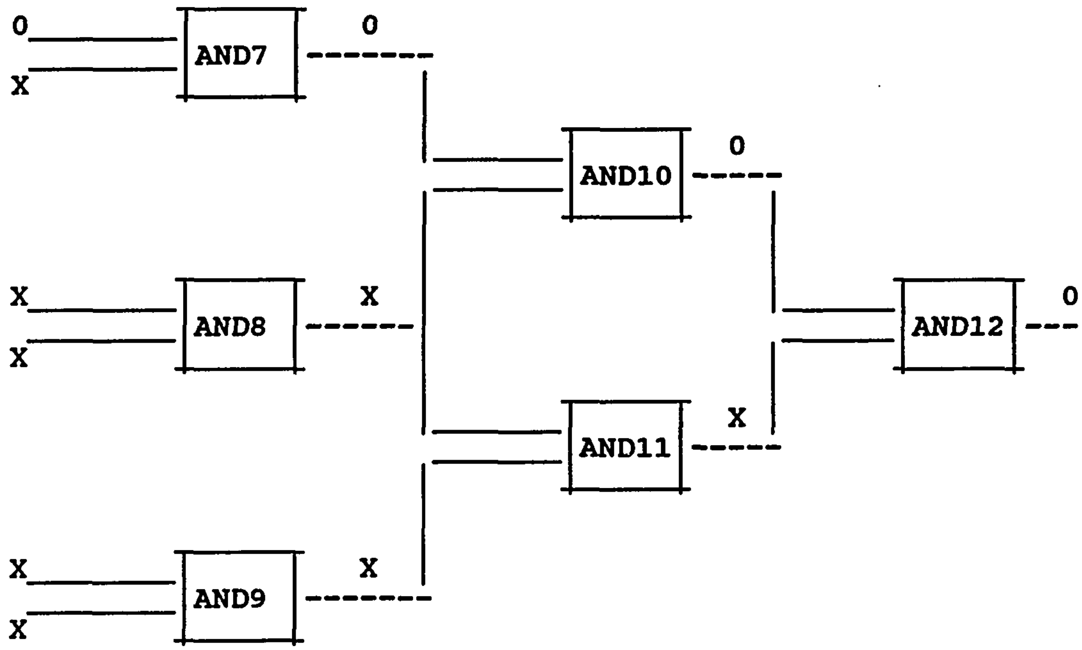


Figure 4.2 CIRCUIT III

After a simulation cycle is finished, the results are retained for later use. In the other words, the state of a circuit is retained. The best possible way to retain a state of a circuit is to store all the internal and external lines' values. If all occured states of a circuit are stored in memory, it may occupy all the memory in a computer system. The issue is how to solve the memory storage problem. Expanding the system memory will raise cost and produce a slower system. Instead of adding hardware to the system, another alternative is to manage the data efficiently in a limited memory system. Not every state of a circuit is useful. If only the useful states are stored, this will save many memory spaces. The following is the mathematical derivation of the maximum amount of memory space required if all states of a circuit are stored.

Definition:

IN = number of primary inputs to a circuit

OUT = number of primary outputs of a cicuit

ELEMENT = number of elements in a cicuit

K = maximum number of inputs to an element in a circuit

STATE = maximum number of states of a circuit

Derivation:

STATE = 2**(IN+1) - 1

memory space required for each state = K*ELEMENT + OUT

maximun memory space = STATE*(K*ELEMENT + OUT)

= [2**(IN+1) - 1]*(K*ELEMENT+OUT)

For example, in Fig. 4.2,

IN=6, OUT=1, ELEMENT=6, K=2,

maximum memory space = [2**(6+1) - 1]*(2*6 + 1)

= 8255 bytes

Actually, the number of useful states are not that many. It is found that the maximum number of useful states for PODEM is equal to the number of primary inputs to a circuit. That is STATE = IN.

Proof:

Suppose (i[1],i[2],...,i[n-1],iI[n]) is an input vector;

n is the number of primary inputs;

i[k] = 0 or 1 for $1 \leq k \leq n$.

Let PREVIOUS_STATE is a function of an designated input;

Let PRESENT_STATE is a function of the combination of inputs.

For $1 \leq k \leq n$;

```
If i[k]=0 then
   PREVIOUS_STATE(i[k])=PRESENT_STATE(i[1],...i[k-1])
Else
   PREVIOUS_STATE(i[k])=PRESENT_STATE(i[1],...,i[k])
```

In the worst case, k=n and i[n]=1

number of previous states = n

=> maximum number of useful states = number of previous states = n

For example, in Fig. 4.2,

IN=6, OUT=1, ELEMENT=6, K=2

STATE = maximum number of useful states

= IN

= 6

maximum memory space = STATE*(K*ELEMENT + OUT)

= 6*(2*6 + 1)

= 102 bytes

% of memory saved = [(8255 - 102) / 8255] * 100%

= 98.8%

Because the data structure of PODEM is implemented in a large one-dimensional array in this thesis research, the suggested memory storage scheme actually shortens the array size required for PODEM to run on a VLSI circuit. Current VLSI circuits have more than 100,000 elements. For example, IN=10, OUT=3, ELEMENT=10,000, and K=2.

maximum memory space = [2**(10+1)-1]*(2*10,000+3)

(old scheme) = 40,946,141 bytes

maximun memory space = 10*(2*10,000+3)

(new scheme) = 200,030 bytes

memory saved = [(40,946,141-200,030)/40,946,141]*100%

= 99.5%

This is an impracticable scheme to declare an array size of, say 40 Mbytes, unless cost and memory speed are not the important factors. The new memory storage scheme requires an array size of 200 Kbytes which is acceptable in available computer sytems.

On the other hand, the D-algorithm cannot use the new memory scheme because it has to store all the possible D-frontiers and activity vectors. Obviously, the D-algorithm has a memory storage problem for an extremely large VLSI circuit. Table 4.1 compares the memory spaces required by the D-algorithm and PODEM to operate on circuits of different sizes. When the number of elements in a circuit greater than a million, it is found that PODEM will not be able to handle such large circuits due to the limitation of computer memory space.

Table 4.1 Comparison of Maximun Memory Space Requirement

| # of elements | # of input | # of output | max. # of input to element | max. space D-algorithm (bytes) | max. space PODEM (bytes) |
|---|---|---|---|---|---|
| 100 | 10 | 5 | 2 | 419,635 | 2,050 |
| 1,000 | 10 | 5 | 2 | 4,104,235 | 20,050 |
| 10,000 | 10 | 5 | 2 | 40,950,235 | 200,050 |
| 100,000 | 10 | 5 | 2 | 409,410,235 | 2,000,050 |
| 1,000,000 | 10 | 5 | 2 | 4,094,010,235 | 20,000,050 |

## 4.2 Subroutines and Functions

The implementation of PODEM in this thesis research uses ten subroutines, two functions and a VAX11/750 system provided subroutine. The names of the ten subroutines are as follows:

1) PODEM
2) PART
3) INITIME
4) INSERT
5) START
6) ASSIGN
7) DGOUT
8) SECOND
9) IMPDR
10) XDUCE

They are stored in files @@@.FOR where @@@ are their corresponding subroutines' names.

The names of the two functions are as follows:

1) ANDF
2) ORF

Both ANDF and ORF are stored in the same file named FUNCTION.FOR.

A VAX11/750 system provided subroutine is used in subroutine INTIME. It is LIB$INIT_TIMER [15].

PODEM is the main subroutine which manages the other subroutines calls. The interrelationships of the subroutines to others will be explained when PODEM subroutine is described. Although there are already many COMMONs used in the implementation of PODEM, those COMMONs are designed for the implementation of D-Algorithm. In order to preserve the data structure of the D-algorithm, an extra COMMON is used to serve for the communications of subroutines designed for PODEM. A Fortran COMMON block called /PODEM/ is declared for this purpose. The following describes how COMMON/PODEM/ works:

COMMON/PODEM/ ISTAT,FTYPE,FSET,PSET,PCX,LL,LLC

ISTAT Integer type variable. It is set by subroutine IMPDR.

If ISTAT=1, that means PODEM reaches its goal state;

If ISTAT=2, that means PODEM has not yet reached its goal state;

If ISTAT=3, that means PODEM reaches one of its termination conditions;

If ISTAT=other integers, that means PODEM reaches an undefined state.

ISTAT is used by subroutine ASSIGN to determine what the next action will be taken for the primary input assignments.

FTYPE A binary type variable. It is assigned by subroutine START.

If the fault to be tested is stuck_at_0, then
FTYPE=C0;

If the fault to be tested is stuck_at_1, then
FTYPE=C1;

It is used by subroutine IMPDR to test the result of simulation of the circuit element where the stuck_at_fault occurs. If the simulation value equals FTYPE at that point, this means that the signal level is same as a faulty signal at the fault site, then PODEM reaches a termination condition, and IMPDR returns ISTAT=3.

FSET Logical variable. It is set to be .TRUE. by subroutine IMPDR if the fault signal at the fault site is activated. IMPDR uses the value of FSET to determine ISTAT.

PSET Logical variable. It is set to be .TRUE. by subroutine IMPDR if any one of the inputs to an simulating element has the fault signal or the complement of fault signal. That means a sensitized path starts to grow. IMPDR uses PSET to determine ISTAT.

PCX Logical variable. It is set to be .TRUE. by subroutine IMPDR if the simulation result of an element is CX. IMPDR uses PCX to determine ISTAT.

LL An integer variable. Every time the subroutine IMPDR is called, a list of the fanout frontier is created. This list is stored in the NRY array pointed by LL.

LLC An integer variable. Each time IMPDR is called, a new state of a circuit may be created. This new state of circuit is stored in CNR array pointed by LLC.

The variables in COMMON/PODEM/ are passed back and forth between subroutine IMPDR and other subroutines. The following sections give detail descriptions of the ten subroutines and the two functions.

### 4.2.1 Subroutine PODEM

This section describes the interrelationships between subroutines. PODEM is the central subroutine that calls other subroutines to finish the job of the PODEM-Algorithm. PODEM starts by calling subroutine PART which prunes the original circuit to a smaller size circuit and puts the pruned circuit in the NRY array. Then, subroutine START is called to perform initialization and setup of the NRY and CNR arrays. PART and START are called only once in PODEM for an injected fault. The other subroutines may be called more than one time in PODEM in order to find the requested number of test vectors for an injected fault. If another fault is injected, PART and START will be called again.

The next subroutine to be called is ASSIGN. ASSIGN finds the next input combination and returns it to PODEM; Otherwise, ASSIGN signals to PODEM that no more input combination is possible to be a test vector for a fault. If no input combination is possible, PODEM will return the number of test vectors found so far and then exits to its calling program. If an input combination is found, PODEM will retrieve FSET, PSET and PCX for the previous state and then will enter subroutine IMPDR. IMPDR gets the input combination and simulates the circuit. The result of the simulation tells whether the input combination is a test vector, the input combination causes the termination condition or the input combination is not a test vector. IMPDR returns the result of the simulation by passing the values of ISTAT, FSET, PSET and PCX to PODEM and then PODEM analyzes the simultion result by testing the conditions of ISTAT, FSET, PSET, and PCX. If ISTAT=1, a goal state is reached, PODEM will enter subroutine XDUCE. XDUCE stores the input vector in CNR array as a test vector for a fault. After XDUCE exits to PODEM, PODEM checks the number of test vectors found. If the requested number of test vectors are found, PODEM will exit to its calling program; otherwise, PODEM will enter ASSIGN to start a new cycle of finding more test vectors. IF ISTAT=2, that means the input combination is not a test vector, PODEM will store FSET, PSET, and PCX in IAVEC of CNR array and then will enter ASSIGN to start a new cycle of test vector finding. If ISTAT=3, that means termination condition has been reached; PODEM will examine the conditions of PSET and PCX to determine what type of termination is reached. If PSET=.TRUE. and PCX=.FALSE., then it means all signal paths are blocked, otherwise, the signal level is same as a fault signal at a fault site. In the case of ISTAT=3, PODEM will not enter XDUCE to store the input combination but will enter ASSIGN to start a new cycle of finding a test vector.

Each time before ASSIGN is called, PODEM makes sure the requested number of test vectors have not been found. PODEM will not enter ASSIGN if the requested number of test vectors are found. There is also a time limit to be set on PODEM for it to find the requested number of test vectors. If the time limit is exceeded, PODEM will return to its calling program. Because ASSIGN and IMPDR are time consuming subroutines, a time limit test is taken after each of these subroutines is called and also before ASSIGN is called.

The subroutine DGOUT is called at several places in PODEM if the variable FBUGS=.TRUE.. FBUGS is set in PODEM's calling program. DGOUT prints all the line values of a pruned circuit on the screen and stores it in a designated output file. This subroutine is useful for debugging the software package and for observing the behavior of a circuit under test.

Finally, PODEM returns the number of test vectors found and the test vectors to its calling program.

### 4.2.2 Subroutine PART

PART is a similar subroutine in NDALG [9]. This subroutine prunes the whole circuit provided by SCIRTSS or from the circuit wiring list of the Stage III Compiler. PART is the largest subroutine. There are some differences between PART in PODEM and PART in NDALG. The PART in NDALG does not form fanout lists for primary input lines of a circuit because it does not consider the case that faults could occur at a primary input line. Practically, this case will happen. In the new version of PART in PODEM, the fanout lists of primary inputs are added to the circuit description list of a circuit in NRY array. Moreover, NDALG's PART does not consider the case that a line in a circuit may connect to ground, -11000. When the circuit wiring list from the Stage III Compiler has the description of a line which is connected to ground, -11000, NDALG's PART [9] does not capture this information and results with an incorrect circuit description list in the NRY array.

Consequently, any test vectors found for a fault may not be the test vectors for the fault in the original circuit. PODEM's PART corrects this mistake by adding this case to the internal index conversion table in the NRY array, Fig. 3.3 shows that the ground case, -11000, is added with the +5 volt case, +11000. As a result, the new version of PART works more optimally but takes more time. This prolongation of the PODEM time must be taken into consideration when the efficiencies of PODEM and NDALG are evaluated.

#### 4.2.3 Subroutine INITIME

This subroutine calls VAX/VMS system library subroutine LIB$INIT_TIMER [15] which stores the current values of specified times and counts for use by LIB$STAT_TIMER [15].

#### 4.2.4 Subroutine INSERT

INSERT superimposes value L onto the CNR array at location CNR(I+J). I is a pointer to a LLC vector in CNR and J is an index to the LLC vector. INSERT starts by examining the content at location CNR(I+J). If CNR(I+J)=L, then no action will be taken. If CNR(I+J)=CX, L will superimposes CX at CNR(I+J). If CNR(I+J) does not equal to CX or L, that means a conflict of signal occurs and INSERT will set the flag NOGO to notify its calling routine that something is wrong. L, I, and J are parameters of INSERT. INSERT is called by subroutine IMPDR. Fig. A.6 and A.7 show the relationship of INSERT and IMPDR.

### 4.2.5 Subroutine START

The basic functions of subroutine START are to do the initialization and set up of the NRY and CNR arrays. The routine first calculates a time limit which is used for the collapsed time calculation. Although the NRY array has been initialized in subroutine PART; PODEM uses NRY differently from the D-algorithm. Some elements of NRY have to be set to some other initial values. PART sets up the NRY array but it does not set up the CNR array. This job remains for START to finish. START sets up the CNR array with pointers LTESTER, IAVEC, ITVEC, IOVEC, and LLC. Fig. 3.4 shows that these pointers point to different locations in the CNR array. The ideas of these pointers are explained in Chapter 3. START initializes IAVEC to be .FALSE. that means FSET, PSET and PCX of all possible states are initialized to be .FALSE.. The first LLC vector is initialized to CX. If there are any lines which are permanently tied to ground (-11000) or +5 volts (+11000), START will superimposes C0 or C1 to the corresponding locations of those lines in LLC vector. The next step is to superimpose fault to the fault location in LLC vector. The last thing to do is to set all elements in the first test cube ITVEC to be CX's.

### 4.2.6 Subroutine ASSIGN

ASSIGN is a subroutine to determine the possible input vectors which can become the test vectors for a fault in a circuit. There are numerous ways to determine what value is assigned to which primary input line. The approach in this research is that primary input lines are arranged in order and the value of 0 is tried firstly. If the assignment of 0 to a primary input line causes termination condition, ASSIGN will tried the value 1 for the same input line. For example, Fig. 4.3 shows how primary inputs are arranged in order.

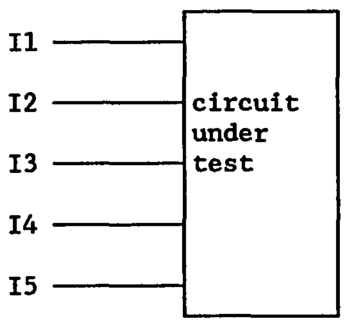


Figure 4.3 Primary Input Ordering

The primary inputs are arranged in the order from I1 to I5.

ASSIGN starts by retrieving the values of three important variables, IPNUM, ISTAT, and IPVAL from main routine PODEM. IPNUM tells which primary input is assigned last time. ISTAT reports the result of last simulation. IPVAL tells what value is assigned to the last primary input. For ISTAT=1, that means the last simulation reached goal state, if the last value to the last input is C0, then C1 will be assigned to the last input again. Along the discussion of this section, (I1,I2,I3,I4,I5) represents an input vector and each input has one of the three values: 0, 1, and X. For example, ISTAT=1, IPNUM=3, IPVAL=0, the last input vector is (0,0,0,X,X), and the next input vector is (0,0,1,X,X). Fig. 4.4 describes the ASSIGN algorithm. A point to be emphasized is that when an input line is permanently tied to ground (-11000) or +5 volts (+11000), ASSIGN skips the line and goes to other non permanently tied line. In Fig. 4.4, steps [2.1.1] and step [2.2.1] search for another input line such that the input line is not permanently tied. Moreover, step [2.1.1] searchs backward until a suitable input line is found or no more input line is available. When an input line is found and a value is assigned to the input, this means an input vector is found and this input vector is saved in ITVEC. Next, the state of a circuit corresponding to that input vector is tranferred from LLC to ITVEC in the CNR array.

For example,

last input vector =(0,1,0,1,1,1)

ISTAT=2

IPNUM=6

IPVAL=1

mechanism:

```
(0,1,0,1,1,1)
      |
      v
(0,1,0,1,1,X)
      |
      v
(0,1,0,1,X,X)
      |
      v
(0,1,0,X,X,X)
      |
      v
(0,1,X,X,X,X)
      |
      v
(0,1,1,X,X,X)   input vector found
```

Another example,

last input vector = (0,1,0,X,X,X)

ISTAT = 2

IPNUM = 3

IPVAL = 0

mechanism:

```
(0,1,0,X,X,X)
       |
       v
(0,1,0,0,X,X)   input vector found
```

```
[1]  get last status, input_line, input_value
[2]  if status=GOAL, then
     [2.1]  if input_value=0, then
                 set input_value=1
               else
     [2.1.1]     (go back to an input_line
                  which has input_value=0)
     [2.1.2]     record the input_line
     [2.1.3]     set input_value=1

     else if status=NON_GOAL, then
            if input_line is the last line, then
               go to [2.1]
              else if input_value=0, then
                 go to [2.1]
            else
     [2.2.1]     take the next input_line
                   set input_value=0

     else if status=TERMINATION, then
            go to [2.1]

  else
        undefined status, program STOP

[3]  end
```

**Figure 4.4 ASSIGN Algorithm**

#### 4.2.7 Subroutine DGOUT

This subroutine prints the line values of a circuit under test to an output file according to the specified debugging mode. There are four types of debugging modes to be chosen and the subroutine parameter of DGOUT specifies which debugging mode of user wants to use. The four debugging modes are summaried as below:

| Debugging mode | Printing action |
|---|---|
| 1 | Inputs (Flip-flops included) |
| 2 | Entire circuit |
| 3 | Flip-flops only |
| 4 | Outputs |

Table 4.2 Debugging Modes

The dispayed code character associated with each line value in ITVEC is stored in th IPROUT list. Table 4.3 shows the character codes.

| Line Value | Display Character | Internal Code |
|---|---|---|
| C0 | 0 | 00 |
| C1 | 1 | FF |
| CD | D | 69 |
| CE | E | 96 |
| CX | X | 80 |

Table 4.3 Diplay Characters For the Debugging Mode

### 4.2.8 Subroutine SECOND

SECOND returns the collapsed CPU time in seconds. It starts by calling the VAX/VMS system library routine LIB$STAT_TIMER [15] which returns a time in 10 millisecond increments. This time is multiplied by 100 and then minuses T1, a parameter of SECOND, the result equals to the collapsed CPU time since T1.

### 4.2.9 Subroutine IMPDR

IMPDR is a routine that simulates a circuit element and examines simulation result according to the specification of the PODEM algorithm. This routine is only called by subroutine PODEM which is the main routine of the implementaion of PODEM algorithm. The status returned by IMPDR provides important information on whether a goal state is reached and gives enough information to ASSIGN routine to find the next input vector. The algorithm of IMPDR is shown in Fig. 4.5.

IMPDR starts by identifying whether a fault is an input stuck or output stuck fault. The result of this identification is used to check signal level in step [2.3] of Fig. 4.5. Next, IMPDR gets a primary input number and forms a fanout list of the primary input in step [1] of Fig. 4.5. IMPDR takes out the first element of the fanout list and simulates that element. To simulate an element, IMPDR gets the inputs of the element from the circuit decription table in the NRY array and branches to a routine which will simulate the element. There are six simulation routines are available in IMPDR. They individually correspond to the simulation of JK flip_flop, SC flip_flop, D flip_flop, AND gate, OR gate, and XOR gate. The simulations of NAND and NOR gates are by treating NAND as AND and NOR as OR, and inverting the simulated value. Actually, eight types of circuit elements can be simulated. The five value (C),C1,CD,CE,CX) operations for OR gate, AND gate, and XOR gate are shown in Table 4.4a, Table 4.4b, and Table 4.4c respectively. The algorithms of the five value operations of SCFF, DFF, and JKFF are shown in Fig. 4.6a, 4.6b, and 4.6c respectively.

IMPDR only simulates those elements where input line signals are changed; otherwise, the next element in the fanout list will become the next element to be simulated. This event driven technique saves a large amount of time from simulating those elements that produce the old values. If the element just simulated is the faulty element and the simulated value equals the faulty signal, this means a termination condition is reached, IMPDR will exits to its calling routine. If the simulated element is a primary output element, IMPDR will simulate the next element in the fanout list; otherwise, a new fanout list will be formed by concatenating the old list to the fanout of the element and the first element of the new fanout list will be simulated. If the simulated element is not the faulty element, IMPDR determines whether a sensitized path starts to grow and the fault is propagated to the primary outputs. The boolean variables FSET, PSET, and PCX are set to indicate whether a fault is activated, a sensitized path grows, and all paths are blocked. When all elements in the fanout list have been simulated, IMPDR examines the values of FSET, PSET and PCX to determine the status of this cycle of simulation. There are three possible status:

1. GOAL_STATE
2. NO GOAL_STATE
3. TERMINATION_STATE

The resulting status is used by the ASSIGN routine to determine the next input vector.

```
[1]    form the fanout list of a primary input
[2]    while fanout list not empty, do

   [2.1]   pop fanout list
   [2.2]   simulate the popped element
   [2.3]   if (signal level=faulty signal
                at fault site), then
              TERMINATION_STATE is reached
              go to step [4]
           else if simulated value is unchanged, then
              go to step [2]
           else
   [2.4]      form a new fanout list of the element
   [2.5]      concatenate new and old fanout lists
              go to step [2]

[3]    if fault propagated to primary ouput, then

   [3.1]   if fault site is activated, then
              GOAL_STATE is reached
           else
              GOAL_STATE is not reached

       else if all paths are blocked, then
           TERMINATION_STATE is reached
       else
           GOAL_STATE is not reached

[4]    return the state of simulation
[5]    end
```

Figure 4.5 Subroutine IMPDR Algorithm

| | C0 | C1 | CD | CE | CX |
|---|---|---|---|---|---|
| C0 | C0 | C1 | CD | CE | CX |
| C1 | C1 | C1 | C1 | C1 | C1 |
| CD | CD | C1 | CD | C1 | CX |
| CE | CE | C1 | C1 | CE | CX |
| CX | CX | C1 | CX | CX | CX |

**Table 4.4a OR Gate Five Value Logic**

| | C0 | C1 | CD | CE | CX |
|---|---|---|---|---|---|
| C0 | C0 | C0 | C0 | C0 | C0 |
| C1 | C0 | C1 | CD | CE | CX |
| CD | C0 | CD | CD | C0 | CX |
| CE | C0 | CE | C0 | CE | CX |
| CX | C0 | CX | CX | CX | CX |

**Table 4.4b AND Gate Five Value Logic**

| | C0 | C1 | CD | CE | CX |
|---|---|---|---|---|---|
| C0 | C0 | C1 | CD | CE | CX |
| C1 | C1 | C0 | CE | CD | CX |
| CD | CD | CE | C0 | C1 | CX |
| CE | CE | CD | C1 | C0 | CX |
| CX | CX | CX | CX | CX | CX |

**Table 4.4c XOR Gate Five Value Logic**

```
[1]  if IVC = CX, then
        IYY = IVY
[2]  if IVA = IVB = 0, then
        IYY = IVY
[3]  if IVA = IVB <> C0, then
        NOGO = .TRUE.
[4]  if {[1], [2], [3] are .FALSE. and
         (IVA=CX or IVB=CX, IVY=CX)}, then
        IYY = CX
     else
        IYY=(IVA.OR.IVY).AND.(.NOT.IVB).AND.IVC
[5]  end
```

**Figure 4.6a SC Flip_flop Simulation Algorithm**

```
[1]  if IVC=CX and IVA=IVY, then
        IYY=IVY
     else if IVC=CX or IVA=CX, then
        IYY=CX
     else
        IYY=IVA.AND.IVC.OR.IVY.AND.(.NOT.IVC)
[2]  end
```

Figure 4.6b   D Flip_flop Simulation Algorithm

```
[1]  {IYY=((.NOT.IVY).AND.IVA.OR.IVY.AND.
      (.NOT.IVB)).AND.IVC.OR.IVY.AND.(.NOT.IVC)}
[2]  if IYY = IVY, then
        IYY = IVY
     else if IVC = ICX, then
        IYY = CX
     else if IVA = IVB = CX, then
        IYY = CX
     else if IVA = IVY = CX, then
        IYY = CX
     else if IVB = IVY = CX, then
        IYY = CX
[3]  end
```

Figure 4.6c   JK Flip_flop Algorithm

#### 4.2.10 Subroutine XDUCE

XDUCE transfers the primary input vector from ITVEC to a location pointed by LTESTER. XDUCE is called only after a test vector is found by PODEM. In the case of testing a sequential circuit, the output value of a memory element equals to a corresponding primary input value in the same state. When a memory element is simulated, its output value changes. In order to recover to its previous state's value, XDUCE sets the output of a memory element equal to its corresponding primary input value. Fig. 4.7 shows the procedure. XDUCE reports the number of test vectors found and updates the pointer LTESTER to store the next test vector.

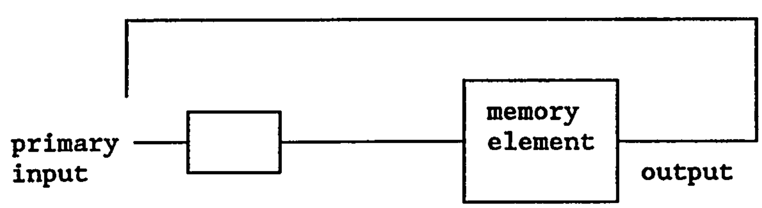


Figure 4.7 Recovery of Memory Element Output

#### 4.2.11 Functions ANDF & ORF

ANDF and ORF are functions which are stored in file FUNCTION.FOR. ANDF performs logical AND operation on two operands. An operand is one of the five external representations which are C0, C1, CD, CE and CX. The internal hexidecimal values of the five external representations are 00, FF, 69, 96, and 80 respectively. They are summaried as below:

| | | Binary | Hex |
|---|---|---|---|
| C0(0) | : | 00000000 | 00 |
| C1(1) | : | 11111111 | FF |
| CD(D) | : | 01101001 | 69 |
| CE(E) | : | 10010110 | 96 |
| CX(X) | : | 10000000 | 80 |

Table 4.2. The Internal Expression of PODEM

Normal AND operation on two of the five values does not produce a value which belongs to the set of five values. However, five values logic is the core of logical operations in PODEM, this is why normal AND and OR operations cannot be used. Instead, modified versions of AND and OR functions are implemented. They are ANDF and ORF. For example, if x=80, y=69, then (x .AND. y)=00, but, ANDF(x,y)=80. According to the five values logic Table 4.4b, (CX .AND. CD)=CX.

## 4.3 Before Interface To SCIRTSS

Before the subroutines in Chapter 4.2 are interfaced to SCIRTSS, it is necessary to make sure each of these subroutines works correctly in SCIRTSS like environment. For proper operation, the subroutines need a circuit description table and fault information. It is desired that there is a simple program which can capture the circuit wiring list of the Stage III Compiler and supply a circuit description table to PODEM. SPODEM is designed to do this job. SPODEM reads data from the output of Stage III Compiler. It reads in description of the network topology, determines the fan-in of each element, sets up a fanout connection list, establishes a simulation sequence ordering relationship, requests fault information from the user, and accepts test vectors and information returned from PODEM. A Fortran "include file" called TEMP.FOR is required for SPODEM and other subroutines to identify the output file's name. There is debugging capability built with SPODEM and other subroutines. The functions of this debugger are to help users to determine the correctness of the program step by step, and to tell users the location of a bug, and to give users a view of signals of internal lines of a circuit during PODEM operation. Figure A.7 shows the interfacing of PODEM to SPODEM. Figure A.4 gives a demonstration of how to run SPODEM.

## 4.4 Interface to SCIRTSS

After a thorough test of PODEM subroutines in Chapter 4.3, these subroutines can now be interfaced to SCIRTSS. There is a subroutine in SCIRTSS calls the D-Algorithm, it is the DALSEN subroutine. The Fortran statement makes such subroutine call is as follow:

CALL DALG

DALG is the name of D-Algorithm subroutine [9]. To make the interfacing easier, PODEM subroutine is rename as DALG so that SCIRTSS remains unchanged. Fig. A.6 show the interfacing of PODEM subroutines to SCIRTSS. Fig. A.2 shows how to link SCIRTSS with PODEM in a VAX/VMS system and Fig. A.3 gives a demonstration of how to run PODEM in SCIRTSS.

## 4.5 Limitation of SCIRTSS

SCIRTSS version 4.0 can handle 999 flip_flops, 10,000 gates, 200, inputs, 200 outputs and 10 modules. The sizes of the NRY and CNR arrays needs to be 1,000,000 and 2,000,000 respectively [1,14].

# CHAPTER 5

## EXPERIMENTAL RESULTS

The results from the depth first D-algorithm, DF3, implemented by Y. H. Chen [13] is used to compare with the time efficiency of PODEM Algorithm implemented in this research. Both the DF3 and PODEM programs work independently from SCIRTSS. There are two versions of SCIRTSS which use DF3 and PODEM as subroutines. SCR3 [13] is SCIRTSS with a DF3 capability. SPODEM is SCIRTSS with a PODEM capability. Three combinational circuits, SELECTOR, 8-Bit Look Ahead Adder [2] and 8-Bit Incrementer [2], and a sequential circuit M5 Machine [3] are circuits under test in this thesis. Selector and the 8-Bit Look Ahead Adder are circuits with more reconverging fanouts. The 8-Bit Incrementer and the M5 Machine are circuits with less reconverging fanouts. DF3 and PODEM are run on these four circuits with information on each individual fault entered from terminal. The CPU times returned from these two program are used to compare the time efficiencies of the D-Algorithm and the PODEM algorithm on circuits with different amount of reconvergent fanouts. Next, SCR3 and SPODEM are run on the four circuits to determine total running time, fault coverage, length of input sequence and number of undetectable faults. The order of magnitude is used as the scale to compare time efficiency of each of these algorithms.

## 5.1 SELECTOR

SELECTOR is a combinational circuit with a fair amount of reconvergent fanout. Its AHPL description, the parameter file used, and its wiring description are in Fig. B.1, Fig. C.1, and Table D.1 respectively.

Firstly, DF3 and PODEM are run on SELECTOR with the users injecting each individual fault. It is noticed that the CPU times used by PODEM to find a test vector for an injected fault is an order of magnitude less than the time used by DF3. For gate 3, input 1 s_a_0, PODEM used 0.04 seconds but DF3 used 0.35 seconds. For gate 14, input 2 s_a_1, PODEM used 0.34 seconds to determine the fault is undetectable but DF3 used 8.13 seconds. Table 5.1a summarizes the CPU time used by PODEM and DF3 on different faults. Table 5.1b summarizes the external input vectors found by DF3 and PODEM for different faults. No significant conclusion is drawn from the results in this table. Supposingly, PODEM should leave more DON'T CAREs in a circuit, but the experimental results do not show much difference between the number of DON'T CAREs left by DF3 and the number of DON'T CAREs left by PODEM. Table 5.1c summarizes the number of DON'T CAREs.

Second, SCR3 and SPODEM are run on the SELECTOR. Table

5.1d summarizes some statistical results. It is found that the total running time of SPODEM is a order of magnitude less than the running time of SCR3 and the number of input sequences generated by SPODEM are fewer than those generated by SCR3. Because SELECTOR has more reconvergent fanouts, SPODEM works faster than SCR3 in this example. SCR3 and SPODEM give the same percentage of fault coverage of 96.2%. Faults of gate 1, output s_a_1 and s_a_0 are undetectable because this gate is a control flip flop. Because AHPL operates in the clock mode, a control flip flop is automatically added to the AHPL description of a combinational circuit. In fact, these two faults do not exist. So, the only the fault, input 2 of gate 14 s_a_1 is undetectable. In Fig. 5.1, for the fault of s_a_1 at input 2 of gate 14, it requires both primary inputs 11005 and 11006 to be 0; however, this assignment would not allow the fault to propagate through gate 25, gate 26, gate 27 and gate 28.

| Gate | Input | Stuck_at | DF3 Computing Time (sec) | PODEM Computing Time (sec) |
|---|---|---|---|---|
| 3 | 1 | 1 | 0.35 | 0.04 |
| 4 | 0 | 0 | 0.20 | 0.07 |
| 14 | 2 | 1 | 8.13 **** | 0.34 **** |
| 17 | 1 | 1 | 0.21 | 0.08 |
| 19 | 0 | 1 | 0.20 | 0.05 |
| 24 | 1 | 1 | 0.21 | 0.09 |

Table 5.1a CPU Time --- SELECTOR

| Gate | Input | Stuck_at | External Input Vector | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 3 | 1 | 1 | XX11X0 | XXXX00 |
| 4 | 0 | 0 | X01111 | XXXX01 |
| 14 | 2 | 1 | **** | **** |
| 17 | 1 | 1 | 011111 | 00XX11 |
| 19 | 0 | 1 | X01111 | XXXX00 |
| 24 | 1 | 1 | XXX110 | 000110 |

Table 5.1b External Input Vectors --- SELECTOR

| Gate | Input | Stuck_at | Number of DON'T CAREs In A Partition Circuit | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 3 | 1 | 1 | 3 | 4 |
| 4 | 0 | 0 | 9 | 8 |
| 14 | 2 | 1 | **** | **** |
| 17 | 1 | 1 | 4 | 9 |
| 19 | 0 | 1 | 9 | 8 |
| 24 | 1 | 1 | 20 | 0 |

Table 5.1c Number of DON'T CAREs --- SELECTOR

Note: 1) "****" means a fault is undectectable

| SELECTOR | SCR3 | SPODEM |
|---|---|---|
| # of gates | 32 | 32 |
| # of flip flops | 1 | 1 |
| # of external inputs | 6 | 6 |
| total # of faults | 78 | 78 |
| # of faults detected | 75 | 75 |
| % of fault coverage | 96.2 | 96.2 |
| Running time (sec) | 240.94 | 24.83 |
| # of input sequences | 26 | 24 |
| Faults not detected | 1 output SA1<br>1 output SA0<br>14 input 2 SA1 | 1 output SA1<br>1 output SA0<br>14 input 2 SA1 |

**Table 5.1d Statistical Summary --- SELECTOR**

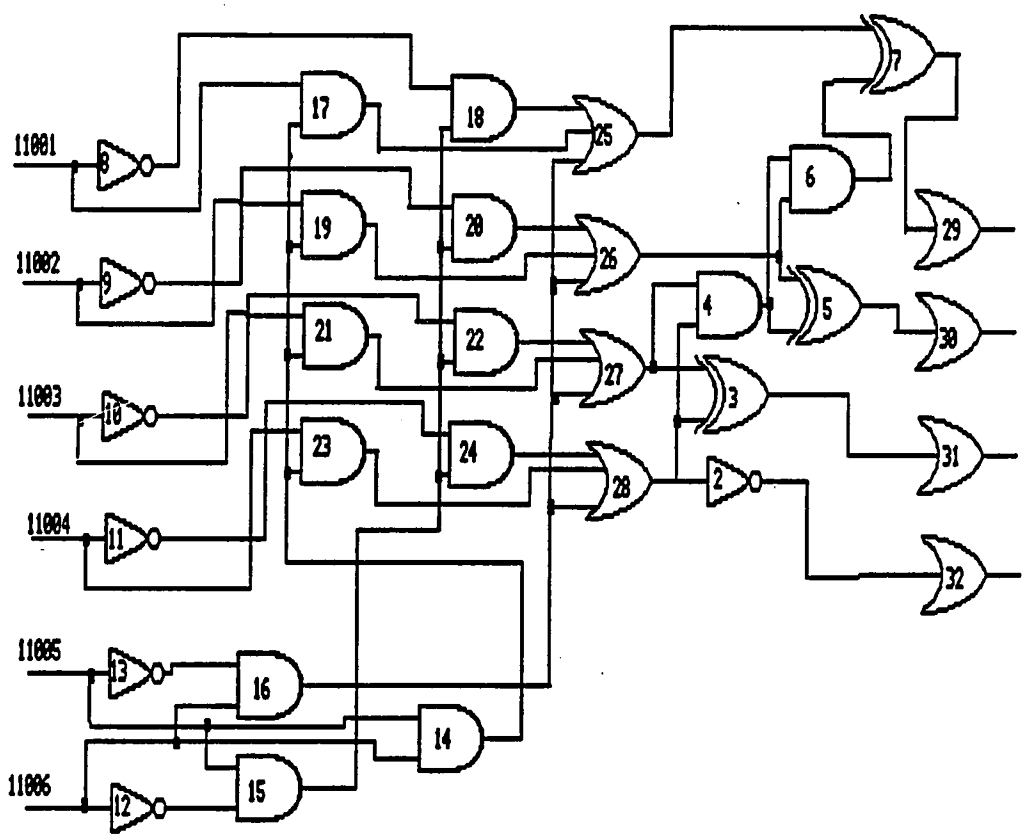


Figure 5.1 Partition Circuit For the Fault of s_a_1 At Input 2 of Gate 14 of SELECTOR Circuit

## 5.2 M5 Machine

M5 Machine is a sequential circuit with a small amount of reconvergent fanout. Its AHPL description, parameter file used, and its wiring description are given in Fig. B.2, Fig. C.2 and Table D.2 respectively.

DF3 and PODEM are run on Machine M5 with users injecting each individual fault. The CPU times used by PODEM and DF3 are the same order of magnitude. Because Machine M5 does not have many reconvergent fanouts, PODEM does not work faster than DF3. For example, the fault s_a_1 at input 1 of gate 14, DF3 used 0.08 seconds and PODEM used 0.04 seconds. Fig. 5.2 shows the partition circuit diagram for this fault. Table 5.2a, 5.2b and 5.2c summarizes the CPU time, external input vectors and the number of DON'T CAREs respectively. No particular conclusion can be drawn from Table 5.2c.

Next, SCR3 and SPODEM are run on M5 Machine. Table 5.2d summarizes some statistical results. It is found that both SCR3 and SPODEM spent almost the same amount of total running time and generated the same length of input sequences. Obviously, SPODEM does not perform any better than SCR3. Again, the faults of output s_a_0 and s_a_1 of gate 1 are undetectable because this gate is a control flip flop.

| Gate | Input | Stuck_at | DF3 Computing Time (sec) | PODEM Computing Time (sec) |
|---|---|---|---|---|
| 10 | 0 | 0 | 0.04 | 0.05 |
| 10 | 1 | 1 | 0.05 | 0.05 |
| 11 | 2 | 1 | 0.04 | 0.05 |
| 13 | 1 | 1 | 0.06 | 0.04 |
| 14 | 1 | 1 | 0.08 | 0.04 |
| 15 | 0 | 0 | 0.03 | 0.03 |
| 16 | 3 | 0 | 0.05 | 0.04 |

**Table 5.2a CPU Time ---M5 Machine**

| Gate | Input | Stuck_at | External Input Vector | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 10 | 0 | 0 | 0 | 0 |
| 10 | 1 | 1 | 1 | 1 |
| 11 | 2 | 1 | 1 | 1 |
| 13 | 1 | 1 | 1 | 1 |
| 14 | 1 | 1 | 1 | 1 |
| 15 | 0 | 0 | 1 | 1 |
| 16 | 3 | 0 | 1 | 1 |

**Table 5.2b External Input Vector --- M5 Machine**

| Gate | Input | Stuck_at | Number of DON'T CAREs In A Partition Circuit | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 10 | 0 | 0 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |
| 11 | 2 | 1 | 2 | 1 |
| 13 | 1 | 1 | 0 | 0 |
| 14 | 1 | 1 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 1 | 0 |

Table 5.2c Number of DON'T CAREs --- M5 Machine

| M5 Machine | SCR3 | SPODEM |
|---|---|---|
| # of gates | 17 | 17 |
| # of flip flops | 3 | 3 |
| # of external inputs | 1 | 1 |
| total # of faults | 61 | 61 |
| # of faults detected | 59 | 59 |
| % of fault coverage | 96.7 | 96.7 |
| Running time (sec) | 18.89 | 18.82 |
| # of input sequences | 17 | 17 |
| Faults not detected | 1 output SA1<br>1 output SA0 | 1 output SA1<br>1 output SA0 |

Table 5.2d Statistical Summary ---M5 Machine

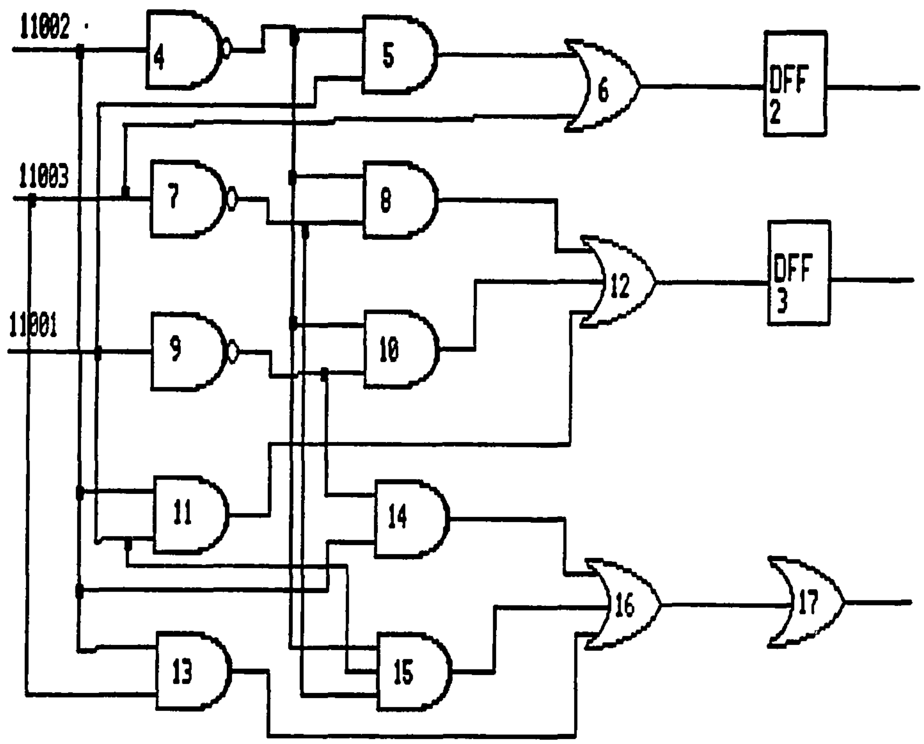


**Fig. 5.2 Partition Circuit For the Fault of s-a-1 at Input 1 of Gate 14 of M5 Machine**

## 5.3 8-Bit Look Ahead Adder

The 8-Bit Look Ahead Adder [2] is a combinational circuit with a fair amount of reconvergent fanout. Its AHPL description, the parameter file used, and its wiring description are in Fig. B.3, Fig. C.3, and Table D.3 respectively.

DF3 and PODEM are run on this circuit with users injecting each individual fault. The CPU time used by DF3 and PODEM for a selected range of faults are summarized in Table 5.3a. It is observed that the time used by PODEM is less than the time used by DF3 but both times are the same order of magnitude. But, the PART subroutine of PODEM takes primary input and output faults into consideration that DF3 ignores. As a partition circuit size increases, the time for PODEM's PART routine to formulate a circuit description list is more than the time for DF3's PART routine. Chapter 4.2.2 gives a detailed description of the difference between PODEM's PART and DF3's PART. For example, for the fault of s_a_0 at output of gate 52, the partition circuit has 123 circuit elements. DF3 used 0.95 seconds to find a test vector for this fault and PODEM used 0.50 seconds. It seems that PODEM is twice as fast as DF3. Because of the big size of the partition circuit, PODEM's PART may spend 0.2 second and DF3's PART may spend 0.1 seconds to formulate the circuit description lists. Without taking the time spent by PARTs, the actual time used by DF3 may be 0.83 seconds and the time used by PODEM may be 0.3. In this case, PODEM is three times faster than DF3. This approach of measuring PODEM and DF3 is more appropriate and fits into the theoretical prediction.

PODEM takes 53.2 seconds to determine the fault of s_a_0 at output of gate 72, but DF3 only took 0.78 seconds. It seems that PODEM misses some test vectors and ends up exploring all test vectors exhaustivly. For the fault of s_a_0 at output of gate 132, DF3 determines that the fault is undetectable but PODEM determines the fault is detectable. The inconsistency of these two results is because gate 132 is a primary output gate and DF3 does not consider faults appearing at primary inputs and primary outputs. Table 5.3b and Table 5.3c summarize the external input vectors and number of DON'T CAREs. Theoretically, PODEM should leave more DON'T CAREs in a partition circuit but the results in Table 5.3c shows the number of DON'T CAREs left by DF3 are more than those left by PODEM. It is suspected that DF3 is not working the way it should be.

No statistical result is obtained from SPODEM because SPODEM spent extremely large amout of time in the PODEM routine to find test vectors for some faults. It may be that the PODEM routine misses some of the test vectors for a fault.

| Gate | Input | Stuck_at | DF3 Computing Time (sec) | PODEM Computing Time (sec) |
|---|---|---|---|---|
| 2 | 0 | 0 | ---- | ---- |
| 12 | 0 | 0 | 0.71 | 0.35 |
| 22 | 0 | 0 | 0.72 | 0.39 |
| 32 | 0 | 0 | 0.73 | 7.98 |
| 42 | 0 | 0 | 0.93 | 0.50 |
| 52 | 0 | 0 | 0.95 | 0.58 |
| 62 | 0 | 0 | 0.17 **** | 0.17 **** |
| 72 | 0 | 0 | 0.78 **** | 53.20 **** |
| 82 | 0 | 0 | 0.48 | 0.34 |
| 92 | 0 | 0 | 0.39 | 0.27 |
| 102 | 0 | 0 | 0.28 | 0.19 |
| 112 | 0 | 0 | 0.15 | 0.11 |
| 122 | 0 | 0 | 0.09 | 0.07 |
| 132 | 0 | 0 | 0.03 **** | 0.02 |

**Table 5.3a CPU Time --- 8-Bit Look Ahead Adder**

**Note: 1) **** means a fault is undetectable**

| Gate | Input | Stuck_at | External Input Vector | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 2 | 0 | 0 | ---- | ---- |
| 12 | 0 | 0 | 11XXXXXX0100XXXX | 1000000000000000 |
| 22 | 0 | 0 | 001XXXXX1110XXXX | 0000000011000000 |
| 32 | 0 | 0 | XX001XXXXX111XXX | 0000000000010000 |
| 42 | 0 | 0 | XXXX011XXXXX1010 | 0000010X0000000X |
| 52 | 0 | 0 | XXXXX001XXXXX111 | 0000000000001010 |
| 62 | 0 | 0 | **** | **** |
| 72 | 0 | 0 | **** | **** |
| 82 | 0 | 0 | 00001XXX11111XXX | 0000000101111111 |
| 92 | 0 | 0 | 00XXXXXX1000XXXX | 0000000010000000 |
| 102 | 0 | 0 | XX00XXXXXX10XXXX | XX000000XX100000 |
| 112 | 0 | 0 | XXXX0XXXXXXX1000 | XXXX0000XXXX1000 |
| 122 | 0 | 0 | XXXXXX0XXXXXXX10 | XXXXXX00XXXXXX10 |
| 132 | 0 | 0 | **** | XXXXXXXXXXXXXXXX |

Table 5.3b External Input Vectors (8-Bit Look Ahead Adder)

Note: 1) **** means a fault is undetectable

| Gate | Input | Stuck_at | Number of DON'T CAREs In A Partition Circuit | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 2 | 0 | 0 | ---- | ---- |
| 12 | 0 | 0 | 48 | 2 |
| 22 | 0 | 0 | 49 | 3 |
| 32 | 0 | 0 | 68 | 5 |
| 42 | 0 | 0 | 84 | 16 |
| 52 | 0 | 0 | 97 | 8 |
| 62 | 0 | 0 | **** | **** |
| 72 | 0 | 0 | **** | **** |
| 82 | 0 | 0 | 27 | 1 |
| 92 | 0 | 0 | 49 | 1 |
| 102 | 0 | 0 | 41 | 1 |
| 112 | 0 | 0 | 15 | 1 |
| 122 | 0 | 0 | 7 | 1 |
| 132 | 0 | 0 | **** | 0 |

Table 5.3c Number of DON'T CAREs (8-Bit Look Ahead Adder)

Note: 1) **** means a fault is undetectable

## 5.4 8-Bit Incrementer

An 8-Bit Incrementer [2] is a combinational circuit with almost no reconvergent fanouts. Its AHPL description, the parameter file used, and its wiring description are in Fig. B.4, Fig. C.4, and Table D.4.

DF3 and PODEM are run on 8-Bit Incrementer with users injecting faults. Table 5.4a gives a summary of CPU time. It is observed that the CPU time used by PODEM is slightly less than the time used by DF3 and both times are the same order of magnitude. For example, for a fault of s_a_0 at the output of gate 12, PODEM spends 0.07 seconds and DF3 spends 0.10 seconds. There are problems for DF3 to detect the fault at gate 18 and gate 20. These gates are primary output gates and DF3 is not implemented to detect faults at primary outputs and primary inputs. In fact, these primary output gates are additional OR gates which do not appear in the AHPL decription of the circuit. The remedy is to substitute DF3's PART routine by PODEM's PART routine. Since PODEM is implemented to be compatable to DF3, it is very easy to do such a substitution. Table 5.4b and 5.4c summarize the external input vectors to the circuit and the number of DON'T CAREs in a partition circuit. The number of X's in an external input vector generated by DF3 is more than those generated by PODEM. When SCIRTSS is run on this circuit, the input vector returned by DF3 or PODEM is used by SCIRTSS for the sensitization search. The more number of X's in an input vector, the easier is the sensitization search.

SCR3 and SPODEM are run on the circuit. Table 5.4d summarizes some statistical results. The percentage of fault coverage and the number of input sequences are the same for SCR3 and SPODEM; but the total running time of PODEM is more than the running time of DF3. This seems to contradict the results in Table 5.4a which shows that the CPU time used by PODEM is less than the CPU time used by DF3. It is mentioned at the end of the above paragraph that the external input vectors generated by PODEM have less X's than the number of X's in those vectors returned by DF3. That means it is more difficult for SPODEM to do the sensitization search. That is why the total running time of SPODEM is more than the running time of SCR3.

The output of gate 1 s_a_1 and s_a_0 are undetectable because the gate is a control flip flop. The input 1 of gate 9 s_a_1 and the input 1 of gate 10 s_a_1 are undetectable because these two input are permanently tied to +5 volts (+11000). Fig. 5.4 describes the connections of these twoinputs to +5 volts.

| Gate | Input | Stuck_at | DF3 Computing Time (sec) | PODEM Computing Time (sec) |
|---|---|---|---|---|
| 2 | 0 | 0 | 0.09 | 0.05 |
| 4 | 0 | 0 | 0.03 | 0.03 |
| 6 | 0 | 0 | 0.06 | 0.03 |
| 8 | 0 | 0 | 0.02 | 0.01 |
| 10 | 0 | 0 | 0.09 | 0.07 |
| 12 | 0 | 0 | 0.10 | 0.07 |
| 14 | 0 | 0 | 0.06 | 0.06 |
| 16 | 0 | 0 | 0.04 | 0.07 |
| 18 | 0 | 0 | @@@@ | @@@@ |
| 20 | 0 | 0 | @@@@ | @@@@ |

Table 5.4a CPU Time --- 8-Bits Incrementer

Note: 1) @@@@ means DF3 does not work but PODEM works on the fault.

| Gate | Input | Stuck_at | External Input Vector | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 2 | 0 | 0 | 01111111 | 10000000 |
| 4 | 0 | 0 | XX011111 | XX100000 |
| 6 | 0 | 0 | XXXX0111 | XXXX1000 |
| 8 | 0 | 0 | XXXXXX01 | XXXXXX10 |
| 10 | 0 | 0 | XXXXXX11 | 00000001 |
| 12 | 0 | 0 | XXXX1111 | 00000111 |
| 14 | 0 | 0 | XX111111 | 00001111 |
| 16 | 0 | 0 | 11111111 | 01111111 |
| 18 | 0 | 0 | @@@@ | @@@@ |
| 20 | 0 | 0 | @@@@ | @@@@ |

Table 5.4b External Input Vectors (8-Bits Incrementer)

Note: 1) @@@@ means DF3 does not work but PODEM works on the fault.

| Gate | Input | Stuck_at | Number of DON'T CAREs In A Partition Circuit | |
|---|---|---|---|---|
| | | | DF3 | PODEM |
| 2 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 23 | 0 |
| 12 | 0 | 0 | 15 | 0 |
| 14 | 0 | 0 | 6 | 0 |
| 16 | 0 | 0 | 0 | 0 |
| 18 | 0 | 0 | @@@@ | @@@@ |
| 20 | 0 | 0 | @@@@ | @@@@ |

Table 5.4c Number of DON'T CARE's (8-Bit Incrementer)

Note: 1) @@@@ means DF3 does not work but PODEM works on the fault.

| 8-Bit Incrementer | SCR3 | SPODEM |
|---|---|---|
| # of gates | 24 | 24 |
| # of flip flops | 1 | 1 |
| # of external inputs | 8 | 8 |
| total # of faults | 102 | 102 |
| # of faults detected | 98 | 98 |
| % of fault coverage | 96.1 | 96.1 |
| Running time (sec) | 16.74 | 20.46 |
| # of input sequences | 16 | 16 |
| Faults not detected | 1 output SA1<br>1 output SA0<br>9 input 1 SA1<br>10 input 1 SA1 | 1 output SA1<br>1 output SA0<br>9 input 1 SA1<br>10 input 1 SA1 |

Table 5.4d Statistical Summary (8-Bit Incrementer)

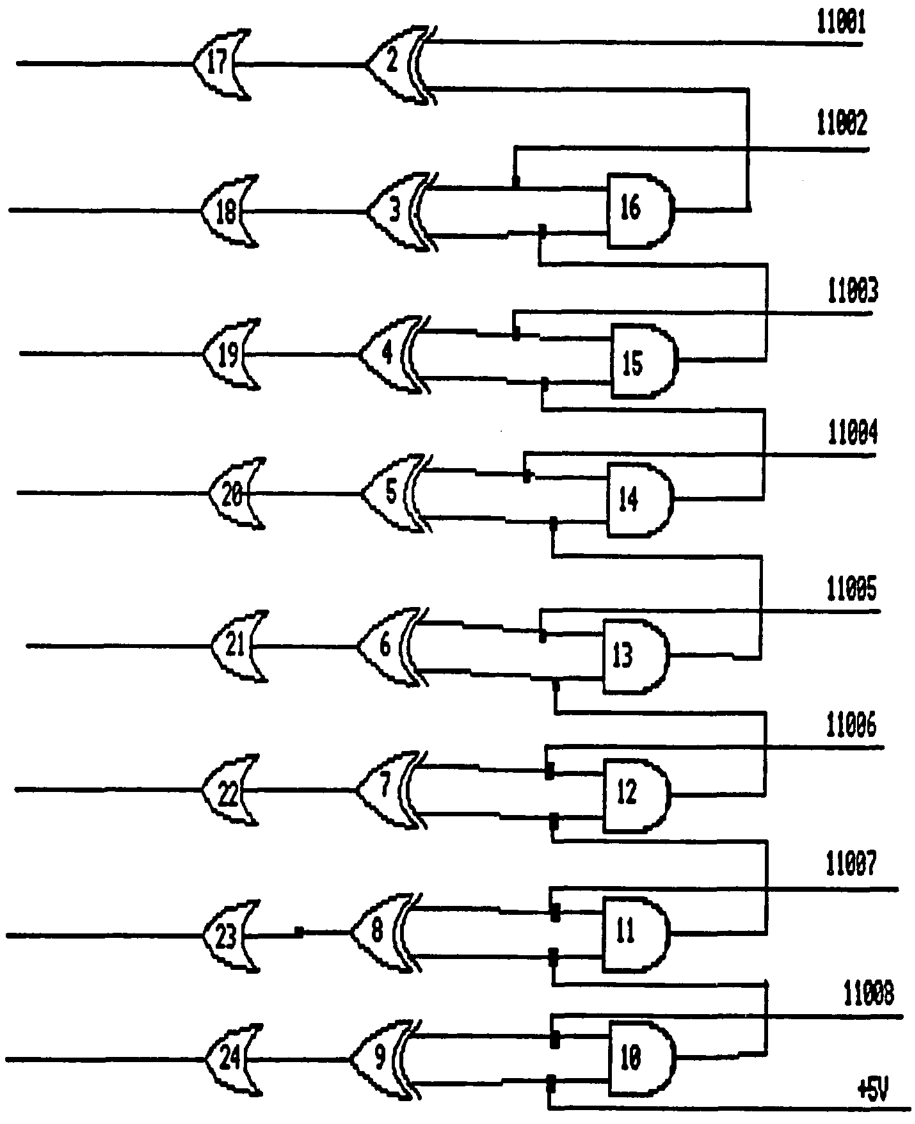


Figure 5.3 Partition Circuit For the Fault of s_a_1 At Input 1 of Gate 9 of 8-Bit Incrementer.

# CHAPTER 6

## CONCLUSION

It is found that the PODEM algorithm is 1 to 23 times faster than the D-Algorithm [9,13] in finding test vectors for faults in combinational circuits. On the other hand, the efficiency of PODEM depends on the characteristic of a circuit. PODEM functions are especially good on circuits with many reconvergent fanouts.

PODEM is designed to find test vectors for faults in combinational circuits, the resulting input vectors are the test vectors. PODEM can also be applied to find test vectors for faults in sequential circuits by partitioning a sequential circuit into a combinational subcircuit for a fault and PODEM is applied to find test vectors for the fault in that combinational subcircuit and then a sensitization search for that input vector is performed. Further propagation search may be required to propagate a fault from a storage element to a primary output. Although the PODEM algorithm works faster than the D-Algorithm in combinational circuits, that may not always improve the total running time for finding test vectors for faults in sequential circuits. It depends on the characteristic of the input vectors returned by the PODEM algorithm. When the sensitization search is performed for an input vector returned from PODEM, the more DON'T CAREs are in the input vector, the easier is the sensitization search to reach goal state; the less DON'T CAREs are in the input vector, the more difficult is the sensitization to reach goal state. Even though the PODEM algorithm finds input vectors much faster than the D-Algorithm, the number of DON'T CAREs in input vectors is an important factor to determine the total running time.

Further research in measuring reconvergent fanouts may help to choose a more suitable test generation algorithm for finding faults in a sequential circuit. Although the PODEM algorithm works faster than the D-Algorithm, the PODEM algorithm leaves many DON'T CAREs in a partition circuit of a sequential circuit and in an returned input vector, that may result in a smaller number of other faults to be detected by the same test vector.

The PODEM algorithm implemented in this thesis does not have much intelligence. The number of backtrackings and the time between backtrackings consume most of the time in test generation. A way to shorten the time of test generation is to improve the above two factors. The number of backtrackings can be reduced by corporating some artificial intelligence strategies into PODEM Algorithm so that it knows which primary input and inital value have higher possiblity to propagate a fault. For example, the FAN algorithm [5] and Miczo's PODEM algorithm [6] have guidiance for choosing primary input and its initial value.

The performance of software implementation is slower than the performance of hardware implementation. Some portions of the PODEM algorithm can be implemented with hardwares so that a faster speed can be achieved without much sacrifice in cost.

Finally, there are some comments on the DF3 version of SCIRTSS. SCIRTSS does not work if the first fault to be detected is a primary output gate. Those primary input gates are additional OR gates which are not part of an original circuit and SCIRTSS is not supposed to test any faults associated with these gates. However, there is a possibility that these faults may be put on the top of a fault list and SCIRTSS will be stuck in this case. Those faults can be detected only by using the test vectors for other faults.

# APPENDIX A

## RUNNING PROGRAMS

## A.1 How To Link SCIRTSS With PODEM

After all subroutines of PODEM-Algorithm are compiled and the object file is named PODEM.OBJ, PODEM.OBJ is linked with SCIRTSS. The executable file is named SPODEM.EXE. Fig. A.1 show a COM file to compile PODEM-Algorithm subroutines and Fig. A.2 shows a COM file to link PODEM with SCIRTSS.

```
$FOR PODEM+IMPDR+INSERT+ASSIGN+XDUCE+DGOUT+PART-
 +START+FUNCTION
```

Figure A.1 Command File C_PDM.COM

```
$SET NOVERITY
LINK SCIRTSS/OPTION,-
SALL,-
BRIDGE/OPTION,-
TIME/OPTION,-
PODEM,-
FLTSIM/OPTION,-
MYSCH/OPTION
$RENAME SCIRTSS.EXE SPODEM.EXE
```

Figure A.2 Command File LSCR_PDM.COM

## A.2 How To Run SCIRTSS With PODEM

When PODEM is linked with SCIRTSS, SCIRTSS asks the same number and same type of questions as it is running SCIRTSS with D-Algorithm. Although SCIRTSS allows users to inject a fault, it is inconvenient to do it in this way. Appendix A.3 describes a much more convenient way to inject a fault. SCIRTSS provides a fault ordering, users are restricted to simulate a fault according to that order. Figure A.3 gives a demonstration of how to run SCIRTSS with PODEM in a VAX/VMS system. SCIRTSS requires at least three input files, they are Stage I output file, Stage III output file, and parameter file.

```
$RUN SPODEM
ENTER SCIRTSS OUTPUT FILE______________:M5.OUT
 ENTER UPDATE INPUT SEQUENCE FILE_______:M5.UPD
 ENTER FILE NAME OF STAGE1 OUTPUT_______:M5.ST1
 DO YOU USE COMSEC DATA?___________[Y/N]:N
 USE 2nd LVL PARM FILE?____________[Y/N]:N
CHANGE PARAMETERS?________________[Y/N]:N
 READ D.I.V. FROM FILE?____________[Y/N]:N
 SAVE D.I.V.?______________________[Y/N]:N
  ENTER MODULE NO.: .
ENTER FILE NAME OF STAGE3 OUTPUT_______:M5.ST3
 ENTER FILE NAME OF PARAMETERS__________:M5.PAR
 USE 3-VALUE SEARCH SIMUL__________[Y/N]:N
CONTINUE SCIRTSS?_________________[Y/N]:Y
 CHANGE PARAMETERS?________________[Y/N]:N
DUMP SETA TO A FILE?______________[Y/N]:N
CONTINUE SCIRTSS?_________________[Y/N]:N
 SURE YOU WANT TO STOP?____________[Y/N]:Y
 ENTER RETAIN CIRCUIT STATE FILE NAME___:.
 FORTRAN STOP
```

**Figure A.3   How To Run SCIRTSS With PODEM**

## A.3 How To Run PODEM Alone

PODEM can be run without SCIRTSS to generate test vectors for injected faults in combinational circuits. After entering the command, RUN PODEM; the program asks for the Stage III Compiler output file and the gate number which a fault occurs. The gate numbers are shown on the screen after the question, "ARE YOU READY TO RUN PODEM-ALGORITHM ?", is answered. And then it asks for which input line with the fault. Input number=0 means the output line of a specified gate has the fault; otherwise, an input line of the gate has the fault. The next question is the type of fault. The answer is either 0 or 1 for stuck_at_0 or stuck_at_1. The program then asks what type of print mode you want. There are three choices: 0, 1, and 2. After PODEM finds the test vector for a fault, you have to tell PODEM whether you want to continue or to stop. One of the advantages of the program is to let the user inject a particular fault for PODEM to find the test vectors. Fig. A.4 gives a demonstration of how to run PODEM alone in a VAX/VMS system. This program require an input file, it is the Stage III output file. Fig. A.1 shows a COM file to compile PODEM-Algorithm subroutines and Fig. A.5 shows a COM file to link SPODEM.OBJ and PODEM.OBJ to produce an executable file PODEM.EXE.

```
$RUN PODEM
 ENTER FILE NAME OF NETWORK DESCRIPTION  M5.ST3
 ARE YOU READY TO RUN PODEM-ALGORITHM ?  Y
 GIVE THE NUMBER OF GATE WITH FAULT  12
 GAVE THE NUMBER OF INPUT WITH FAULT  0
 GIVE THE TYPE OF FAULT  1
 WHICH PRINT MODE DO YOU WANT?
NO PRINT TYPE 0
CIRCUIT PRINT TYPE 1
DEBUGGING MODE TYPE 2  0
   PODEM-ALGORITHM STARTS !!
DO YOU HAVE ANY MORE FAULTS TO BE TESTED?  N
 FORTRAN STOP
```

**Figure A.4 How To Run PODEM Alone**

```
$LINK/EXE=PODEM SPODEM,PODEM
```

**Figure A.5 Command File LPDM.COM**

## A.4 File Structure

Figure A.6 shows the file structure of different subroutines to be used in PODEM which is interfaced to SCIRTSS. PODEM is called by DALSEN, a subroutine of SCIRTSS.

```
SCIRTSS
          .
          .
          .
          UNDTGF
                   .
                   .
                   .
                   DALSEN
                            .
                            .
                            .
                            PODEM
                                 TEMP (include file)
                                 PART
                                          START
                                 ASSIGN

                                          DGOUT

                                 IMPDR

                                          FUNCTION
                                          INSERT
                                          DGOUT

                                 XDUCE
                                 DGOUT
```

Figure A.6 File Sructure of SCIRTSS with PODEM

Figure A.7 shows the file structure of different subroutines to be used in PODEM which is not interfaced to SCIRTSS but is called by a main program called SPODEM. SPODEM formulates circuit description table from circuit wiring list of the output of Stage 3 Compiler. The advantage of using SPODEM to called PODEM rather than using SCIRTSS is that the former scheme allows users to specify a fault for PODEM to find a test vector for a fault in a combinational circuit. Although SCIRTSS has an option to let users to specify a fault, users have to run SCIRTSS again for another fault.

```
SPODEM

      TEMP (include file)
      INITIME
      SECOND
      PODEM

              TEMP (include file)
              PART
              START
              ASSIGN

                     DGOUT

              IMPDR

                     FUNCTION
                     INSERT
                     DGOUT

              XDUCE
              DGOUT

      DGOUT
```

Figure A.7 File Structure of PODEM As A Stand Alone Program

# APPENDIX B

# AHPL DESCRIPTION

```
MODULE       :SELECTOR.
 EXINPUTS     :X[4];Y[2];CLOCK.
 BUSES        :BUS1[4].
 OUTPUTS      :Z[4].
 CLUNITS      :INC[4]<: INCER(4).
 BODY SEQUENCE:CLOCK.
 1  =>1.
   ENDSEQUENCE
   CONTROLRESET(1);
   Z=INC(BUS1);
   BUS1=(X!^X!\1,1,1,1\)*(Y[0]&Y[1],Y[0]&^Y[1],^Y[0]&Y[1]).
 END.
```

```
CLU :INCER(X) (I).
 INPUTS :X[I].
 OUTPUTS:Y[I].
 CTERMS :RESULT[I];CARRY[I-1].
 BODY
    CARRY[I-2],RESULT[I-1] = X[I-1],^X[I-1];
    FOR J=I-2 TO 0 CONSTRUCT
       RESULT[J] = CARRY[J] @ X[J];
       IF J<>0 THEN CARRY[J-1]=CARRY[J]&X[J] FI
    ROF;
    Y = RESULT.
 END.
```

**Figure B.1   AHPL Description of SELECTOR**

```
MODULE: M5.
    EXINPUTS: X;CLOCK.
    EXOUTPUTS: Z.
    MEMORY: Y[2].
    BODY SEQUENCE: CLOCK.
     1 =>(1).
    ENDSEQUENCE
    CONTROLRESET(1);
    Y <= Y[1]+^Y[0]&X,^Y[0]&^Y[1]+^Y[0]&^X+Y[0]&X;
    Z=Y[0]&Y[1]+Y[0]&^X+^Y[0]&^Y[1]&X.
END.
```

Figure B.2 AHPL Description of M5 Machine

```
MODULE:LKAHDTST.
 EXINPUTS:A[8];B[8];CLOCK.
 MEMORY:AC[8];L.
 EXOUTPUTS:Z[9].
 CLUNITS:ADD[9]<:ADDLA8.
 BODY SEQUENCE: CLOCK.
  1 L,AC<=ADD(A;B;\0\);
         =>(1).
  ENDSEQUENCE
  CONTROLRESET(1);
  Z = L,AC.
 END.
```

```
CLU:ADDLA8(X;Y;CIN).
 INPUTS:X[8];Y[8];CIN.
 OUTPUTS:TERMOUT[9].
 CLUNITS:LA[6]<:CLA.
 CLUNITS:PG[2]<:PROPGEN.
 CLUNITS:SUMBIT<:SUM.
 CTERMS:G[8];P[8];C[9];GC.
 BODY
  FOR I=0 TO 7 CONSTRUCT
  G[I],P[I]=PG(X[I];Y[I])
  ROF;
  C[5:8]=LA[2:5](G[4:7];P[4:7];CIN);
  GC=LA[0](G[4:7];P[4:7];CIN) +
     LA[1](G[4:7];P[4:7];CIN)&CIN;
  C[1:4]=LA[2:5](G[0:3];P[0:3];GC);
  C[0]=LA[0](G[0:3];P[0:3];GC) +
       LA[1](G[0:3];P[0:3];GC)&GC;
  TERMOUT[0] = C[0];
  FOR I=0 TO 7 CONSTRUCT
  TERMOUT[I+1]=SUMBIT(P[I];C[I+1])
  ROF.
 END.
```

Figure B.3  AHPL Description of 8-bit Look Ahead Adder

```
CLU: CLA(G;P;CIN).
 INPUTS: G[4]; P[4]; CIN.
 OUTPUTS: TERMOUT[6].
 BODY
  TERMOUT[5] = CIN;
  TERMOUT[4] = G[3] + P[3]&CIN;
  TERMOUT[3] = G[2] + P[2]&G[3] + P[3]&P[2]&CIN;
  TERMOUT[2] = G[1] + P[1]&G[2] + P[2]&P[1]&G[3] +
               P[3]&P[2]&P[1]&CIN;
  TERMOUT[0] = G[0] + P[0]&G[1] + P[1]&P[0]&G[2] +
               P[2]&P[1]&P[0]&G[3];
  TERMOUT[1] = P[3]&P[2]&P[1]&P[0].
 END.


CLU: PROPGEN(X;Y).
 INPUTS: X[1];Y[1].
 OUTPUTS:TERMOUT[2].
 BODY
  TERMOUT[0]=X&Y;
  TERMOUT[1]=X&^Y + ^X&Y.
 END.


CLU:SUM(PROP;CIN).
 INPUTS:PROP;CIN.
 OUTPUTS:TERMOUT.
 BODY
  TERMOUT=PROP&^CIN + ^PROP&CIN.
 END.


MODULE:LKAHDTST.
EXINPUTS:A[8];B[8];CLOCK.
MEMORY:AC[8];L.
EXOUTPUTS:Z[9].
CLUNITS:ADD[9]<:ADDLA8.
BODY SEQUENCE: CLOCK.
1   L,AC<=ADD(A;B;\0\);
    =>(1).
ENDSEQUENCE
CONTROLRESET(1);
Z = L,AC.
END.
```

Figure B.3 (continued)

```
CLU:ADDLA8(X;Y;CIN).
 INPUTS:X[8];Y[8];CIN.
 OUTPUTS:TERMOUT[9].
 CLUNITS:LA[6]<:CLA.
 CLUNITS:PG[2]<:PROPGEN.
 CLUNITS:SUMBIT<:SUM.
 CTERMS:G[8];P[8];C[9];GC.
 BODY
  FOR I=0 TO 7 CONSTRUCT
  G[I],P[I]=PG(X[I];Y[I])
  ROF;
  C[5:8]=LA[2:5](G[4:7];P[4:7];CIN);
  GC=LA[0](G[4:7];P[4:7];CIN) +
     LA[1](G[4:7];P[4:7];CIN)&CIN;
  C[1:4]=LA[2:5](G[0:3];P[0:3];GC);
  C[0]=LA[0](G[0:3];P[0:3];GC) +
       LA[1](G[0:3];P[0:3];GC)&GC;
  TERMOUT[0] = C[0];
  FOR I=0 TO 7 CONSTRUCT
  TERMOUT[I+1]=SUMBIT(P[I];C[I+1])
  ROF.
 END.
```

```
CLU: CLA(G;P;CIN).
 INPUTS: G[4]; P[4]; CIN.
 OUTPUTS: TERMOUT[6].
 BODY
  TERMOUT[5] = CIN;
  TERMOUT[4] = G[3] + P[3]&CIN;
  TERMOUT[3] = G[2] + P[2]&G[3] + P[3]&P[2]&CIN;
  TERMOUT[2] = G[1] + P[1]&G[2] + P[2]&P[1]&G[3] +
               P[3]&P[2]&P[1]&CIN;
  TERMOUT[0] = G[0] + P[0]&G[1] + P[1]&P[0]&G[2] +
               P[2]&P[1]&P[0]&G[3];
  TERMOUT[1] = P[3]&P[2]&P[1]&P[0].
 END.
```

Figure B.3 (continued)

```
CLU: PROPGEN(X;Y).
 INPUTS: X[1];Y[1].
 OUTPUTS:TERMOUT[2].
 BODY
  TERMOUT[0]=X&Y;
  TERMOUT[1]=X&^Y + ^X&Y.
 END.
```

```
CLU:SUM(PROP;CIN).
 INPUTS:PROP;CIN.
 OUTPUTS:TERMOUT.
 BODY
  TERMOUT=PROP&^CIN + ^PROP&CIN.
 END.
```

Figure B.3 (Continued)

```
MODULE: INCTR8.
  EXINPUTS: X[8];CLOCK.
  EXOUTPUTS: Z[8].
  CLUNITS: INC[8]<: INCRMT{8}.
  BODY SEQUENCE: CLOCK.
    1  => 1.
  ENDSEQUENCE
  CONTROLRESET(1);
  Z=INC(X).
  END.


CLU: INCRMT(X) {I}.
  INPUTS: X[I].
  OUTPUTS: TERMOUT[I].
  CTERMS: TA[I].
  BODY
    FOR J = 0 TO I-1 CONSTRUCT
      TERMOUT[J] = X[J]@TA[J]
    ROF;
    FOR J = I-1 TO 0 CONSTRUCT
      IF J=I-1 THEN
      TA[J]=\1\
      ELSE
      TA[J]=X[J+1]&TA[J+1]
      FI
    ROF.
  END.
```

Figure B.4   AHPL Description of 8-bit Incrementer

# APPENDIX C

## PARAMETER FILES

```
SELECTOR
GFFLAG---FAULT
SWITCH___OFF  OFF  OFF  OFF  OFF   ON
   .2  1.0   .2   .8
   10    5    0  200   50    5    1
10
11
ABBREV       -6   +1    2    1
RANDOM        0                RETAIN
```

Figure C.1 Parameter File of SELECTOR

```
M5: HOIMING LEE
GFFLAG---FAULT                 100
SWITCH___OFF   OFF  OFF  OFF  OFF  OFF
   .2  0.2   .2   .2
   10    5    0  200   20    5    1
111
00
ABBREVIATE   -3   +1    0    2
RANDOM        0                RETAIN
```

Figure C.2 Parameter File of M5 Machine

```
lk8add
GFFLAG   FAULT
SWITCH   OFF   OFF  OFF  OFF  OFF  ON
   .2  1.0   .2   .2
   10    5    0  200   20    5    1
111111111111
111111111111
ABBREV        6   +1    0
    1    2    3    4    5    6
RANDOM        0                RETAIN
```

Figure C.3 Parameter File of 8-bit Look Ahead Adder

```
INCTR8: HOIMING LEE
GFFLAG---FAULT                100
SWITCH___OFF    OFF  OFF  OFF  OFF  OFF
   .2  1.0   .2   .2
   10    5    0  200   20    5    1
0
0
ABBREVIATE    -1    1    0
HEXD           2               RETAIN
AB000000
CD000000
68000000
D4000000
E6000000
F3000000
99000000
B2000000
bb000000
99000000
11000000
33000000
55000000
98000000
98000000
89000000
```

Figure C.4    Parameter File of 8-bit Incrementer

# APPENDIX D

## THE CIRCUIT WIRING DESCRIPTION

```
NUMBER OF ELEMENT :       32
NUMBER OF FLIPFLOP:        1
NUMBER OF EXINPUT :        6
NUMBER OF EXOUTPUT:        4

CIRCUIT DESCRIPTION:

ELEMENT TYPE        INPUT CONNECTIONS
------- ----        -----------------
   1    DFF             1
   2    NAND           28
   3    XOR            27      28
   4    AND            27      28
   5    XOR            26       4
   6    AND            26       4
   7    XOR            25       6
   8    NAND        11001
   9    NAND        11002
  10    NAND        11003
  11    NAND        11004
  12    NAND        11006
  13    NAND        11005
  14    AND         11006   11005
  15    AND            12   11005
  16    AND         11006      13
  17    AND         11001      14
  18    AND             8      15
  19    AND         11002      14
  20    AND             9      15
  21    AND         11003      14
  22    AND            10      15
  23    AND         11004      14
  24    AND            11      15
  25    OR             17      18      16
  26    OR             19      20      16
  27    OR             21      22      16
  28    OR             23      24      16
  29    OR     _ 1      7
  30    OR     _ 2      5
  31    OR     _ 3      3
  32    OR     _ 4      2
ELEMENT    29 HAS ZERO FANOUT.
ELEMENT    30 HAS ZERO FANOUT.
ELEMENT    31 HAS ZERO FANOUT.
ELEMENT    32 HAS ZERO FANOUT.
```

Table D.1 Wiring Description of SELECTOR

NUMBER OF ELEMENT : 17
NUMBER OF FLIPFLOP: 3
NUMBER OF EXINPUT : 1
NUMBER OF EXOUTPUT: 1

CIRCUIT DESCRIPTION:

| ELEMENT | TYPE | | INPUT CONNECTIONS | | |
|---|---|---|---|---|---|
| 1 | DFF | | 1 | | |
| 2 | DFF | | 6 | | |
| 3 | DFF | | 12 | | |
| 4 | NAND | | 2 | | |
| 5 | AND | | 11001 | 4 | |
| 6 | OR | | 5 | 3 | |
| 7 | NAND | | 3 | | |
| 8 | AND | | 7 | 4 | |
| 9 | NAND | | 11001 | | |
| 10 | AND | | 9 | 4 | |
| 11 | AND | | 11001 | 2 | |
| 12 | OR | | 11 | 10 | 8 |
| 13 | AND | | 3 | 2 | |
| 14 | AND | | 9 | 2 | |
| 15 | AND | | 11001 | 7 | 4 |
| 16 | OR | | 15 | 14 | 13 |
| 17 | OR | 1 | 16 | | |

ELEMENT 17 HAS ZERO FANOUT.

Table D.2 Wiring Description of M5 Machine

NUMBER OF ELEMENT : 139
NUMBER OF FLIPFLOP: 10
NUMBER OF EXINPUT : 16
NUMBER OF EXOUTPUT: 9

CIRCUIT DESCRIPTION:

| ELEMENT | TYPE | INPUT CONNECTIONS | |
|---|---|---|---|
| 1 | DFF | 1 | |
| 2 | DFF | 95 | 1 |
| 3 | DFF | 100 | 1 |
| 4 | DFF | 105 | 1 |
| 5 | DFF | 110 | 1 |
| 6 | DFF | 115 | 1 |
| 7 | DFF | 120 | 1 |
| 8 | DFF | 125 | 1 |
| 9 | DFF | 130 | 1 |
| 10 | DFF | 90 | 1 |
| 11 | AND | 11009 | 11001 |
| 12 | NAND | 11009 | |
| 13 | AND | 12 | 11001 |
| 14 | NAND | 11001 | |
| 15 | AND | 11009 | 14 |
| 16 | OR | 15 | 13 |
| 17 | AND | 11010 | 11002 |
| 18 | NAND | 11010 | |
| 19 | AND | 18 | 11002 |
| 20 | NAND | 11002 | |
| 21 | AND | 11010 | 20 |
| 22 | OR | 21 | 19 |
| 23 | AND | 11011 | 11003 |
| 24 | NAND | 11011 | |
| 25 | AND | 24 | 11003 |
| 26 | NAND | 11003 | |
| 27 | AND | 11011 | 26 |
| 28 | OR | 27 | 25 |
| 29 | AND | 11012 | 11004 |
| 30 | NAND | 11012 | |
| 31 | AND | 30 | 11004 |
| 32 | NAND | 11004 | |

Table D.3 Wiring Description of 8-bit Look Ahead Adder

| | | | | | |
|---|---|---|---|---|---|
| 33 | AND | 11012 | 32 | | |
| 34 | OR | 33 | 31 | | |
| 35 | AND | 11013 | 11005 | | |
| 36 | NAND | 11013 | | | |
| 37 | AND | 36 | 11005 | | |
| 38 | NAND | 11005 | | | |
| 39 | AND | 11013 | 38 | | |
| 40 | OR | 39 | 37 | | |
| 41 | AND | 11014 | 11006 | | |
| 42 | NAND | 11014 | | | |
| 43 | AND | 42 | 11006 | | |
| 44 | NAND | 11006 | | | |
| 45 | AND | 11014 | 44 | | |
| 46 | OR | 45 | 43 | | |
| 47 | AND | 11015 | 11007 | | |
| 48 | NAND | 11015 | | | |
| 49 | AND | 48 | 11007 | | |
| 50 | NAND | 11007 | | | |
| 51 | AND | 11015 | 50 | | |
| 52 | OR | 51 | 49 | | |
| 53 | AND | 11016 | 11008 | | |
| 54 | NAND | 11016 | | | |
| 55 | AND | 54 | 11008 | | |
| 56 | NAND | 11008 | | | |
| 57 | AND | 11016 | 56 | | |
| 58 | OR | 57 | 55 | | |
| 59 | AND | -11000 | 58 | | |
| 60 | OR | 59 | 53 | | |
| 61 | AND | 53 | 52 | | |
| 62 | AND | -11000 | 52 | 58 | |
| 63 | OR | 62 | 61 | 47 | |
| 64 | AND | 47 | 46 | | |
| 65 | AND | 53 | 46 | 52 | |
| 66 | AND | -11000 | 46 | 52 | 58 |
| 67 | OR | 66 | 65 | 64 | 41 |
| 68 | AND | 41 | 40 | | |
| 69 | AND | 47 | 40 | 46 | |
| 70 | AND | 53 | 40 | 46 | 52 |
| 71 | OR | 70 | 69 | 68 | 35 |
| 72 | AND | 40 | 46 | 52 | 58 |
| 73 | AND | -11000 | 72 | | |

Table D.3 (continued)

| | | | | | |
|---|---|---|---|---|---|
| 74 | OR | 73 | 71 | | |
| 75 | AND | 74 | 34 | | |
| 76 | OR | 75 | 29 | | |
| 77 | AND | 29 | 28 | | |
| 78 | AND | 74 | 28 | 34 | |
| 79 | OR | 78 | 77 | 23 | |
| 80 | AND | 23 | 22 | | |
| 81 | AND | 29 | 22 | 28 | |
| 82 | AND | 74 | 22 | 28 | 34 |
| 83 | OR | 82 | 81 | 80 | 17 |
| 84 | AND | 17 | 16 | | |
| 85 | AND | 23 | 16 | 22 | |
| 86 | AND | 29 | 16 | 22 | 28 |
| 87 | OR | 86 | 85 | 84 | 11 |
| 88 | AND | 16 | 22 | 28 | 34 |
| 89 | AND | 74 | 88 | | |
| 90 | OR | 89 | 87 | | |
| 91 | NAND | 83 | | | |
| 92 | AND | 91 | 16 | | |
| 93 | NAND | 16 | | | |
| 94 | AND | 83 | 93 | | |
| 95 | OR | 94 | 92 | | |
| 96 | NAND | 79 | | | |
| 97 | AND | 96 | 22 | | |
| 98 | NAND | 22 | | | |
| 99 | AND | 79 | 98 | | |
| 100 | OR | 99 | 97 | | |
| 101 | NAND | 76 | | | |
| 102 | AND | 101 | 28 | | |
| 103 | NAND | 28 | | | |
| 104 | AND | 76 | 103 | | |
| 105 | OR | 104 | 102 | | |
| 106 | NAND | 74 | | | |
| 107 | AND | 106 | 34 | | |
| 108 | NAND | 34 | | | |
| 109 | AND | 74 | 108 | | |
| 110 | OR | 109 | 107 | | |
| 111 | NAND | 67 | | | |
| 112 | AND | 111 | 40 | | |
| 113 | NAND | 40 | | | |
| 114 | AND | 67 | 113 | | |

Table D.3 (continued)

| | | | | |
|---|---|---|---|---|
| 115 | OR | | 114 | 112 |
| 116 | NAND | | 63 | |
| 117 | AND | | 116 | 46 |
| 118 | NAND | | 46 | |
| 119 | AND | | 63 | 118 |
| 120 | OR | | 119 | 117 |
| 121 | NAND | | 60 | |
| 122 | AND | | 121 | 52 |
| 123 | NAND | | 52 | |
| 124 | AND | | 60 | 123 |
| 125 | OR | | 124 | 122 |
| 126 | NAND | | -11000 | |
| 127 | AND | | 126 | 58 |
| 128 | NAND | | 58 | |
| 129 | AND | | -11000 | 128 |
| 130 | OR | | 129 | 127 |
| 131 | OR | _ 1 | 10 | |
| 132 | OR | _ 2 | 2 | |
| 133 | OR | _ 3 | 3 | |
| 134 | OR | _ 4 | 4 | |
| 135 | OR | _ 5 | 5 | |
| 136 | OR | _ 6 | 6 | |
| 137 | OR | _ 7 | 7 | |
| 138 | OR | _ 8 | 8 | |
| 139 | OR | _ 9 | 9 | |

ELEMENT 131 HAS ZERO FANOUT.
ELEMENT 132 HAS ZERO FANOUT.
ELEMENT 133 HAS ZERO FANOUT.
ELEMENT 134 HAS ZERO FANOUT.
ELEMENT 135 HAS ZERO FANOUT.
ELEMENT 136 HAS ZERO FANOUT.
ELEMENT 137 HAS ZERO FANOUT.
ELEMENT 138 HAS ZERO FANOUT.
ELEMENT 139 HAS ZERO FANOUT.

Table D.3 (continued)

```
NUMBER OF ELEMENT :        24
NUMBER OF FLIPFLOP:         1
NUMBER OF EXINPUT :         8
NUMBER OF EXOUTPUT:         8


CIRCUIT DESCRIPTION:

ELEMENT   TYPE          INPUT CONNECTIONS
-------   ----          -----------------


   1     DFF                1
   2     XOR               16   11001
   3     XOR               15   11002
   4     XOR               14   11003
   5     XOR               13   11004
   6     XOR               12   11005
   7     XOR               11   11006
   8     XOR               10   11007
   9     XOR            11000   11008
  10     AND            11000   11008
  11     AND               10   11007
  12     AND               11   11006
  13     AND               12   11005
  14     AND               13   11004
  15     AND               14   11003
  16     AND               15   11002
  17     OR       - 1       2
  18     OR       - 2       3
  19     OR       - 3       4
  20     OR       - 4       5
  21     OR       - 5       6
  22     OR       - 7       7
  23     OR       - 7       8
  24     OR       - 8       9
ELEMENT    17 HAS ZERO FANOUT.
ELEMENT    18 HAS ZERO FANOUT.
ELEMENT    19 HAS ZERO FANOUT.
ELEMENT    20 HAS ZERO FANOUT.
ELEMENT    21 HAS ZERO FANOUT.
ELEMENT    22 HAS ZERO FANOUT.
ELEMENT    23 HAS ZERO FANOUT.
ELEMENT    24 HAS ZERO FANOUT.
```

Table D.4 Wiring Description of 8-bit Incrementer

## REFERENCES


1. A. Mohsseni, M. Patel, and F. J. Hill, SCIRTSS User' Manual Version 4.0, Department of Electrical and Computer Engineering, University of Arizona, 1985. University of Arizona, 1985.

2. F.J. Hill and G.R. Peterson, Digital Systems: Hardware Organization and Design, 2nd ed., John Wiley and Sons, New York, 1987.

3. Z. Kohavi, Switch and Finite Automata Theory, McGraw -Hill, 1978.

4. M. A. Breuer and A. D. Friedman, Diagnosis and Reliable Design of Digital Systems, Computer Science Press, Inc., Woodland Hills, Calif.

5. H. Fujiwara, Logic Testing and Design for Testability, MIT Press, 1985.

6. A. Miczo, Digital Logic Tesing and Simulation, Wiley, 1986.

7. Chiang, Manning, Metze, Fault Diagnosis of Digital System, John Wiley & Son, 1970.

8. A. Mohsseni, Intelligence Driven Automatic Test Sequence Generator for VLSI, Ph.D. Dissertation, University of Arizona, 1984.

9. Z. Wang, "D-Algorithm Installation on VAX-11," SCIRTSS Memo Number 7, Department of Electrical and Computer Engineering, University of Arizona, 1984.

10. D. Smith, "Finding Fault: An Update on Fault Simulation", VLSI Systems Design, Vol. III, Oct., 1987.

11. P. Goel, "An implicit enumeration algorithm to generate tests for combination logic circuits", IEEE Tran. Comput. C-30(3):215-222, 1981.

12. P. D. Stigall and Y. M. Erten, "Test Generation for Combinational Logic Circuit Using Information Theory", Compt. & Elect. Engng Vol. 12, No. 1/2, pp.51-56, 1986.

13. Y. H. Chen, <u>A New Test Generation Algorithm Implementation</u>, Master Thesis, Department of Electrical and Computer Engineering, University of Arizona, 1987.

14. M. R. Patel, <u>Hardware Compiler Driven Heuristic Search for Digital IC Test Sequences</u>, Master Thesis, University of Arizona, 1985

15. <u>VAX/VMS Run-Time Library Routines Reference Manual</u>, Digital Equipment Corporation, April 1986.