

# **IMPLEMENTATION OF AN ATPG USING PODEM ALGORITHM**

**SACHIN DHINGRA**  
ELEC 7250: VLSI testing

## **OBJECTIVE:**

**Write a test pattern generation program using the PODEM algorithm.**

## **ABSTRACT:**

*PODEM (Path-Oriented Decision Making) is an Automatic Test Pattern Generation (ATPG) algorithm which was created to overcome the inability of D-Algorithm (D-ALG) to generate test vectors for circuits involving Error Correction and Translation. The aim of this project is to implement the PODEM algorithm to generate test vectors for a given fault. External tools such as HITEC/PROOFS package are used to convert a netlist of a circuit into a leveled circuit description. HITEC/PROOFS package is also used to calculate the Testability Measures required for implementation of PODEM. A sample circuit is chosen for verification purposes. Various subroutines of the PODEM algorithm are individually verified. Finally the test vectors generated by the program are compared with manual implementation of the PODEM algorithm.*

## **I. INTRODUCTION:**

The D-ALG is proven to be ineffective in generation of test vectors for circuits that involve the XOR gates with re-convergent fan-outs. These kind of circuits are commonly found in applications requiring Error Correction and Translation (ECAT). Automatic Test Pattern Generators (ATPGs) using PODEM and D-ALG are complete i.e. they will generate a vector for a fault if the fault is testable. D-ALG generates a decision structure to evaluate the value of every node in the circuit to obtain the test vectors. The drawback of this approach was discovered while generating test vectors for typical circuits like the ECAT. D-ALG takes extremely long time to generate tests for such circuits; PODEM on the other hand confines its search space only to the primary inputs. This technique proves to be much faster and efficient as compared to D-ALG

In order to describe the circuit a five valued logic is used. The five values are 0, 1, X (unknown), D (Logic 1 in good circuit and 0 in the faulty circuit), D' (Logic 0 for a good circuit and 1 for a faulty circuit). The algorithm examines all the possible input values till a test is found for a given fault. Initially all the primary inputs are unassigned, these inputs are assigned values one by one depending on the fault location & fault type (sa0 or sa1), the path chosen for fault propagation also determine the input assignment. The implications of primary inputs are evaluated every time a new value is assigned to an input. These implications are used to determine testability of the faults. If any input does not contribute in fault detection then it is dropped from the input list. So the algorithm basically assigns primary inputs with different values sequentially till a test vector for a given fault is found.

HITEC/PROOFS is a package created in the University of Illinois, Urbana, capable of fault list generation, fault simulation and fault collapsing for stuck-at faults in sequential/combinational circuits. The input to the package is a netlist which is converted into a *circuit.lev* file which describes the circuit in its leveled form along with the testability

measures. The package generates a fault list of all the stuck-at faults in the circuit and if needed perform equivalent and dominance collapsing of the faults. Although the primary objective is perform fault simulation and test pattern generation, for the project, the package is only used for levelization and calculation of the testability measures for the circuit under test. This helps us reduce the amount of coding required to implement the PODEM algorithm.

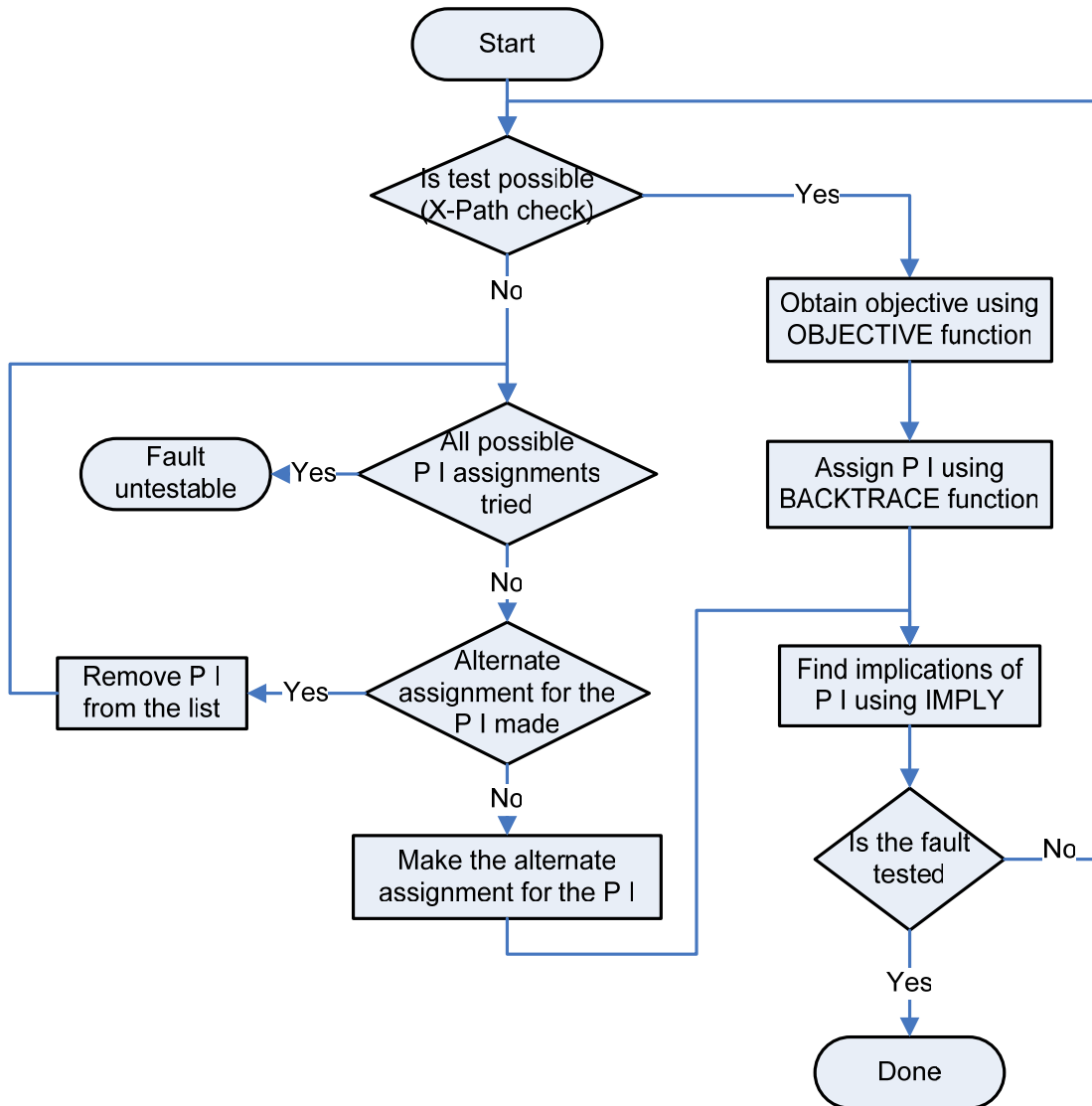
In this project PODEM algorithm is implemented in 'C' programming language to generate test vectors for stuck-at faults in a given circuit. The input to the program is a circuit netlist and its fault list. The netlist is parsed and stored as a data structure in the form of a leveled circuit. The faults are chosen one at a time from the fault list. PODEM algorithm is executed to determine a test vector which can detect the fault under consideration. The PODEM algorithm is explained in Section III. Section IV describes the implementation of the algorithm in 'C' with the help of HITEC/PROOFS. The Results are presented in Section V, followed by conclusion.

## **II. ALGORITHM:**

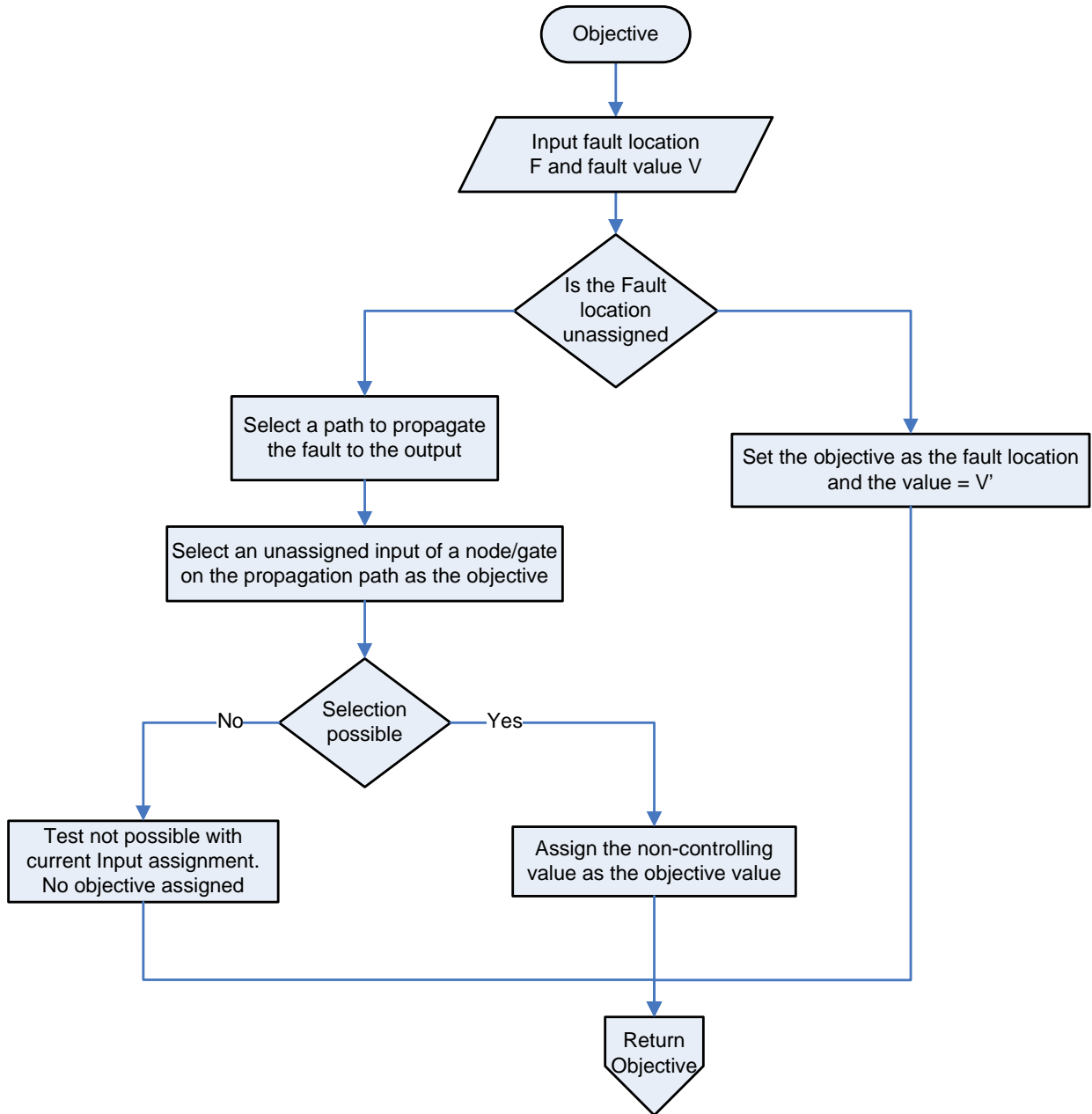
PODEM proves to be more efficient as compared to a D-ALG because it limits its search space only to Primary Inputs (PIs) of the circuits. D-ALG on the other hand has a search space comprising of all the internal nodes of the circuit along with the PIs. The first objective of the algorithm is to sensitize the fault. After the fault is sensitized the objectives are changed in order to propagate the fault to a Primary Output (PO). Function OBJECTIVE is used to determine objectives for the program. Depending on the current objective, a function called BACKTRACE is used to determine the value of one of the PIs. For every PI assigned, logic simulation is performed to check for two conditions: desensitization of the fault and disappearance of fault propagation path (also known as X-PATH CHECK). If any one of the two conditions is violated, the program backtracks and changes the value assigned to the most recent PI. This process of assigning values to PIs is repeated till PIs form a test vector or no more combinations of PIs are possible. The latter case implies that the test is untestable.

The simplified flowchart of the algorithm and its major functions (Backtrace, Forward Implication, X-Path Check and Objective) are shown in the following text.

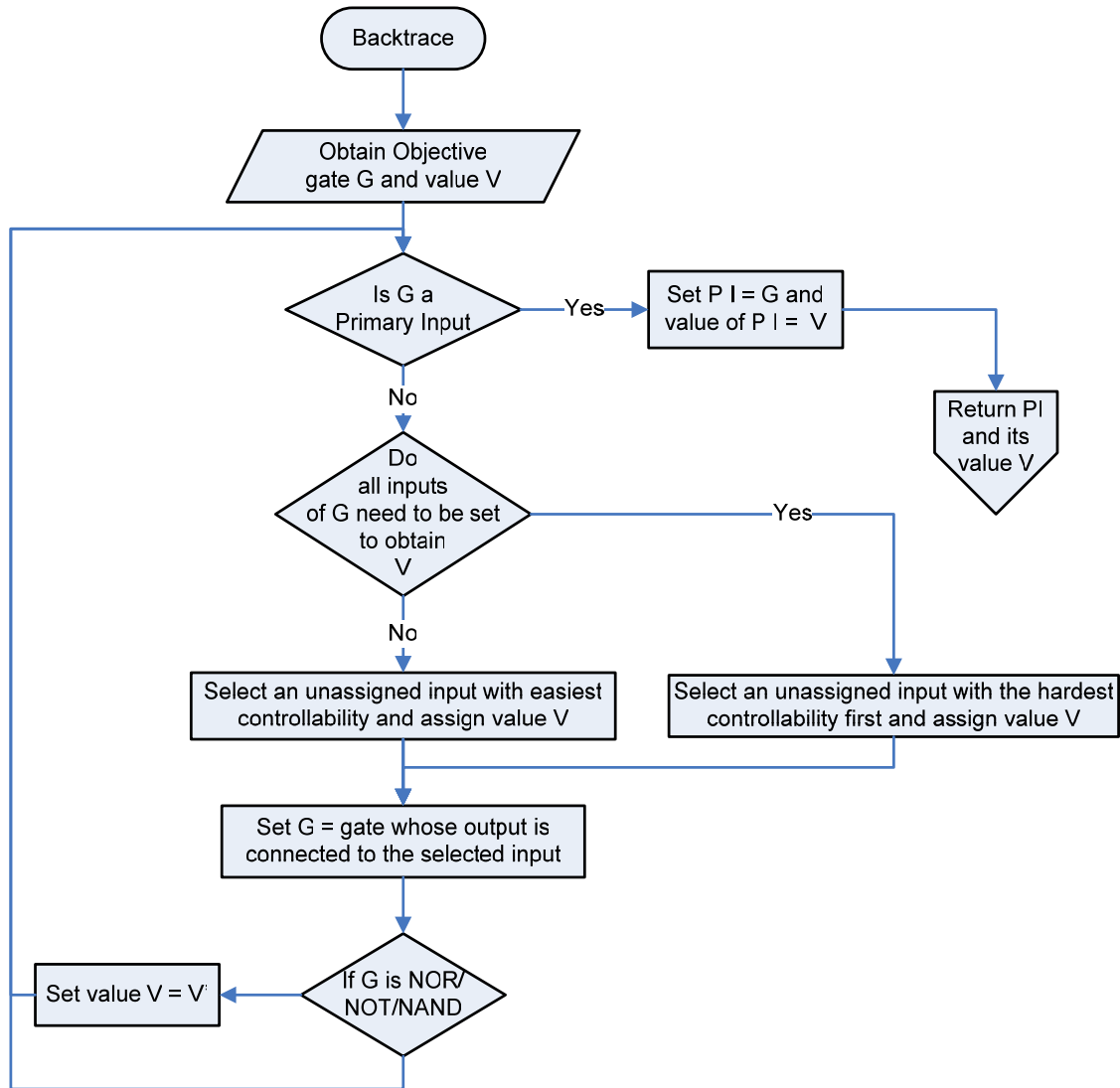
Podem:



Objective:



*Backtrace:*



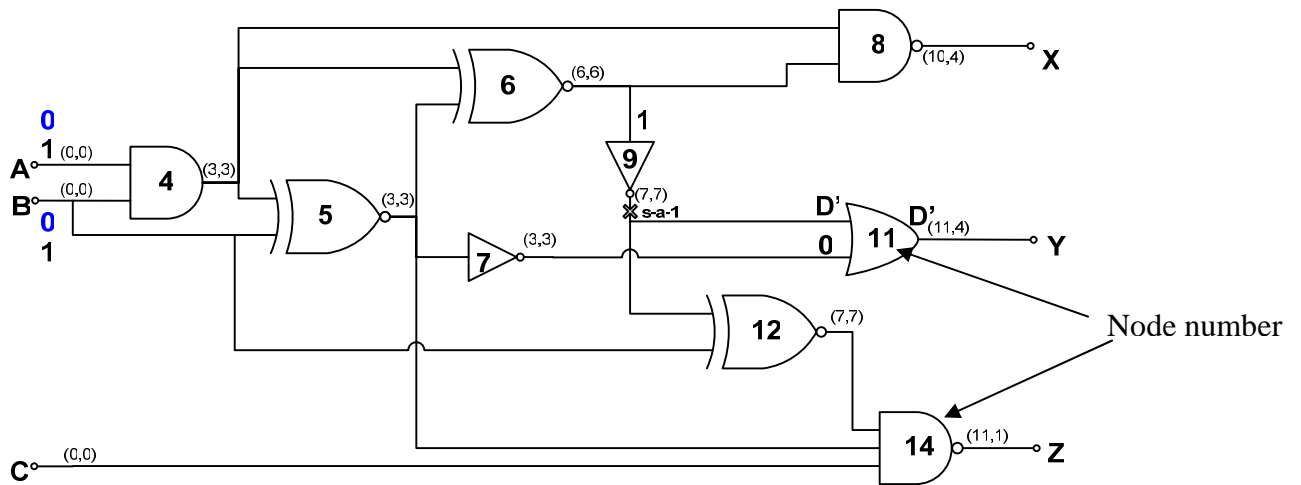
*Forward Implication (IMPLY function):*

This function is used to perform a logic simulation of the circuit depending on the values of the Primary Inputs. The values of all the nodes/gates in the circuit are evaluated. These results are then used to determine the testability of the fault. IMPLY function also checks for sensitization of the fault.

*X-Path check:*

This function checks for the availability of path for fault propagation. It essentially checks for existence of D-frontiers. This is done by checking for D/D' on one of the inputs of the gate on the fault propagation path and non-controlling/unknown values on the remaining inputs.

The following figure illustrates the implementation of PODEM to detect s-a-1 fault at the output of gate 9 shown in the circuit.



Note: The testability measures used by HITEC/PROOFS are not the same as the SCOAP measures.

**Figure 1. Illustration of PODEM Algorithm**

**Steps:**

1. Is test possible: YES
2. Objective: Set node 9 to '0'
3. Backtrace: Set B = '0'
4. Imply: node 4 = '0', 5 = '0', X = '1', 6 = '0', **9 = '1'**, 7 = '1', 14 = '1'....
5. Fault detected: NO
6. Is test possible (X-Path check): NO
7. Alternate assignment of PI: B = '1'
8. Imply:
9. Fault detected: NO
10. Is test possible (X-Path check): YES
11. Objective: Set 9 = '0'
12. Backtrace: Set A = '0'
13. Imply: 4 = '0', 5 = '1', 6 = '0', **9 = '1'**
14. Fault detected: NO
15. Is test possible (X-Path check): NO
16. Alternate assignment of PI: A = '1'
17. Imply: 4 = '1', 5 = '1', 6 = '1', 7 = '0', X = '0', 9 = '0'
18. Fault detected: YES
19. Test found: **A = '1', B = '1', C = 'X'**

### III. IMPLEMENTATION

The netlist of the circuit is provided in benchmark file format to HITEC/PROOFS package. The netlist is leveled and a new file *circuit.lev* file is created. The leveled file for the sample circuit is shown in figure 2.

```

16
10
1 1 0 0 1 4 8 ; 0 0
2 1 0 0 3 12 5 4 8 ; 0 0
3 1 0 0 1 14 11 ; 0 0
4 6 5 2 1 2 1 2 3 6 8 5 7 ; 3 3
5 4 10 2 4 2 4 2 3 6 7 14 8 ; 3 3
6 4 15 2 4 5 4 5 2 9 8 4 ; 6 6
7 10 15 1 5 5 1 11 8 ; 3 3
8 7 20 2 6 4 6 4 1 10 0 ; 10 4
9 10 20 1 6 6 2 12 11 4 ; 7 7
10 2 25 1 8 8 0 0 O 10 4
11 8 25 2 9 7 9 7 1 13 0 ; 11 4
12 4 25 2 9 2 9 2 1 14 4 ; 7 7
13 2 30 1 11 11 0 0 O 11 4
14 7 30 3 12 5 3 12 5 3 1 15 0 ; 11 1
15 2 35 1 14 14 0 0 O 11 1
END

```

**Figure 2. Leveled File**

The first integer value denotes the number of nodes of the circuit. Input/Output terminals and Gates of the circuit are known as nodes of a circuit. The second line is a garbage value and is to be ignored. The remaining lines contain information about the nodes and their testability measures. The *circuit.lev* file is appended with an “END” statement for ease of parsing. The format in which the node information is stored is shown in figure 3.

1	2	3	4	5	6	7	8	9	10	11
<i>node</i>	<i>gatetype</i>	<i>fan-ins</i>	<i>CC0 input order</i>	<i>CC1 input order</i>	<i>fan-outs</i>	<i>CO output order</i>	<i>CO</i>	<i>PO</i>	<i>CC0</i>	<i>CC1</i>

**Figure 2. Node Information Format**

1. *node*: Denotes the number assigned to a node in the leveled circuit.
2. *gatetype*: Describe the type of the node for e.g. 1/2 denote Input/Output, 6 denotes a n AND gate, 10 denotes a NOT gate etc.
3. *fanins*: Number of inputs of a node/gate
4. *CC0 input order*: List of input nodes ordered according to decreasing ‘0’ combinational controllability values

5. *CCI input order*: List of input nodes are ordered according to decreasing '1' combinational controllability values
6. *fanouts*: Number of fanouts/successors of the node
7. *CO output order*: Outputs in the order of their observability.
8. *CO*: Combinational Observability value
9. *PO*: Primary Output node; 'O' denotes an output node, other nodes are denoted by ';'.
10. *CC0*: Value of '0' combinational controllability
11. *CCI*: Value of '1' combinational controllability

The fault list is also generated using HITEC/PROOFS. The levelized circuit file is parsed and stored in data structure called node. A fault is provided as an input to the program, the program is executed which returns test vector if it exists. The following steps describe the process of generation of the levelized circuit file and the fault list.

1. Create a netlist in the format required by HITEC/PROOFS (*circuit.bench* file)
2. Type the command *do\_proofs circuit*
3. Type the command *level circuit* (generates the *circuit.lev* file to be used as input to the ATPG program)
4. Type the command *faultlist* to obtain the list of all possible stuck-at faults in the circuit
5. The fault list can be collapsed by using *equiv* and *dominators* commands to perform equivalent and dominance structural collapsing respectively.

The algorithm is implemented using the following functions and data structures:

#### *Struct Node*

The data structure 'Node' is used to store the circuit information in levelized format along with the testability measures

#### *Struct Signal*

This data structure is used to store the signal value of each node along with the flags needed by the algorithm

*void fileread(char \*name);*

This function reads the *circuit.lev* file.

*void parse(char \*buf, int i);*

This function is used by *void fileread(char \*name)* to parse the file and store it in a data structure.

*struct signal backtrace(struct signal);*

The objective (node, value) is given as input to backtrace function. A corresponding (PI, PI value) is returned by the function

*struct signal objective(struct signal);*

Depending on the fault location and fault propagation path the objective (node, value) is returned by this function

*char pathgen(int , char);*

This recursive function generates all the fault propagation paths from the fault to the primary outputs

*void imply(struct signal);*

This function evaluates the logic values of all the nodes based on the values assigned to the PIs and checks for sensitization of the fault

*char xpathcheck(struct signal);*

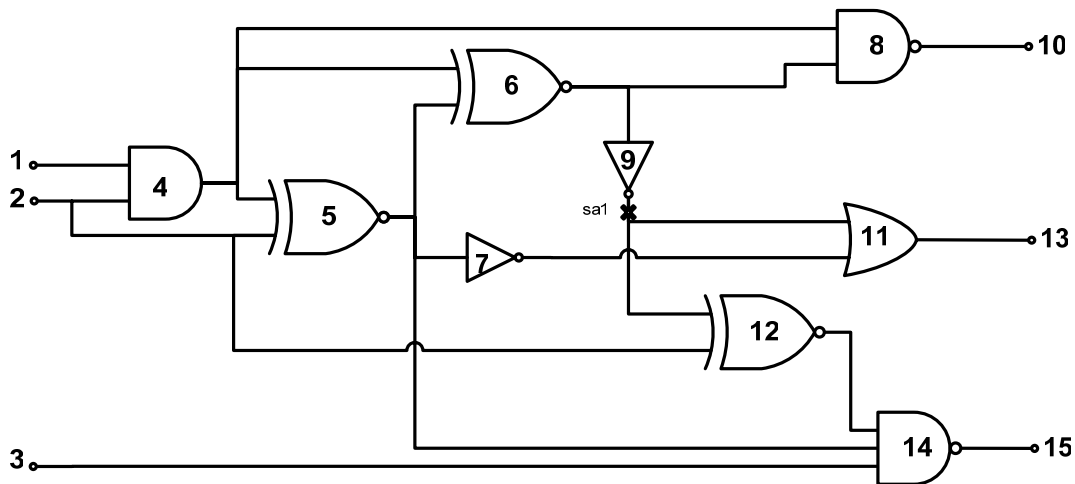
X-Path check is performed using this function. It returns a true or false value depending on the availability of X-Paths

*char podem(struct signal);*

This is the main function that implements the PODEM algorithm.

#### IV. RESULTS:

The following sample circuit was used to verify the working of the algorithm.



**Figure 3. Sample Circuit**

The command line format for the ATPG usage is shown below.

`podem leveled_circuit_filename fault_node fault_port fault_type`

*leveled\_circuit\_filename* – filename of the input file in leveled format

*fault\_node* – The location of the fault, denoted by node number

*fault\_port* – Input /Output port; ‘0’ denotes an output port and 1,2,3.. indicate the respective input ports.

*fault\_type* – Stuck at 1(Sa1) fault is indicated by a ‘1’ and Sa0 fault by a ‘0’

The program was run for sa1 fault at node 9 in circuit shown in figure 3

> podem circuit722 9 0 1

**OUTPUT:**

Circuit Description

```
1 1 0 0 1 4 8 ; 0 0
2 1 0 0 3 1 2 5 4 8 ; 0 0
3 1 0 0 1 1 4 1 1 ; 0 0
4 6 5 2 1 2 1 2 3 6 8 5 7 ; 3 3
5 4 1 0 2 4 2 4 2 3 6 7 1 4 8 ; 3 3
6 4 1 5 2 4 5 4 5 2 9 8 4 ; 6 6
7 1 0 1 5 1 5 5 1 1 1 8 ; 3 3
8 7 2 0 2 6 4 6 4 1 1 0 0 ; 1 0 4
9 1 0 2 0 1 6 6 2 1 2 1 1 4 ; 7 7
10 2 2 5 1 8 8 0 0 0 1 0 4
11 8 2 5 2 9 7 9 7 1 1 3 0 ; 1 1 4
12 4 2 5 2 9 2 9 2 1 1 4 4 ; 7 7
13 2 3 0 1 1 1 1 1 0 0 0 1 1 4
14 7 3 0 3 1 2 5 3 1 2 5 3 1 1 5 0 ; 1 1 1
15 2 3 5 1 1 4 1 4 0 0 0 1 1 1
```

Number of PIs = 3

Fault Propagation Paths

Maximum Circuit Level = 7

```
Path 9 11
Path 9 12 14
```

Fault Detected = FALSE

X-Path Check = TRUE

Objective

```
9 0
```

```
Backtrace 6 1 4 0 2 0
```

Forward Implication

```
2 0 2 0 1 0 0 1 1 1 1 0
  1 1 1
```

Implication Stack

```
2 0 0
```

Fault Detected = FALSE

X-Path Check = FALSE

Backtrack

Implication Stack

```
2 0 0
```

Implication Stack

```
2 1 1
```

```

Forward Implication
2      1      2      2      2      2      2      2      2      2
      2      2      2

```

Fault Detected = FALSE

X-Path Check = TRUE

```

Objective
9      0

```

```

Backtrace      6 1      5 0      4 1      1 1

```

```

Forward Implication
1      1      2      1      1      1      0      0      0      0      0
      0      1      1

```

```

Implication Stack
2 1 1      1 1 0

```

Fault Detected = TRUE

TEST FOUND = 1

VECTORS

```

1      1
2      1
3      2

```

The program successfully generated test vectors for most of the faults. The program could not come up with test vectors for some of the testable faults; this is caused by improper operation of the Backtrace algorithm.

## V. CONCLUSION:

PODEM uses X-Path Check and Backtrace functions to in order to efficiently generate test vectors for circuits with reconvergent XOR gate functions. The project is implemented primarily in 'C', HITEC/PROOFS package is used to generate testability measures and convert the netlist into a leveled circuit file format. All the functions of the algorithm were implemented and verified for a single circuit. Recursive functions and Dynamic memory allocation are utilized to reduce the memory requirements of the program. The algorithm successfully generates test vectors with a few exceptions which have to be debugged. In order to accurately to verify error free operation of the program, the ATPG should used to generate test vectors for different circuits.

## REFERENCES:

- [1] M.L. Bushnell, V.D. Agrawal, *Essentials of Electronics Testing for Digital, Memory & Mixed Signal VLSI Circuits*, Kluwer Academic Publishers, Boston MA, 2000
- [2] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE transactions on Computers*, Vol. C-30, no. 3, pp. 215-222, 1981
- [3] P. Goel, B.C. Rosales, "PODEM-X: An automatic test generation system for VLSI logic structures," *Proceedings of the 18th conference on Design automation*, pp. 260-268, 1981
- [4] \_\_, "HITEC/PROOFS User's Manual," 1996