INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Implementation of a Wang algorithm based ATPG for combinational logic circuits

Cunning, Steven J., M.S.

The University of Arizona, 1992
IMPLEMENTATION OF A WANG ALGORITHM BASED
ATPG FOR COMBINATIONAL LOGIC CIRCUITS

by

Steven Cunning

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

1992
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 acknowledgement of source is made. Requests for permission for extended quotation from reproduction of this manuscript in whole or in part may be granted by the 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:   [Signature]

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

[Signature]  8 Dec. 1992

Frederick J. Hill
Professor of Electrical and Computer Engineering
ACKNOWLEDGEMENTS

In the final analysis, this project certainly required a greater effort than was originally anticipated. It has been completed however, and I would like to acknowledge those who helped me see it through.

A special thanks goes do Dr. Fredrick Hill who introduced me to this project, maintained his patience with the slow development of the final result, and for his helpful suggestions. I would like to thank Ron, Lance, Mark, Harry, and Gerri, a few of my co-workers at Hughes Aircraft Co. for their support and assistance.

I would like to thank my parents, Charles and Barbara, for their encouragement toward my academic interests and for giving me the work ethic that allowed me to finish this project. And finally, I would like to thank my wife Kathleen, for her continuous support and for putting up with me through this rather difficult time.
TABLE OF CONTENTS

LIST OF FIGURES ........................................ 6
LIST OF TABLES ........................................... 7
ABSTRACT .................................................. 8
CHAPTER 1 - INTRODUCTION ......................... 9
  1.1 Automatic Test Pattern Generation ............ 9
  1.2 The Wang Algorithm .............................. 10
  1.3 Objective .......................................... 11
CHAPTER 2 - IMPLEMENTED ALGORITHM .......... 13
  2.1 A Simplified Algorithm ......................... 13
  2.2 Enhancements To The Simplified Algorithm .... 13
    2.2.1 Path Selection .............................. 14
    2.2.2 Mark Circuit Inputs Necessary for Search . 16
    2.2.3 Assignments of Don't Care Values To FOS Outputs ........................................ 17
    2.2.4 Use of Bias Values .......................... 19
    2.2.5 Use of Don't Care Values During Search . 21
    2.2.6 Determination of Additional Fault Coverage 21
  2.3 Deviations From The Wang Algorithm .......... 26
  2.4 The Complete Testgen Algorithm ............... 27
  2.5 Illustrated Example of Testgen Algorithm .... 30
CHAPTER 3 - TESTGEN ................................ 40
  3.1 Testgen Overview ................................ 40
  3.2 Control Flow ................................... 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Variable, Type Definitions, And Macros</td>
<td>42</td>
</tr>
<tr>
<td>CHAPTER 4 - USING TESTGEN</td>
<td>53</td>
</tr>
<tr>
<td>4.1 General</td>
<td>53</td>
</tr>
<tr>
<td>4.2 Batch Version</td>
<td>53</td>
</tr>
<tr>
<td>4.3 Interactive Version</td>
<td>56</td>
</tr>
<tr>
<td>CHAPTER 5 - RESULTS</td>
<td>60</td>
</tr>
<tr>
<td>5.1 Test Results</td>
<td>60</td>
</tr>
<tr>
<td>5.2 Achievements</td>
<td>67</td>
</tr>
<tr>
<td>CHAPTER 6 - FUTURE RESEARCH</td>
<td>69</td>
</tr>
<tr>
<td>6.1 Future Enhancements</td>
<td>69</td>
</tr>
<tr>
<td>APPENDIX A - AHPL LISTING OF EXAMPLE CIRCUIT</td>
<td>76</td>
</tr>
<tr>
<td>APPENDIX B - TESTGEN OUTPUT FOR EXAMPLE CIRCUIT</td>
<td>77</td>
</tr>
<tr>
<td>APPENDIX C - CALL TREE FOR BATCH VERSION OF TESTGEN</td>
<td>81</td>
</tr>
<tr>
<td>APPENDIX D - REFERENCE LIST FOR BATCH VERSION OF TESTGEN</td>
<td>85</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>112</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 2.1</td>
<td>FOS ORDERING COMPUTATIONS</td>
<td>15</td>
</tr>
<tr>
<td>FIGURE 2.2</td>
<td>ASSIGNMENT OF DON'T CARES TO FOS OUTPUTS</td>
<td>18</td>
</tr>
<tr>
<td>FIGURE 2.3</td>
<td>USE OF BIAS VALUES</td>
<td>20</td>
</tr>
<tr>
<td>FIGURE 2.4</td>
<td>DETERMINATION OF ADDITIONAL FAULT COVERAGE</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 2.5</td>
<td>BLOCK DIAGRAM OF TESTGEN ALGORITHM</td>
<td>28</td>
</tr>
<tr>
<td>FIGURE 2.6</td>
<td>TESTGEN ALGORITHM AREAS OF INFLUENCE</td>
<td>30</td>
</tr>
<tr>
<td>FIGURE 2.7</td>
<td>WORKLOAD VALUES</td>
<td>31</td>
</tr>
<tr>
<td>FIGURE 2.8</td>
<td>PATH SELECTION AND SENSITIZATION</td>
<td>32</td>
</tr>
<tr>
<td>FIGURE 2.9</td>
<td>ASSIGNMENT OF DON'T CARE AND BIAS VALUES</td>
<td>33</td>
</tr>
<tr>
<td>FIGURE 2.10</td>
<td>SEARCH TREE FOR EXAMPLE CIRCUIT</td>
<td>37</td>
</tr>
<tr>
<td>FIGURE 2.11</td>
<td>STATE OF CIRCUIT AFTER SUCCESSFUL SEARCH</td>
<td>38</td>
</tr>
<tr>
<td>FIGURE 3.1</td>
<td>EXAMPLE CIRCUIT FOR ILLUSTRATION OF DATA STRUCTURES</td>
<td>52</td>
</tr>
<tr>
<td>FIGURE 3.2</td>
<td>ILLUSTRATION OF DATA STRUCTURES</td>
<td>52</td>
</tr>
<tr>
<td>FIGURE 4.1</td>
<td>INTERACTIVE TESTGEN MAIN MENU</td>
<td>58</td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE 2.1 - DECISION ARRAY FOR SEARCH . . . . . . . . . . 34
TABLE 2.2 - ADDITIONAL FAULT COVERAGE FOR EXAMPLE CIRCUIT 39
TABLE 3.1 - SOURCE FILES FOR TESTGEN. . . . . . . . . . . 42
TABLE 4.1 - USER ASSIGNABLE PARAMETERS. . . . . . . . . . 55
TABLE 5.1 - TEST CIRCUIT STATISTICS . . . . . . . . . . . 61
TABLE 5.2 - TEST RUN TIME LIMITS. . . . . . . . . . . . . 62
TABLE 5.3 - TEST RESULTS. . . . . . . . . . . . . . . . . . 63
TABLE 5.4 - ADDITIONAL FAULT COVERAGE STATISTICS. . . . 67
ABSTRACT

Automatic Test Pattern Generation (ATPG) is the process of generating test vectors for a circuit given only the physical description of that circuit. Many of today's ATPG systems are based on the D-Algorithm, PODEM, or FAN. This research is focused on the development and use of a more time efficient ATPG system for combinational logic circuits. The ATPG system developed by this research will be based on the Wang algorithm which uses a 9-value calculus. The fault model used is the single stuck-at fault model.

Two versions of the system will be developed. The batch version will attempt to determine tests for all single stuck-at faults in the given circuit. The interactive version will allow the operator to select a single fault to be tested. The system will be written in C and developed for the MS-DOS microcomputer environment.
CHAPTER 1. INTRODUCTION

1.1 Automatic Test Pattern Generation

Today's integrated circuit packages present a test problem that is characterized by highly complex circuits with limited access. In order to effectively test these circuits an Automatic Test Pattern Generation (ATPG) system is necessary. The LSI and VLSI level circuits of recent years have shown the need for more efficient ATPG systems [1].

In general, an ATPG system involves fault modeling and reduction, test pattern generation, fault simulation, and fault coverage evaluation. The fault model selected will reflect the class of faults to be screened such as single stuck-at or stuck open. The fault reduction involves determination of equivalent faults from the circuit description. The test pattern generation process will be based on a particular algorithm (e.g. the D-Algorithm) and will develop a test for a single fault. After a test vector has been developed the circuit may then be simulated to determine all faults that will be detected by the test. Finally, the fault coverage can be evaluated and compared to a previously set goal. If the coverage goal has been met, the ATPG process may be terminated. If the coverage goal is not met, an as yet untested fault should be selected and control should return to the test pattern generator.
The ATPG process described above consists of many separate functions. These functions should interact effectively to produce an efficient ATPG system that is capable of meeting present test generation demands.

1.2 The Wang Algorithm

The automatic test pattern generator resulting from this research project is based on an algorithm developed by Dr. Xiaolin Wang. This algorithm, hereafter referred to as the Wang algorithm, utilizes a 9-value calculus and is described in detail in [2].

The general steps of the Wang algorithm are:

1. Selection of a sensitization path that will allow the fault to be observed at a circuit output.

2. Sensitization of the path in terms of the 9-value calculus which avoids the problems caused by reconvergent fan out.

3. Propagation of the values required to sensitize the path in both the backward and forward directions.

4. Search over circuit inputs to find an input vector that is consistent with the sensitization path values. It should be noted that this final step is similar to PODEM with the advantage of having had the search space reduced by the preceding steps.

The implementation resulting from this project is a simplification of the Wang algorithm and will be referred to as Testgen. The Testgen algorithm is described in detail in the following chapter.
1.3 **Objective**

The objective of this project is to develop a working implementation of an automatic test pattern generator for combinational logic circuits that is based upon the Wang algorithm. The ATPG will take as its input the gate list file output of the AHPL hardware compiler. The code shall be written in C with speed and portability the primary objectives. There will be an interactive and a batch version. The interactive version will allow the user to select individual faults for which tests are desired. The batch version will attempt to produce tests for every single stuck-at fault in the given circuit.

The result of this project, Testgen, should be a functional automatic test pattern generator that is competitive with currently existing systems. The goal here is to compare the Wang algorithm with algorithms such as the D-algorithm and PODEM. This comparison should ideally be accomplished through direct experimentation if it is possible to obtain implementations of these algorithms for the PC environment. If this is not possible, an alternate approach would be to compare Testgen with the published results of other ATPG systems that have been run against the same test circuits.

Finally, Testgen should also provide a framework for further research in the development of this test pattern
generation algorithm.
CHAPTER 2. THE IMPLEMENTED ALGORITHM

2.1 A Simplified Algorithm

The starting point for the Testgen algorithm is a simplification of the Wang algorithm described by F. Hill in [3]. The algorithm described by Hill is essentially a no frills Wang algorithm. In other words, the required portions of the Wang algorithm have been extracted to produce a straightforward procedure that allows ease of initial implementation.

The first step of the Simplified algorithm is selection of a sensitization path that will allow propagation of the fault to a circuit output. The second step is to sensitize the path in terms of the 9-value calculus. The next step is to compute the values of all constrained network points resulting from the path sensitization. And finally, a PODEM like search technique is used to determine an input vector that is consistent with the sensitized path.

This simplified algorithm provides a working baseline from which enhancements may be added and evaluated.

2.2 Enhancements To The Simplified Algorithm

In order to improve the performance of the simplified
algorithm, an initial set of enhancements have been incorporated into Testgen. The following sections describe these enhancements and how they will improve the test pattern generator. Also listed in each section is a description of how these enhancements are called from within Testgen.

2.2.1 Path Selection

The first enhancement is in the area of path selection. The simplified algorithm contains no path selection criteria resulting in random path selection. In Testgen, a simple path selection criteria has been added. The selection criteria implemented is one that favors a combination short path length, minimal lead lines, and most easily controlled lead lines. The term lead line is used in this paper to refer to any line leading to a gate that lies on the sensitization path provided that the line is not actually part of the sensitization path.

When selecting a path, each fan out stem encountered requires a decision as to which output to select. The proper path selection is accomplished by organizing fan out stem output lines in the order to be selected. This ordering is accomplished by assigning a workload value to each output line and selecting the smallest workload values first.

The computation of the workload values is a two pass
process. First, a forward pass over the circuit it made to compute the distance to a circuit input at all lines. Circuit inputs are assigned a distance of one. Fan out stem outputs are assigned the distance of the fan out stem input plus one. Gate outputs are assigned the sum of the distances of the gate inputs plus one. The input distances are a crude estimate of the controllability of each line.

**Input distances**

- Circuit inputs = 1
- FOS outputs = distance of FOS input + 1
- Gate outputs = sum (gate inputs) + 1

<table>
<thead>
<tr>
<th>x</th>
<th>= x + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>= x + 1</td>
</tr>
<tr>
<td>z</td>
<td>= x + 1</td>
</tr>
</tbody>
</table>

**Workload values**

- Circuit outputs = 1
- FOS inputs = min(FOS outputs) + 1
- Gate inputs = gate output + 1 + sum (remaining gate inputs)

<table>
<thead>
<tr>
<th>x</th>
<th>= min(x,y,z) + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>= min(x,y,z) + 1</td>
</tr>
<tr>
<td>z</td>
<td>= min(x,y,z) + 1</td>
</tr>
</tbody>
</table>

**FOS ORDERING COMPUTATIONS**

Figure 2.1

A summary of the forward pass operations are shown in Figure 2.1 (a). The second pass is made backward starting from the circuit outputs. All outputs are assigned a workload value of zero. Fan out stem inputs are assigned the minimum of the workloads on all fan out stem outputs plus one.
Choosing the minimum represents selecting the easiest path at the fan out stems. For gate input lines, the workload value is computed by adding the workload value of the gate output plus one to the sum of the workload values of the remaining gate inputs. The former represents the difficulty in reaching a circuit output from this gate and the latter represents the difficulty in sensitizing this gate. The backward pass operations are summarized in Figure 2.1 (b).

It should be noted that this selection criteria finds the local minimum only. In other words, after a path has failed, backtrack only proceeds to the most recent fan out stem. If there are untried outputs at this fan out stem, a path is selected based solely on the workload values at this fan out stem.

This enhancement adds no overhead during test generation since the computation of workload values and ordering of fan out stem outputs is performed once at program start up by a call to the function orderfos(). For an initial implementation, this procedure seems to provide adequate improvement in the path selection mechanism.

2.2.2 Mark Circuit Inputs Necessary For Search

Another search space reduction technique is to limit the circuit inputs to be considered by the final search. After a
path has been selected, the portion of the circuit that may be involved in propagating the fault to a circuit output has been defined. We can take advantage of this knowledge to identify the circuit inputs from which the sensitization path can be reached. All remaining unmarked inputs are those from which the sensitization path cannot be reached. Since the set of unmarked lines will have no bearing upon the success or failure of the search process, they should not be considered by search. This elimination of unnecessary inputs will reduce the search space and remove useless gate evaluations and logic value propagation.

The identification of unnecessary inputs is accomplished by the function \textit{markinputs()}. This function is called from \textit{gentest()} immediately following path selection and sensitization.

\textbf{2.2.3 Assignment of Don't Care Values To FOS Outputs}

This enhancement is designed to confine both the search space for final search and the propagation of path lead line values. During the process of following path lead lines backward for the purpose of marking circuit inputs that reach the path, we can also mark all fan out stem outputs that reach the path. This information may then be used to assign don't care values to all fan out stem outputs that do not reach the
sensitization path. This will restrict the propagation of path lead line values and limit search to only the required areas of the circuit.

The concept is illustrated in Figure 2.2. The area containing the sensitization path is the required search space. The don't care values assigned to all fan out stem outputs that lead out of the required area will stop the propagation of path lead line values as well as propagation of search values. Also, as described in the previous section, circuit inputs that are outside of the required area are not considered by search.

The function assXL() performs the assignment of don't
care values described above. The function `assX2()` performs the assignment of don't care values originating from path lead lines. These functions are called from `gentest()` immediately following the marking of the unnecessary inputs.

### 2.2.4 Use of Bias Values

The use of bias values is an enhancement designed to reduce the search space for the final search. The bias values are designations that indicate that search may stop if assigning the proper logic value to a line that has been assigned a matching bias value. The bias values used are T for 1 biased and Z for 0 biased.

Figure 2.3 shows an example of the use of bias values. The heavy lines indicate the sensitization path. Gates 5 and 6 do not have output values that uniquely determine their input values. For gate 5 a 0 on any combinations of input lines will satisfy the path sensitization requirements. The same can be said for a 1 on any combination of input lines for gate 6. This situation is indicated by assigning the proper bias values to the inputs to these gates. The bias values are then propagated backward in all cases that an assignment of a matching logic value will guarantee path sensitization.
The procedure used by search regarding bias values may be summarized by two rules:

A. If the logic value assigned to a line matches that lines bias value, the required value for path sensitization is guaranteed and search may stop forward propagation of logic values from this line.

B. If the logic value assigned to a line does not match that lines bias value, neither satisfaction of or conflict with, the required sensitization value is guaranteed and search must continue forward propagation of logic values from this line.

In the example above, search would not be required to evaluate gate 2 if an assignment of a 1 was made at either input to this gate. If however, a 0 was assigned, the
propagation of logic values would have to continue and gate 2 would be evaluated.

Bias values are assigned to the good and bad networks independently by the function assbias(). This function is called from gentest() immediately after assignment of don't care values.

2.2.5 Use of Don't Care Values During Search

The last search space reduction strategy is the use of don't care values during the final search. If, during the search process, a logic value assignment uniquely determines the output of a gate, all other unassigned gate input lines may then be assigned don't care values. The generated don't care values may then be propagated backward toward the circuit inputs. This backward propagation of don't care values produces the desired search space reduction.

The function srchassX() is called from search() to perform the assignment and propagation of don't care values whenever possible.

2.2.6 Determination of Additional Fault Coverage

After a successful test generation is completed, we know that the original fault will be tested by the generated test vector. In addition there will be other faults that will be
tested by the same test vector. The enhancement described here is intended to determine some of these additional faults. This enhancement will be used by the batch version of Testgen and will be considerably faster than generating a new test vector for each individual fault.

The technique implemented in Testgen is a path oriented procedure that utilizes the fact that a sensitized path has already been generated. This procedure looks backward from the fault site and from path lead lines. Any values that are required to excite the fault or to sensitize the path identify other faults that may also be tested.

Due to the fact that reconvergent fan out may invalidate the sensitization of a path, all reconvergent fan out along the sensitization path must be identified and characterized. If a fan out stem exists along the sensitization path, then the existence of reconvergent fan out is possible. The fan out lines may be traced forward to determine if there are any points at which they reconverge with the sensitization path. Once these points have been identified, they may be characterized by whether the good or bad network value is required for path sensitization. If the good network value is required, the path will remain sensitized even without the existence of the primary fault. Path sensitization that relies upon the bad network value when the good and bad network values differ implies that the existence of a fault
prior to the fan out stem is required for path sensitization. The area between the fan out point and the reconvergent point that requires a fault for path sensitization will be considered a dead zone from which no additional faults will be propagated to the circuit output. Figure 2.4 illustrates the concept of dead zones.

The implemented procedure follows all lines that fan out from the sensitization path forward in search of points of reconvergence that rely upon the bad network value as described above. If found, the dead zones are defined by
marking the fan out stem and the gate at which reconvergence occurs. After all dead zones are marked, the search for additional faults tested will begin. This task begins by looking back from the fault site and then proceeds forward along the sensitization path looking back from lead lines. This continues until a marked fan out stem or a circuit output is reached. Reaching a circuit output implies that the entire sensitization path has been traversed and that no dead zones exist. If a marked fan out stem is reached, the procedure will jump to the circuit output at the end of the sensitization path and begin working backward until a marked reconvergent gate is reached.

The procedure to follow path lead lines backward will continue through gates as long as there are gate inputs that are required. This procedure will also continue through fan out stems as long as there is only one fan out stem output that reaches the sensitization path. If multiple fan out stem outputs reach the path, the sensitization of the path may be invalidated by the existence of a fault at the fan out stem input. This is another example of reconvergent fan out. The difference here is that the fan out is not on the sensitization path for the primary fault. In order to determine whether the path remains sensitized in the presence of the fault at the fan out stem input, the network with this fault injected would have to be simulated. Rather than re-
simulate portions of the circuit, Testgen will simply stop the
backtrace from a particular lead line when a fan out stem
meeting the criteria described above is encountered.

Illustrated in Figure 2.4 is a weakness of the
implemented procedure. This weakness exists if serial and
disjoint dead zones exist. The faults tested between these
zones will not be discovered by the implemented procedure.

The procedure described above is initiated by a call to
faultstested() from results(). The function results() is
called from search() whenever search is successful and a test
vector has been generated. The function faultstested() calls
simulate() and recon() for the determination and
identification of any dead zones. After the dead zones have
been identified, faultstested() and the function lookback() are used to identify and list the additional faults tested by
the generated test vector.

From the preceding paragraphs it is apparent that a great
deal of extra work is involved in determination of additional
fault coverage. Reconvergent fan out must be explicitly dealt
with and portions of the circuit must be re-simulated. Having
to deal with reconvergent fan out in itself seem contrary to
the idea behind using the 9-value calculus. At this point it
seems that a strong argument may be made for pairing Testgen
with a fault simulator. This idea will be discussed in detail
in Chapters 5 and 6.
2.3 Deviations From The Wang Algorithm

The differences between the Testgen and Wang algorithms described in this section are considered to be of a fundamental approach type. The enhancement differences have been described in the previous section.

The first difference is in the area of path selection. The Wang algorithm uses correlation values to dynamically calculate workload values. The workload values are used to determine the path most likely to be successfully sensitized and with the least amount of backtrace search. This determination has been omitted from the Testgen algorithm since the assumption has been made that it will be possible to sensitize most paths from the fault to a circuit output. In other words, the added complexity and overhead is considered to be unjustified for this implementation.

The Wang algorithm also uses the correlation values to determine when to terminate backtrace. If the assigned path sensitization values do not match the correlation values, the backtrace must continue. The continuation of backtrace may result in the need for search in order to find matching values. Finding values that match the correlation values during backtrace guarantees that the final forward search will be successful. This guarantee of success is achieved by
moving a large portion of the search into the backtrace. It is argued here that this is not the most efficient manner of search. During backward search, all possible inputs to each gate must be enumerated in order to keep track of those tried thus far. This constitutes a large overhead that the backtrace procedure must endure. Also, by searching in the backward direction, inputs that may be physically impossible to realize will be enumerated and attempted. With the forward search the circuit structure will automatically restrict the search. Also, the overhead of adjusting guide values, and as a result workload values, will again burden the backtrace routines. With all of the search restricted to the forward propagation of input values it will be possible for the search to fail. However, it is believed that the cost of backtrack and repeated search will not be as great as the overhead of search during the backtrace in the majority of cases.

2.4 The Complete Testgen Algorithm

Combining the simplified algorithm and the enhancements described in the previous section produces the complete Testgen algorithm. A block diagram representation of the algorithm is illustrated in Figure 2.5. Block (1) is the static fan out stem ordering that is performed once during circuit initialization. This ordering enables the path
selection enhancement previously described. The loop containing the 'FOS' and 'GATE' outputs of the of the 'NEXT ELEMENT TYPE' decision makes up the path selection and sensitization procedure. Embedded in this loop is the backtracking that takes place when a path fails to be sensitized or when final search fails. Testgen will follow
this loop until backtracking leads back to the primary fault site or until a circuit output has been reached. The former indicates an undetectable fault and the latter indicates that a complete path has been selected and sensitized. When the path has been selected the left most branch of the diagram will be traversed.

Blocks 2 and 3 map out and define the area of the circuit that is related to the primary fault site and selected sensitization path. The boundaries of this area are defined through the assignment of don't care values to circuit inputs and fan out stems. Block 4 completes the 9-value assignments by assigning don't care values that originate at path lead lines and behind the fault site. Block 5 indicates the assignment of bias values.

Block 6 is the final forward search over the circuit inputs. Blocks 7 and 8 are the determination and listing of additional fault coverage. These steps are only performed in the batch version of Testgen.

Figure 2.6 is a graphical representation of the circuit areas in which the different steps of the Testgen algorithm take place. The X and solid line indicate the primary fault site and sensitization path respectively. Area (1) around the path is the area of propagation of path lead line logic values. Area (2) indicates the area for assignment of don't care and bias values. Area (3) is the portion of the circuit
covered by the final forward search. Area (4) is the portion of the circuit that is not related to this particular fault and path. This diagram illustrates an ideal situation. In reality, the boundaries between areas will be somewhat fuzzy as these areas will most certainly overlap.

2.5 Illustrated Example of Testgen Algorithm

The steps of the Testgen algorithm will now be
illustrated through the use of a simple example. This circuit is similar to the reconvergent fan out example circuit from [4]. Figure 2.7 shows the example circuit after the workload values for fan out stem ordering have been computed.

For this example we will chose the output of gate 2 Stuck-at-1 as the primary fault. Figure 2.8 shows the selected and sensitized path. The path selection process for this fault encounters two fan out stems. At the first fan out stem the workload values are equal due to the symmetry of this circuit. For this fan out stem the selection of an output is arbitrary. For the second fan out stem, the output leading to
gate 11 is obviously the shortest and most easily sensitized path. As a matter of fact, the path through gate 10 is impossible to sensitize. In this case the workload value at this fan out stem causes selection of the proper path.

The next set of steps are to mark the inputs reaching the sensitization path and the assignment of don't care and bias values. In this example all inputs reach the sensitization path. Don't care values are assigned in the good network to fan out stem outputs leading from gate 4 to gates 9, 10, and 12 since these lines do not reach the sensitization path. Don't care values are also assigned to path lead lines at
ASSIGNMENT OF DON'T CARE AND BIAS VALUES

Figure 2.9

gates 2, 6, and 8. These values are then propagated in the backward direction. Bias values are generated at gates 2, 4, and 7 due to the 0 values at the outputs of these NOR gates. These bias values are also propagated in the backward direction. Figure 2.9 shows the state of the circuit after the completion of these steps.

The next step is the forward search process. During the search process a decision must be made after each assignment of a search value. This decision is whether to continue evaluations, stop evaluations due to consistency between the search value and the backtrace value, or to backtrack the
search due to a conflict between the search value and the backtrace value. These decisions are made independently for the good and bad networks. Table 2.1 defines these decisions.

<table>
<thead>
<tr>
<th>Backtrace Value</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>T</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Value</td>
<td>S</td>
<td>X</td>
<td>S</td>
<td>C</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>C</td>
</tr>
</tbody>
</table>

**DECISION ARRAY FOR SEARCH**

Table 2.1

For the example circuit, the final search will start by assigning a zero value to input A. This line has a good network backtrace value of X which is consistent with the search value. As a result search need not propagate values in this network. Since there has been no backtrace value assigned to this line in the bad network, the search value propagation must continue in this network. The input to gate 1 again has no backtrace value assigned resulting in the assignment of the search value to this line. The input to gate 5 has an X backtrace value and again, search may stop. Since the input line to gate 1 has been changed an attempt to evaluate this gate is made. This evaluation fails since the other input line is not yet set and search must stop along this path also.
Search now assigns a 0 value to input B. Since no backtrace values were assigned, search must propagate values in both networks. At the input to gate 4, the backtrace value is X/T resulting in the need to assign a value to the bad network only. The opposite is true at the input to gate 2 where only the good network value need be assigned. At the input to gate 3 the lack of backtrace values requires assignments in both networks. Evaluation of gate 2 in the good network and gate 4 in the bad network results in a 0 value at the output of each gate due to the T values on the other inputs. Since these values are consistent with the backtrace values on these lines, search may stop propagations along both paths. At gate 3 the good network evaluates to a 1 which is consistent with the backtrace value at the output of that gate and the bad network cannot be evaluated due to the lack of a value on the other input. Search may now stop propagation of values along this path as well.

Due to the T bias value in the good network at input C, search will assign a 1 value to this input. The lack of bad network backtrace value requires that the search value be propagated in this network. The input to gate 1 must be assigned resulting in the attempted evaluation of gate 1. This evaluation produces a 0 at the output of gate 1. The T bias value on this line requires that search continue propagation of values along this path. Evaluation of gate 4
produces a 1 which conflicts with the backtrace value at this line.

Search must now backtrack and assign a 0 to input C. This value requires propagation in both networks. At the input to gate 1 the bad network is assigned and the gate is evaluated. This produces a 1 value that is consistent with the T bias value at this line. At gate 2 the good network is assigned and the gate is evaluated. This evaluation produces a 1 which fails to excite the fault.

Search must again backtrack, this time to input B. Input B is then assigned a 1 value. The inputs to gates 4 and 2 need not be assigned due to the consistent backtrace values of X/T and T/X respectively. The input to gate 3 is assigned values in both networks and the gate is evaluated. The bad network evaluation fails and the good network evaluation produces a 0. The 0 value is inconsistent with the T bias value on this line which requires search to continue along this path. Evaluation of gate 7 produces a 1 due to the T value on the other input. This value is consistent and search stops prorogations along this path.

Input C is again assigned a 1 value. At the input to gate 1 the bad network is assigned and gate 1 is evaluated. This evaluation produces a 0 which is inconsistent with the T backtrace value on this line. Continuing, gate 4 is evaluated producing a 0 which allows search to stop along this path.
The inputs to gates 2 and 7 need not be assigned due to the T/X and T/T backtrace values assigned to these lines.

Finally, a zero may be assigned to input D. At the input to gate 6 the backtrace and search values are consistent. At the input to gate 3 the bad network value must be assigned and the gate evaluated. The evaluation produces a 0 which is inconsistent with the T bias value assigned to this line. Search must continue and evaluate gate 7. This evaluation produces another 0. This value is consistent and search may stop along this path.

Since assignments have been made to all primary inputs without conflict a valid test vector has been generated. The test vector is ABCD = 0110 and the result of the test may be observed at the circuit output Z. Figure 2.10 shows the search tree for the search process described above. Figure
2.11 shows the state of the circuit after the successful test generation process.

The final step is the determination of additional fault covered by the generated test. After performance of the procedure described in section 2.2.6, the additional faults determined to be tested are listed in the left hand column of Table 2.2. The right hand column of this table lists the faults that are tested but overlooked due to the fan out stem with multiple outputs that reach the sensitization path.
ADDITIONAL FAULT COVERAGE FOR EXAMPLE CIRCUIT

Table 2.2

<table>
<thead>
<tr>
<th>ADDITIONALLY TESTED FAULT LIST</th>
<th>OVERLOOKED FAULT LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-A-1 AT OUTPUT OF GATE 8</td>
<td>S-A-0 AT THE INPUT OF GATE 7 FROM GATE C</td>
</tr>
<tr>
<td>S-A-1 AT INPUT OF GATE 11 FROM GATE 8</td>
<td>S-A-0 AT THE OUTPUT OF GATE C</td>
</tr>
<tr>
<td>S-A-1 AT INPUT OF GATE 11 FROM GATE 7</td>
<td></td>
</tr>
<tr>
<td>S-A-1 AT OUTPUT OF GATE 11</td>
<td></td>
</tr>
</tbody>
</table>

Appendix A contains the AHPL description for the example circuit used in this section. Appendix B contains the output of the batch version of Testgen when run against this example circuit. In this output file the first fault listed for each generated test vector is the primary fault.
CHAPTER 3. TESTGEN

3.1 Testgen Overview

Testgen is an automatic test pattern generator for combinational logic circuits. The fault model used is the single stuck-at fault model where faults on fan out stem outputs are considered to be unique and isolated from the fan out stem input. Stuck at faults on the circuit inputs are also considered.

The system was implemented in the IBM PC DOS environment and written in the C programming language [5]. Testgen was compiled using Microsoft's C Compiler Version 6.0 [6][7]. The large memory model was chosen to handle the large memory requirements and to improve portability. The make file, testgen.mak, used to compile Testgen will be provided with the source code.

There are two versions of Testgen. One is a batch version that will attempt to find tests for all single stuck-at faults in the given circuit. The other version is an interactive version that allows the user to select individual faults through a series of menus. Both version use the gate list file (filename.glf) output of HPCOM [8] as the circuit description input file. HPCOM is a hardware compiler that
will generate a gate list description of a circuit from the AHPL description of that circuit.

The present implementation of HPCOM requires that at least one control flip-flop be defined. The result is that it is impossible to automatically create a gate list that contains only a combinational logic circuit. This requires Testgen to remove the control circuit by making two passes over the gate list. The first pass reads in the entire gate list and identifies the control flip-flops. All lines connected to these flip-flops are then followed until the complete control circuit has been identified. The second pass re-reads the gate list skipping over the control circuit elements. This produces the desired combinational logic circuit. The tasks described above are performed by the functions `initread()`, `listcc()`, and `rdfile()`. Future revisions of HPCOM may lift the control circuit requirement. If so, Testgen has been verified to correctly read gate lists both with and without control circuits.

Table 3.1 lists the source modules for Testgen. The file `testgen.h` contains function declarations and identifies the source file in which each function is located. Although many of the modules used by the batch and interactive versions use share the same name, the modules are not interchangeable.
For both versions of Testgen, reading the input file, initialization of the circuit, and identifying the primary fault is accomplished in main(). The function gentest() is then called from main() to control the test generation process. Refer to Appendix C for a compiler generated call tree for the batch version of Testgen.

### 3.3 Variables, Type Definitions, And Macros

In this section the most important program elements will be described. This should allow future researchers to gain a basic understanding of the workings of Testgen.

All macros are defined in the header file const.h. This file contains a MAXIMUM and MINIMUM macro and definitions for various constants that are intended to improve the readability of the Testgen source code.

All global variables are defined in the header file
globe.h. The variables glt1, glt2, and glt3 are pointers to each part of the gate list table. The variables ltable1 and ltable2 are pointers to each part of the line table. These tables were broken into parts to allow larger circuits to be loaded without exceeding the 64K byte limit for single memory elements imposed by DOS. The gate list and line tables fully define a circuit as needed by Testgen. The variables inlist and outlist are pointers to lists of the circuit inputs and outputs respectively. The sensitization path and the primary fault are also accessible through global variables. Decision arrays, counter variables for statistics, user definable memory allocation block sizes and time out limits, and a pointer to the output file stream are also global.

The type definitions for all structures are located in the header file struct.h. Each of these structures will be described in further detail below. The number of bytes used by each structure in the PC DOS environment is also listed.

typedef struct symbol {
    char          symname[11];  /*BYTES: 11*/
} symbol;  /*TOTAL: 11*/

Circuit inputs and outputs will have symbols associated with each from the AHPL circuit descriptions. These symbols are stored in the symbol structure. A structure was used here in order to simplify pointer arithmetic on arrays of symbols.
An array of these structures is pointed to by the global variable \texttt{symt}. This array contains all symbols for the given circuit.

\begin{verbatim}
typedef struct symdesc {
    unsigned int symtrow, /* BYTES: 2 */
    numcols, /* BYTES: 2 */
    numrows; /* BYTES: 2 */
} symdesc; /* TOTAL: 6 */
\end{verbatim}

This structure is used to describe the symbols used by the present circuit. Each symbol will have a number of rows and columns associated with it as described by AHPL. The number of rows and columns for a particular symbol are stored in this structure. Also stored is the index into the symbol table for access to the actual character string representing the symbol. An array of these structures is pointed to by the global variable \texttt{sdt} and contains descriptions for all symbols in the given circuit.

\begin{verbatim}
typedef struct btrack {
    size_t lindex; /* BYTES: 2 */
    char gb; /* BYTES: 1 */
} btrack; /* TOTAL: 3 */
\end{verbatim}

This structure is used as an element in backtrack lists. When line values are changed due to path sensitization, assignment of don't care or bias values, or due to the final search, the indexes into the line table of the lines that have
been changed are stored in lists so that the line values may be reset if it is necessary to backtrack. For lines changed during backtrace, including don't care and bias values, the lists are stored at the sensitization path lead line from which the changes originated. During the final search the line indexes are stored at the input element from which the change originated. The \texttt{gb} field is a flag to indicate whether the good, bad, or both network values have been changed.

\begin{verbatim}
typedef struct btqelem {
    size_t index, /* BYTES: 2 */
    lindex; /* BYTES: 2 */
    char gb; /* BYTES: 1 */
} btqelem; /* TOTAL: 5 */
\end{verbatim}

The \texttt{btqelem} structure is used as an element in various queues during the backtrace process. These queues are local to the function using them and exists only when the function is active. The element differs from the \texttt{btrack} structure only in the fact that it identifies a gate as well as a line.

\begin{verbatim}
typedef struct inputelem {
    size_t index; /* BYTES: 2 */
    btrack *btlist; /* BYTES: 4 */
    size_t numlines; /* BYTES: 2 */
} inputelem; /* TOTAL: 8 */
\end{verbatim}

The \texttt{inputelem} structure is used in a list to identify the circuit inputs. The index field is an index into the gate list table for the input element. The \texttt{btlist} pointer points
to a list of indexes into the line table of lines that were changed during search. The length of the list is stored in numlines.

typedef struct measelem {
    long int distin, /* BYTES: 4 */
    diffest; /* BYTES: 4 */
} measelem;

The measelem structure is used by the function orderfos() to store measures used to order the circuit fan out stem outputs. One element is allocated for each line in the circuit. This memory is freed prior to exiting orderfos().

typedef struct ldesc1 {
    unsigned int to : 16; /* BYTES: 2 */
    unsigned short gval : 3,
    bval : 3,
    reachpath : 1,
    xorsecond : 1, /* BYTES: 1 */
    gsrchval : 2,
    sa0tc : 5,
    onpath : 1, /* BYTES: 1 */
    bsrchval : 2,
    saltc : 5,
    reconbound: 1; /* BYTES: 1 */
} ldesc1;

The ldesc1 structure represents part 1 of a complete line description. The to field is an index into the gate list table of the element the line leads to. The gval and bval fields are the good and bad network backtrace values respectively. The reachpath flag is set by markinputs() to
indicate if the sensitization path is reachable from the particular line. The xorsecond flag is used to indicate whether both sensitization values have been tried when sensitizing through an EXOR gate. The gsrchval and bsrchval fields are the good and bad network search values respectively. The saltc and sa0tc are the S-A-1 and S-A-0 test codes for each line. The test codes indicate whether the fault is tested, undetectable, untested, or that a time out or memory failure occurred while attempting to generate a test for this fault. The onpath flag indicates whether the line is on the current sensitization path. And finally, the reconbound flag is used to mark the boundaries of reconvergent fan out dead zones during determination of additional fault coverage for a generated test vector.

```c
typedef struct ldesc2 {
    btrack *btlist; /* BYTES: 4 */
    size_t numlines; /* BYTES: 2 */
} ldesc2; /* TOTAL: 6 */
```

The ldesc2 structure is the second part of a line description. The btlist pointer will be set to point to the backtrack list of lines changed during the backtrace for path lead lines. The numlines field will be set to the length of the backtrack list.
typedef struct gdescl {
    unsigned int elemnum,      /* BYTES: 2 */
        type : 6,
    symind : 10,      /* BYTES: 2 */
        numin : 5,
    numout : 5,
    ginsetcnt: 6,      /* BYTES: 2 */
        gnum0 : 5,
    gnuml : 5,
    binsetcnt: 6,      /* BYTES: 2 */
        row : 11,
    col : 5,      /* BYTES: 2 */
        bnum0 : 5,
    bnuml : 5,
    nout : 5;      /* BYTES: 2 */
} gdescl;
/* TOTAL: 12 */

The gdescl structure is the first part of a complete gate description. All elements of a circuit are assigned an element descriptor including fan out stems and circuit inputs and outputs. The elemnum field will contain a unique number that identifies the particular element. The element numbers for gates and circuit inputs and outputs are taken directly from the gate list file. Fan out stems are numbered sequentially starting after the largest number used in the gate list file. The type field will identify the element type. Valid element types are: INPUT, OUTPUT, AND, NAND, OR, NOR, EXOR, and FOS.

The symind indicates if a symbol is associated with the particular element. If non-zero, symind is an index into the symbol descriptor table. The row and col fields identify the elements particular row and column for a multi-dimensional symbol.
The numin field contains the number of input lines for the element. The numout field contains the number of output lines for the element as found in the gate list file. Fan out stems are not explicitly identified in the gate list file circuit description. Instead, gates are simply listed with multiple outputs. Since the circuit representation used in Testgen explicitly identifies fan out stems, the number of outputs for a multi-output gate will be transferred to the fan out stem at the gates output. The nout field is then used as the output number count after the introduction of fan out stems. For gates this number will always be 1 and for fan out stems it will always be greater than 1.

The remaining fields are used by search to keep track of the number of 1's and 0's at each elements inputs. This is done in order to save time when a gate's output is not determined. The first time a gate is reached, all 1's and 0's are counted. The next time the gate is reached, rather than recounting the number of 1's and 0's, the value of the line through which the gate was reached is simply added to the appropriate count. The ginsetcnt and binsetcnt fields are the total input lines that have values assigned in the good and bad networks respectively. The gnuml, gnum0, bnuml, and bnum0 fields are the counts of 1's and 0's for each network.
typedef struct gdesc2 {
    size_t *inlist, /* BYTES: 4 */
    olines, /* BYTES: 2 */
    *ilines; /* BYTES: 4 */
} gdesc2; /* TOTAL: 10 */

The \texttt{gdesc2} structure is used for part two of a gate descriptor. The \texttt{inlist} field points to a list of indexes into the gate list table for the elements at this gate's inputs. The \texttt{olines} field is an index into the line table for the first output line for this element. If this element has more than one output, the remaining output line indexes sequentially follow the value in \texttt{olines}. The \texttt{ilines} field points to a list of indexes into the line table for the input lines of this element.

typedef struct gdesc3 {
    size_t *outlist; /* BYTES: 4 */
} gdesc3; /* TOTAL: 4 */

The \texttt{gdesc3} structure is used for part three of the gate description. The \texttt{outlist} field points to a list of output element numbers. This structure is used during circuit initialization and reads the output numbers directly from the gate list file. After the circuit has been built, the output elements can easily be reached through the \texttt{to} field of the output line descriptor. As a result, this portion of the gate list table exists only during circuit initialization.

The preceding paragraphs describe the primary components
used by Testgen for circuit state representation. For more information refer to Appendix D which contains a complete reference list for functions, variables, macros, and type definitions that was generated by the compiler.

Figure 3.1 and 3.2 illustrate the use of most of the components described above. Figure 3.1 shows the schematic of a trivial example circuit. Figure 3.2 shows how Testgen represents the circuit structure. The fields that are not set are used to describe the state of the circuit and will be set as necessary during test generation.
EXAMPLE CIRCUIT FOR ILLUSTRATION OF DATA STRUCTURES

FIGURE 3.1

ILLUSTRATION OF DATA STRUCTURES

FIGURE 3.2
CHAPTER 4. USING TESTGEN

4.1 General

The present implementation of Testgen may be executed on any IBM compatible PC DOS system based on a 286 or later processor. The 286 processor limit is due to the options under which Testgen was compiled. Another limit that should be considered is the 640 KByte memory limit imposed by DOS. It is recommended that as much of this memory as possible be made available for use by Testgen. This will allow larger circuits to be analyzed and will improve performance.

4.2 Batch Version

To invoke the batch version of Testgen, simply enter the following command line at the DOS prompt:

>testgen filename.glf [filename.par]

The first argument is the gate list file description of the circuit as generated by the hardware compiler HPCOM. The gate list file must be located in the current directory. The output file will also be placed in the current directory. The output file will be given the same name as the gate list file with the extension changed to tst.
The parameter file is an optional argument that allows the user to define certain variables that influence the execution of Testgen. If omitted, Testgen will run with default values. The parameter file is a text file with a simple format. One parameter may be specified on each line. The identifier should be entered in upper or lower case. For numeric parameters the identifier should be followed by a value to be assigned to that parameter. The identifier and value should be separated by white space. For logical parameters, simply listing the identifier will change the switch to the opposite of the default state.

Table 4.1 lists the present user assignable parameters along with their default values and valid ranges. The valid ranges simply represent the range of values that are representable by the variable type chosen to store the parameter. Parameters specified by the user are checked by Testgen to be within the appropriate range. These ranges obviously do not reflect the range of realistic values. As a result, it is up to the user to select appropriate values. For example, it would not be to one's benefit to select 0 for a memory block size or a time out limit.

The parameters spathblock, qblock, and btblock are memory block sizes used when requesting memory from the operating system. The parameter spathblock is used when requesting memory for storing the current sensitization path. Functions
requesting memory for queues will do so in block sizes specified by \texttt{qblock}. The parameter \texttt{btblock} is used when requesting memory for backtrack lists. The user may increase these block sizes for larger circuits in order to avoid spending undue time allocating memory blocks and to reduce memory fragmentation.

<table>
<thead>
<tr>
<th>PARAMETER IDENTIFIER</th>
<th>DEFAULT VALUE</th>
<th>VARIABLE TYPE</th>
<th>PARAMETER TYPE</th>
<th>VALID RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>spathblock</td>
<td>50</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>qblock</td>
<td>100</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>btblock</td>
<td>50</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>srchtlimit</td>
<td>5</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>pftlimit</td>
<td>30</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>tottlimit</td>
<td>3600</td>
<td>unsigned int</td>
<td>NUMERIC</td>
<td>0-65535</td>
</tr>
<tr>
<td>printstats</td>
<td>FALSE</td>
<td>short int</td>
<td>LOGICAL</td>
<td>ON/OFF</td>
</tr>
</tbody>
</table>

The time limit parameters are used to limit the amount of time Testgen will spend performing certain operations. These time limits have not been set up to interrupt immediately upon expiration. Instead, the time is simply checked at the completion of an appropriate sub-task, and if the time limit has been reached, the given operation is aborted.

The parameter \texttt{srchtlimit} (search time limit) is the time Testgen is allowed to spend in the final search. The time is checked after the assignment and propagation of each circuit input logic value. The parameter \texttt{pftlimit} (per-fault time limit) is the time Testgen is allowed to spend on each
individual fault. This time limit is checked after the completion of any the major tasks in the function gentest(). The tasks are; path selection and sensitization, marking necessary input lines, assignment of don't care values, assignment of bias values, and final search. The parameter tottlimit (total time limit) is the amount of time Testgen is allowed to work on a given circuit.

The logical parameter printstats is used to enable printing of statistics to the screen during execution. If this switch is turned on, the count of faults tested, undetectable faults, search time outs, per fault time outs, and memory failures will be displayed prior to attempting the next fault. This switch is useful for monitoring the progress of Testgen during a test generation run.

4.3 Interactive Version

The interactive version of Testgen contains the identical test generation routines as the batch version. The difference is the user interface. The interactive version is a menu driven test generator with a menu system that is modeled after the schematic generator SUBGRAPH [9]. Through the menus the user may load gate list files, select individual faults to be targeted for test generation, request information on the current circuit, display or print results, and change test
parameters.

To invoke the interactive version of Testgen, simply type the following command line at the DOS prompt:

```
testgen [filename.par]
```

The optional argument may be used to specify a parameter file. The format and use of this file is identical to that used in the batch version. The available parameters are also the same excluding the `tottlimit` and `printstats` parameters. These parameters have no meaning in the interactive environment. The use of the parameter file has been described in the previous section.

The interactive version of Testgen uses a three window format. The top line of the terminal screen is the title window. This window will be displayed in reverse video and displays the current version of Testgen. The bottom line is used as the status window. This window is also in reverse video and will be used to display the current gate list file and various error and instruction messages. The remaining lines make up the main window. This window is used to display the menus and the outputs requested by the user.

The main menu is shown in Figure 4.1. Through this menu the user will be lead to other menus that will allow the user to complete the operation requested.

The `Load Gate List` option will display all of the gate list files in the current directory. The user may select one
of these files or enter another file name including a path to another directory. After the gate list has been loaded the gate count will be temporarily displayed on the status line. The current gate list file name will then be displayed in the left-most portion of the status line.

<table>
<thead>
<tr>
<th>Main Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit to DOS</td>
</tr>
<tr>
<td>Load Gate List</td>
</tr>
<tr>
<td>Test Generation</td>
</tr>
<tr>
<td>Gate Information</td>
</tr>
<tr>
<td>Edit Parameters</td>
</tr>
</tbody>
</table>

**INTERACTIVE TESTGEN MAIN MENU**

**FIGURE 4.1**

The Test Generation option will allow the user to specify a fault location and type and to invoke the test generator. After the successful generation of a test vector the user will have the option of displaying the results in the main window or writing the results to the printer or to a user definable file. If a file is selected, the user may select to append the results to the file or to overwrite the contents of that file. The append option will allow the user to accumulate test vectors for a given circuit in a single file.

The generated test vectors represent a value to assign to each primary input of the circuit under test. The ordering of these values matches the ordering of the circuit inputs when
listed from the gate information feature.

The Gate Information option allows the user to view the contents of the current gate list. The gate list entries may be sorted by gate type or the entire list may be viewed. As with the test results, the output may be sent to the main window, the printer, or an output file.

The Edit Parameters option allows the user to update the same parameters that may be specified through the .par file. This will allow the user to change these parameters without having to exit, update the parameter file, and reinvoke Testgen.
CHAPTER 5. RESULTS

5.1 Test Results

Performance results have been compiled by running Testgen against several test circuits. The computer system used for these test runs is a 386 20MHz based IBM compatible personal computer. The Dhrystones and Whetstones marks are 4552 and 70.1K respectively. All runs were made using the hard disk drive that has a 28ms average access time and a 122.3Kbyte/second DOS transfer rate. The performance marks cited were calculated by the utility program QAPLUS. The amount of available memory below the 640 Kbyte DOS limit was 592016 bytes. This number was reported by the DOS command mem.

The circuits used for the performance evaluation of Testgen originated from two sources. The first two circuits, ADDER8 AND ADDER32, are 8 and 32 bit full adders. These circuits were generated by the hardware compiler HPCOM from their AHPL circuit descriptions. The AHPL descriptions and the generated gate list files for these circuits will be provided.

The remaining ten circuits are the combinational benchmark circuits from [10]. These circuits were obtained in
a neutral netlist format that is incompatible with Testgen. These circuits were then translated into a gate list format that could be read by Testgen. The program Translat was written to perform this task. Translat will produce a pseudo gate list file that contains all of the portions of a standard gate list file that are needed by Testgen. The original netlist descriptions and the generated pseudo gate list descriptions as well as the source and executable code for Translat will be provided.

<table>
<thead>
<tr>
<th>CIRCUIT NAME</th>
<th>TOTAL GATES</th>
<th>TOTAL LINES</th>
<th>INPUT LINES</th>
<th>OUTPUT LINES</th>
<th>FANOUT STEM</th>
<th>AVG FANOUT</th>
<th>MAX FANOUT</th>
<th>AVG FANIN</th>
<th>MAX FANIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDER8</td>
<td>120</td>
<td>281</td>
<td>17</td>
<td>9</td>
<td>24</td>
<td>6.00</td>
<td>6</td>
<td>2.07</td>
<td>4</td>
</tr>
<tr>
<td>ADDER32</td>
<td>480</td>
<td>1121</td>
<td>65</td>
<td>33</td>
<td>96</td>
<td>6.00</td>
<td>6</td>
<td>2.07</td>
<td>4</td>
</tr>
<tr>
<td>C432</td>
<td>160</td>
<td>472</td>
<td>36</td>
<td>7</td>
<td>89</td>
<td>2.65</td>
<td>9</td>
<td>2.10</td>
<td>4</td>
</tr>
<tr>
<td>C499</td>
<td>202</td>
<td>499</td>
<td>41</td>
<td>32</td>
<td>59</td>
<td>4.34</td>
<td>12</td>
<td>2.02</td>
<td>5</td>
</tr>
<tr>
<td>C880</td>
<td>303</td>
<td>880</td>
<td>60</td>
<td>26</td>
<td>125</td>
<td>3.50</td>
<td>8</td>
<td>1.90</td>
<td>4</td>
</tr>
<tr>
<td>C1355</td>
<td>546</td>
<td>1355</td>
<td>41</td>
<td>32</td>
<td>259</td>
<td>2.97</td>
<td>12</td>
<td>1.95</td>
<td>5</td>
</tr>
<tr>
<td>C1908</td>
<td>880</td>
<td>1908</td>
<td>33</td>
<td>25</td>
<td>385</td>
<td>2.58</td>
<td>16</td>
<td>1.70</td>
<td>8</td>
</tr>
<tr>
<td>(1)C2670</td>
<td>1193</td>
<td>2670</td>
<td>233</td>
<td>140</td>
<td>454</td>
<td>2.74</td>
<td>11</td>
<td>1.74</td>
<td>5</td>
</tr>
<tr>
<td>(2)C2670</td>
<td>1193</td>
<td>2594</td>
<td>157</td>
<td>64</td>
<td>454</td>
<td>2.74</td>
<td>11</td>
<td>1.74</td>
<td>5</td>
</tr>
<tr>
<td>C3540</td>
<td>1669</td>
<td>3540</td>
<td>50</td>
<td>22</td>
<td>579</td>
<td>3.15</td>
<td>16</td>
<td>1.76</td>
<td>8</td>
</tr>
<tr>
<td>C5115</td>
<td>2307</td>
<td>5315</td>
<td>178</td>
<td>123</td>
<td>806</td>
<td>3.51</td>
<td>15</td>
<td>1.90</td>
<td>9</td>
</tr>
<tr>
<td>C6288</td>
<td>2416</td>
<td>6288</td>
<td>32</td>
<td>32</td>
<td>1456</td>
<td>2.64</td>
<td>16</td>
<td>1.99</td>
<td>2</td>
</tr>
<tr>
<td>C7552</td>
<td>2912</td>
<td>7552</td>
<td>207</td>
<td>108</td>
<td>1300</td>
<td>2.95</td>
<td>15</td>
<td>1.75</td>
<td>5</td>
</tr>
</tbody>
</table>

(1) Original C2670 circuit
(2) C2670 circuit obtained for this research

**TEST CIRCUIT STATISTICS**

**TABLE 5.1**

Table 5.1 lists the circuit statistics for the set of test circuits. It must be noted at this time that the circuit C2670 that was obtained for this research does not appear to be the original circuit published in [10]. For circuit C2670, Table 5.1 lists the statistics for the circuit obtained and for the original C2670 circuit. By comparing these
statistics, one may deduce that the only difference between the two versions of the C2670 circuit is that the original c2670 had 76 lines that went straight from the circuit inputs to the circuit outputs. In terms of test generation for single stuck-at faults, this is a trivial difference.

Table 5.2 lists the time limits used during the test runs on each circuit. The memory block parameters were left at their default values for all runs.

<table>
<thead>
<tr>
<th>CIRCUIT</th>
<th>SEARCH TIME LIMIT</th>
<th>PER FAULT TIME LIMIT</th>
<th>TOTAL TIME LIMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDER8</td>
<td>2</td>
<td>5</td>
<td>3600</td>
</tr>
<tr>
<td>ADDER32</td>
<td>2</td>
<td>5</td>
<td>3600</td>
</tr>
<tr>
<td>C432</td>
<td>2</td>
<td>5</td>
<td>3600</td>
</tr>
<tr>
<td>C499</td>
<td>2</td>
<td>5</td>
<td>3600</td>
</tr>
<tr>
<td>C880</td>
<td>2</td>
<td>5</td>
<td>3600</td>
</tr>
<tr>
<td>C1355</td>
<td>4</td>
<td>30</td>
<td>3600</td>
</tr>
<tr>
<td>C1908</td>
<td>4</td>
<td>30</td>
<td>7200</td>
</tr>
<tr>
<td>C2670</td>
<td>4</td>
<td>30</td>
<td>7200</td>
</tr>
<tr>
<td>C3540</td>
<td>5</td>
<td>60</td>
<td>28800</td>
</tr>
<tr>
<td>C5315</td>
<td>5</td>
<td>60</td>
<td>28800</td>
</tr>
<tr>
<td>C6288</td>
<td>5</td>
<td>60</td>
<td>28800</td>
</tr>
<tr>
<td>C7552</td>
<td>5</td>
<td>60</td>
<td>28800</td>
</tr>
</tbody>
</table>

**TEST RUN TIME LIMITS**

**TABLE 5.2**

Table 5.3 lists the results of running Testgen against the set of test circuits. The numbers listed in parenthesis are the percent of total faults for each category.

Since Testgen does not perform any explicit fault reduction, the total number of faults for each circuit will
reduction, the total number of faults for each circuit will simply be twice the total number of lines in that circuit. Fault equivalences are exposed during the determination of additional fault coverage for generated test vectors.

<table>
<thead>
<tr>
<th>CIRCUIT</th>
<th>TOTAL FAULTS</th>
<th>FAULTS TESTED</th>
<th>UNDETECTABLE FAULTS</th>
<th>SEARCH TIME OUTS</th>
<th>PER FAULT TIME OUTS</th>
<th>MEMORY FAILURES</th>
<th>EXEC. TIME</th>
<th>TEST VECTORS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDE8</td>
<td>562</td>
<td>562 (100.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>19</td>
<td>121</td>
</tr>
<tr>
<td>ADDE32</td>
<td>2242</td>
<td>2242 (100.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>198</td>
<td>476</td>
</tr>
<tr>
<td>C432</td>
<td>864</td>
<td>854 (98.84)</td>
<td>0 (0.00)</td>
<td>10 (1.16)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>120</td>
<td>207</td>
</tr>
<tr>
<td>C499</td>
<td>998</td>
<td>987 (98.89)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>11 (1.11)</td>
<td>0 (0.00)</td>
<td>313</td>
<td>266</td>
</tr>
<tr>
<td>C880</td>
<td>1760</td>
<td>1760 (100.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>176</td>
<td>444</td>
</tr>
<tr>
<td>C1355</td>
<td>2710</td>
<td>2688 (99.18)</td>
<td>0 (0.00)</td>
<td>0 (0.00)</td>
<td>22 (0.82)</td>
<td>0 (0.00)</td>
<td>3065</td>
<td>587</td>
</tr>
<tr>
<td>C1908</td>
<td>3816</td>
<td>3569 (93.00)</td>
<td>110 (2.88)</td>
<td>146 (3.82)</td>
<td>11 (0.30)</td>
<td>0 (0.00)</td>
<td>4254</td>
<td>562</td>
</tr>
<tr>
<td>C2670</td>
<td>5188</td>
<td>5021 (96.78)</td>
<td>69 (1.32)</td>
<td>9 (0.17)</td>
<td>89 (1.73)</td>
<td>0 (0.00)</td>
<td>5609</td>
<td>989</td>
</tr>
<tr>
<td>C3540</td>
<td>7080</td>
<td>6103 (86.20)</td>
<td>752 (10.62)</td>
<td>61 (0.86)</td>
<td>162 (2.28)</td>
<td>2 (0.04)</td>
<td>22100</td>
<td>1143</td>
</tr>
<tr>
<td>C3515</td>
<td>10630</td>
<td>10246 (96.38)</td>
<td>89 (0.83)</td>
<td>65 (0.61)</td>
<td>231 (2.18)</td>
<td>0 (0.00)</td>
<td>27138</td>
<td>2274</td>
</tr>
<tr>
<td>C6288</td>
<td>12756</td>
<td>865 (6.87)</td>
<td>16 (0.12)</td>
<td>0 (0.00)</td>
<td>170 (1.35)</td>
<td>664 (5.29)</td>
<td>28829</td>
<td>70</td>
</tr>
<tr>
<td>C7552</td>
<td>15104</td>
<td>13356 (88.42)</td>
<td>58 (0.38)</td>
<td>82 (0.54)</td>
<td>73 (0.48)</td>
<td>1535 (10.18)</td>
<td>25476</td>
<td>2353</td>
</tr>
</tbody>
</table>

**TEST RESULTS**
**TABLE 5.3**

The numbers listed in the FAULTS TESTED column give the fault coverage for that particular circuit. It should be noted that the fault coverage computed here is based upon no fault reduction and cannot be compared directly with the fault
particular time out limit was exceeded. The memory failure column gives the count of faults for which a memory allocation failure occurred. The execution times listed are real time measurements from the time Testgen is invoked to the time of completion. The units are in seconds. The number of test vectors are the total unreduced test vectors generated by Testgen.

All of the listed statistics are computed by Testgen and are displayed on the terminal screen at the end of each run as well as being listed at the end of the output file. Appendix B may be referred to for an example of the output of the batch version of Testgen.

The number of undetectable faults for circuit C3540 was initially found to be suspiciously high. This was particularly true after reviewing the percentage of fault coverage published for other test generation systems run on this same circuit [10][11][12][13][14][15][16]. As a result, each of the 752 faults listed as undetectable was manually verified. After verification of the proper operation of Testgen, it was concluded that the discrepancy in fault coverage was due to the use of fault reduction by the other systems.

Performance on the circuit C6288 was particularly poor. The test run on this circuit was aborted after the eight hour total time limit was reached. During the test run, 1715
faults were considered. Of those faults, 664 (39%) resulted in memory allocation failures. When a memory failure or time out occurs, all of the time spent on the primary fault has been wasted with no results. When a test vector is successfully generated, Testgen may then very quickly find additional faults covered by that test vector. When a high number of memory failures or time outs occur, the performance of Testgen will considerably decline. In fact, it has been observed that reducing a time out limit to the point that the number of time out aborted faults increases, will actually increase the total time for the test generation run. As a result, it may then be concluded that the size and structure of the circuit C6288 is too large to be effectively evaluated by Testgen in the PC environment.

It can been seen from the memory failure column that the upper limit on circuit size set by memory requirements is approximately the size of circuits C6288 and C7552. Although C7552 is larger than C6288, C6288 has greater memory requirements due to its physical characteristics. The circuit C6288 has few inputs and outputs and has a short wide structure. This produces long sensitization paths and large backtrack lists and queues. The circuit C7552 has more circuit inputs and outputs and has a taller and narrower structure. This structure allows the paths, backtrack lists and queues to be shorter.
In general, when compared to other similar systems in the literature [10-16], the batch version of Testgen did not compare as favorably as was hoped. The primary reason for this appears to be the low fault yield from the path oriented procedure used to determine additional fault coverage. Table 5.4 shows the average number of faults listed per generated test vector. Also shown is the percentage of individual faults that were considered by Testgen during the batch runs for each of the test circuits. The term considered faults used here refers to all primary faults for which Testgen explicitly attempted to develop a test.

An efficient ATPG system will list a large number of faults per test vector which will result in a low number of considered faults. Testgen averaged from about 4 to 6 faults per test vector. These numbers are essentially independent of circuit size. These low yields imply that a large percentage of the total faults must be individually considered by Testgen. The data shown in Table 5.4 indicate a serious weakness in the batch version of Testgen. It should be possible to overcome this weakness by adding a fault simulator to the Testgen ATPG system. The simulator would determine all faults detected by the generated test vectors and will significantly increase the fault yield per test vector. This will in turn require that fewer primary faults must be explicitly considered by Testgen.
### ADDITIONAL FAULT COVERAGE STATISTICS

**TABLE 5.4**

<table>
<thead>
<tr>
<th>CIRCUIT</th>
<th>TOTAL FAULTS</th>
<th>TEST VECTORS</th>
<th>AVERAGE FAULTS PER TEST VECTOR</th>
<th>PERCENTAGE CONSIDERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDER8</td>
<td>562</td>
<td>121</td>
<td>4.65</td>
<td>21.53</td>
</tr>
<tr>
<td>ADDER32</td>
<td>2242</td>
<td>476</td>
<td>4.71</td>
<td>21.23</td>
</tr>
<tr>
<td>C432</td>
<td>864</td>
<td>207</td>
<td>4.12</td>
<td>25.40</td>
</tr>
<tr>
<td>C499</td>
<td>998</td>
<td>266</td>
<td>3.71</td>
<td>27.96</td>
</tr>
<tr>
<td>C880</td>
<td>1760</td>
<td>444</td>
<td>3.96</td>
<td>25.22</td>
</tr>
<tr>
<td>C1355</td>
<td>2710</td>
<td>587</td>
<td>4.58</td>
<td>22.65</td>
</tr>
<tr>
<td>C1908</td>
<td>3816</td>
<td>562</td>
<td>6.31</td>
<td>18.84</td>
</tr>
<tr>
<td>C2670</td>
<td>5188</td>
<td>988</td>
<td>5.08</td>
<td>22.24</td>
</tr>
<tr>
<td>C3540</td>
<td>7080</td>
<td>1143</td>
<td>5.34</td>
<td>29.94</td>
</tr>
<tr>
<td>C5315</td>
<td>10630</td>
<td>2274</td>
<td>4.51</td>
<td>25.00</td>
</tr>
<tr>
<td>C6288</td>
<td>12756</td>
<td>INCOMPLETE TEST GENERATION RUN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C7552</td>
<td>15104</td>
<td>2353</td>
<td>5.68</td>
<td>27.15</td>
</tr>
</tbody>
</table>

5.2 Achievements

This research project has produced two versions of a working automatic test pattern generator that is based upon the Wang algorithm. These programs have been developed for the widely accessible personal computer environment.

The interactive version is a one fault at a time test pattern generator with a user friendly menu driven interface.
This version may find practical application in the realm of education.

The original goal for the batch version was that it outperform systems based upon other algorithms. What was not immediately obvious to this author is that efficient ATPG systems include both a test pattern generator and a fault simulator [1][10][11][13][14][16]. It seems unfair to compare a system with a fault simulator to one without. Testgen seems to be fairly competitive when run against the smaller circuits. For larger circuits the weakness of its low yield of faults covered per generated test vector make it impossible to compete with systems employing a fault simulator.

The expectations for the batch version were initially set a bit too high. This lead to some disappointment when first reviewing the results. But in the final analysis, an efficient automatic test pattern generator has been developed. This generator will provide the basis for future improvements and may also be joined with a fault simulator to produce a more competitive ATPG system.
CHAPTER 6. FUTURE RESEARCH

6.1 Future Enhancements

Below is a list and brief description of some possible areas for future research aimed at improving the Testgen algorithm.

Minimal test vector utility: The goal for a practical ATPG is to produce the fewest test vectors with the highest fault coverage possible. Due to the localized approach to test generation used in Testgen, the generated test vectors are only partially specified and may be duplicated. A utility that will combine the test vectors from the output produced by Testgen into a minimal test set would be of significant practical importance.

Improve path selection: Alternate path selection strategies could be implemented and evaluated in terms of the number of back tracks. One possible secondary strategy that could be applied to any primary strategy is one that would force selection of untried paths first. In other words, when selecting a fan out stem output, select on output that has not previously been part of a successful test generation. This should reduce the repeated channeling of faults through the same set of paths. This
should in turn improve the yield for each generated test. This strategy could be employed until the number of backtracks exceeds some limit, at which time the strategy would return to best path first.

**Improve search:** The final search is a significant part of the test generation process and may become an unreasonably large task for some faults. Any improvements made to search or to reduce the search space have the potential to significantly improve the performance of Testgen.

One possible improvement that could be made to search would be to have search start with the circuit inputs that are most likely to cause a conflict. This would push the conflicts higher up in the search tree resulting in a smaller search space. The determination of the inputs that are most likely to cause a conflict is the difficult part of this improvement. A static approach might be possible with the development of a suitable heuristic. A dynamic approach might be to determine all inputs that produced the most recent conflict and to restart search with these inputs at the top of the search tree. This process could be repeated a set number of times or until the search tree ordering converges, at which time search would be allowed to proceed in an exhaustive manner.

**Improve determination of additional fault coverage:** Perhaps
the area with most room for improvement in the Testgen algorithm is in the area of increasing the fault yield per generated test vector. As described in section 2.2.6, the procedure to determine additional faults tested by generated test vectors has trouble dealing with reconvergent fan out. To improve Testgen, the procedure could be enhanced to deal more efficiently this circuit structure.

The changes mentioned above may improve performance somewhat but will not overcome the more general problem encountered when using Testgen as the only component of the ATPG system. The problem is that in order to produce a test vector as quickly as possible, the Testgen algorithm simulates only the portion of the circuit that is necessary to generated the test vector. The result is that without additional simulation only a small set of additional faults may be determined to also be tested by a given test vector. It should be quite advantageous to pair the Testgen algorithm with a fault simulator. After Testgen produces a test vector, the remaining unspecified inputs could be set randomly or by some other means and then passed on to the fault simulator. The remaining circuit could then be simulated to determine all faults tested by the generated test vector.

The marriage of the Testgen test pattern generator
to a fault simulator could be accomplished in one of two ways. First, the process described above could be used at the start of the test generation run when the yield would justify the cost of simulation. Later, when the yield from the fault simulator drops below some threshold, the simulation step could be replaced by the less costly approach currently used by Testgen. The second approach would be to rely solely on the fault simulator and remove the code for the current approach used in Testgen. In either case the use of a fault simulator should have the potential for significant speed improvement for Testgen.

If the first approach is adopted, logic could be added to the function \texttt{results()} to call either \texttt{faultstested()} or a function that would initiate the fault simulator. The new function, in addition to calling the simulator, would have the responsibility of assigning values to any unspecified inputs and to pass the fully specified vector to the simulator in the appropriate format. If the second approach is selected, the call to \texttt{faultstested()} could simply be replaced by the simulator initiating function and the functions \texttt{faultstested()}, \texttt{lookback()}, \texttt{simulate()}, and \texttt{recon()} could be removed from Testgen.

In addition to the above mentioned changes, a fault
table that is accessible to both the simulator and Testgen will also need to be developed. The fault simulator will use this table to mark the appropriate faults as tested and Testgen will use it to pick primary faults to target for test generation. It may also be advantageous to apply a fault reduction routine to the fault table to remove equivalent faults. This will allow direct comparison of fault coverage results with other systems that employ fault reduction. Also, if the fault simulator utilizes fault reduction, which seems likely, it may actually be necessary to add this step.

**Early detection of invalid sensitization values:** During the path sensitization process it is possible to require opposite logic values on a lead line in the good and bad networks. If the lead line is reachable from the primary fault site it might be possible for this requirement to be met. If the lead line is not reachable from the fault site it will be impossible to produce the required sensitization values. In the current implementation of Testgen, this situation will only be detected if both the good and bad network values reach a primary input where it is obviously impossible to have opposite values. If this situation is not detected, the final search will be invoked and will be doomed to fail. An improvement would be to map out the area of influence of the primary fault.
If, during the sensitization process, opposite logic values are required, the line could be checked to determine if it is reachable from the primary fault site. If so, search may succeed and the path selection process may continue. If not, path selection has failed and the path selection procedure should backtrack. This improvement will save Testgen from attempting searches that are assured of failure.

**Remove 640K memory limit:** Testgen is currently limited by DOS to the available memory below the 640K byte boundary. When Testgen runs out of available memory the fault is abandoned. This memory limit will then in turn limit the size of circuits for which Testgen will be useful. In the PC environment this memory limit could be lifted if Testgen is modified to utilize extended memory.

**Port to a mainframe environment:** The source code for the batch version of Testgen has been written and compiled to the ANSI C standard. Porting this version of Testgen to other computer systems should be possible with a minimum of difficulty. Porting Testgen would also be another method of removing the limits of the PC environment.

**Suppress Screen Outputs:** An option could be added to the .par file that would suppress all outputs to the terminal screen. This would allow Testgen to be run in the background on multi-tasking operating systems.
Fault specification: It might be useful to add an option to the \texttt{.par} file to cause the batch version of Testgen to work on a single fault or a list of faults. This would allow the operator to adjust parameters and to allow Testgen to attack the more difficult faults in a circuit.
Appendix A. AHPL LISTING OF EXAMPLE CIRCUIT

MODULE: RECON.
EXINPUTS: A; B; C; D; CLOCK; RESET.
EXOUTPUTS: X; Y; Z.
BODY SEQUENCE: CLOCK.

1 =>(1).
ENDSEQUENCE

CONTROLRESET (RESET)/(1);
X,Y,Z = NRCT1(A; B; C; D).
END.

CLU: NORCKT1(A; B; C; D).
  INPUTS: A; B; C; D.
  OUTPUTS: NORCKT1[3].
  CTERMS: H; C1; C2; C3; C4; C5; C6; C7; C8; C9; C10; C11;
          C12; C13; C14; C15; C16; C17; X; Y; Z.
BODY
  C1 = A + C;
  C2 = B + C;
  C3 = B + D;
  C4 = ^C1;
  H = ^C2;
  C5 = ^C3;
  C6 = B + C4;
  C7 = A + H;
  C8 = D + H;
  C9 = C + C5;
  C10 = ^C6;
  C11 = ^C7;
  C12 = ^C8;
  C13 = ^C9;
  C14 = C10 + C11 + C12 + C13;
  C15 = ^C14;
  C16 = C10 + C15;
  C17 = ^C16;
  X = ^C10;
  Y = C10 & C17;
  Z = C13 + C15;
  NORCKT1 = X,Y,Z.
END.
Appendix B. TESTGEN OUTPUT FOR EXAMPLE CIRCUIT

UNIVERSITY OF ARIZONA             DEPARTMENT OF ELECTRICAL ENGINEERING
WANG ALGORITHM TEST GENERATOR       Saturday October 17, 1992 at 22:08:01

Single stuck-at fault tests for circuit description found in EXCKT.GLF

SYMBOL TABLE:

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ELEM</th>
<th>SYMBOL</th>
<th>ELEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>X</td>
<td>7</td>
<td>Y</td>
<td>8</td>
</tr>
<tr>
<td>Z</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

----- Faults tested:
S-A-0 at output of 1
S-A-0 at input of 42 from 1
S-A-0 at output of 42
S-A-1 at output of 45
S-A-1 at input of 48 from 2
S-A-1 at output of 2
S-A-1 at output of 48
S-A-0 at output of 52
S-A-0 at input of 62 from 52
S-A-1 at output of 62

TEST VECTOR: 100X
OUTPUT ELEM: 7 VALUE: 0/1

----- Faults tested:
S-A-1 at output of 1
S-A-1 at input of 42 from 3
S-A-1 at output of 3
S-A-1 at input of 42 from 1
S-A-1 at output of 42
S-A-0 at output of 45
S-A-0 at output of 48
S-A-1 at output of 52
S-A-1 at input of 52 from 52
S-A-0 at output of 62

TEST VECTOR: 000X
OUTPUT ELEM: 7 VALUE: 1/0

----- Faults tested:
S-A-0 at output of 2
S-A-0 at input of 48 from 2

TEST VECTOR: 111X
OUTPUT ELEM: 7 VALUE: 1/0
Faults tested:
S-A-0 at output of 3
S-A-0 at input of 42 from 3

TEST VECTOR: 001X
OUTPUT ELEM: 7 VALUE: 0/1

Faults tested:
S-A-0 at output of 4
S-A-0 at input of 50 from 4
S-A-1 at output of 50
S-A-1 at input of 58 from 55
S-A-1 at output of 53
S-A-0 at output of 49
S-A-0 at input of 49 from 1
S-A-1 at input of 58 from 52
S-A-1 at output of 58
S-A-0 at output of 59
S-A-0 at input of 64 from 59
S-A-0 at output of 64

TEST VECTOR: 1111
OUTPUT ELEM: 9 VALUE: 1/0

Faults tested:
S-A-1 at output of 4
S-A-1 at input of 50 from 46
S-A-1 at input of 50 from 4
S-A-1 at output of 50
S-A-0 at output of 54
S-A-1 at output of 59
S-A-1 at input of 64 from 59
S-A-1 at input of 64 from 55
S-A-1 at output of 64

TEST VECTOR: 1110
OUTPUT ELEM: 9 VALUE: 0/1

Faults tested:
S-A-0 at output of 43

TEST VECTOR: 0110
OUTPUT ELEM: 9 VALUE: 0/1

Faults tested:
S-A-1 at output of 43
S-A-1 at input of 43 from 3
S-A-1 at input of 43 from 2
S-A-0 at output of 46
S-A-0 at input of 50 from 46
S-A-0 at input of 49 from 46
TEST VECTOR: 0000
OUTPUT ELEM: 9 VALUE: 1/0

-----> Faults tested:
    S-A-0 at output of 44
    S-A-0 at input of 44 from 2
    S-A-1 at output of 47
    S-A-1 at input of 51 from 3
    S-A-1 at output of 51
    S-A-0 at output of 55
    S-A-0 at input of 64 from 55

TEST VECTOR: 0100
OUTPUT ELEM: 9 VALUE: 1/0

-----> Faults tested:
    S-A-1 at output of 44
    S-A-1 at input of 44 from 4
    S-A-1 at input of 44 from 2
    S-A-0 at output of 47
    S-A-0 at output of 51
    S-A-1 at output of 55
    S-A-0 at input of 58 from 52

TEST VECTOR: 1000
OUTPUT ELEM: 9 VALUE: 0/1

-----> Faults tested:
    S-A-1 at output of 46

TEST VECTOR: 0110
OUTPUT ELEM: 9 VALUE: 0/1

-----> Faults tested:
    S-A-1 at output of 49
    S-A-1 at input of 49 from 46
    S-A-1 at input of 49 from 1
    S-A-0 at output of 53

TEST VECTOR: 0111
OUTPUT ELEM: 9 VALUE: 0/1

-----> Faults tested:
    S-A-0 at output of 60
    S-A-0 at input of 60 from 52
    S-A-1 at output of 61
    S-A-1 at output of 63

TEST VECTOR: 1010
OUTPUT ELEM: 8 VALUE: 0/1
Faults tested:
S-A-0 at input of 51 from 3
TEST VECTOR: 0011
OUTPUT ELEM: 9 VALUE: 0/1

Faults tested:
S-A-0 at input of 44 from 4
TEST VECTOR: 1001
OUTPUT ELEM: 9 VALUE: 1/0

Faults tested:
S-A-1 at input of 63 from 52
TEST VECTOR: 0100
OUTPUT ELEM: 8 VALUE: 0/1

THE FOLLOWING FAULTS ARE UNDETECTABLE:
S-A-1 at output of 60
S-A-0 at output of 61
S-A-0 at output of 63
S-A-0 at input of 43 from 2
S-A-0 at input of 43 from 3
S-A-0 at input of 63 from 52
S-A-1 at input of 60 from 52
S-A-0 at input of 58 from 55
S-A-1 at input of 60 from 59
S-A-0 at input of 60 from 59

******* SUMMARY OF TEST GENERATION RESULTS *******

<table>
<thead>
<tr>
<th></th>
<th>TOTAL FAULTS</th>
<th>PERCENTAGE OF TOTAL FAULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tested faults:</td>
<td>80</td>
<td>88.88</td>
</tr>
<tr>
<td>Undetectable faults:</td>
<td>10</td>
<td>11.12</td>
</tr>
<tr>
<td>Total faults:</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Total unreduced test vectors:</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Elapsed time:</td>
<td>2 seconds</td>
<td></td>
</tr>
</tbody>
</table>
TESTGEN.c
  initglobe
  main
    time[2]? 
    panic[3]
      printf[16]?
      exit[15]?
    fopen[2]?
    strcpy?
    strlen?
    strcat?
    initglobe...
  rdpars
    fopen?
    printf[3]?
    getch?
    toupper[2]?
    panic...
    fget[7]?
    sscanf?
    strcmp[7]?
  initread
    printf?
    strncmp[7]?
    fgets[17]?
    panic[16]...
    atoi[7]?
    malloc[5]?
    sscanf[3]?
  listcc
    adddel[3]
      listmem
      malloc?
      panic[2]...
      realloc?
      listmem...
  rdfile
    rewind?
    strncmp[7]?
    fgets[17]?
    atoi[6]?
    malloc[4]?
    panic[6]...
    strcpy?
  strip
    strcpy[2]?
    strlen?
gevalgate
gevalgate...
bbtrace
realloc[5]?
binconsistent
bevalgate
bevalgate...
free?
tcupdate[12]
selpath
free?
dellist
addlist[2]...
sensgate
malloc[4]?
realloc[3]?
free[6]?
btccont[2]...
difftime[5]?
markinputs
malloc?
free[3]?
srchcleanup1[2]
realloc?
assX1
assX2
malloc?
free[23]?
srchcleanup1[22]...
realloc[11]?
assbias
malloc[2]?
free[30]?
srchcleanup1[14]...
realloc[7]?
ssearch
time[2]?
malloc[2]?
free?
srchassX[4]
malloc?
free[21]?
realloc[10]?
srchcleanup1[9]...
srchcleanup2[9]
free[2]?
realloc[2]?
difftime?
results
panic...
fprintf[9]?
fprintf[9]?
faultstested
### Appendix D. Reference List for Batch Version of Testgen

<table>
<thead>
<tr>
<th>Function</th>
<th>Called by List</th>
</tr>
</thead>
<tbody>
<tr>
<td>_atold:</td>
<td></td>
</tr>
<tr>
<td>_bcalloc:</td>
<td></td>
</tr>
<tr>
<td>_bexpand:</td>
<td></td>
</tr>
<tr>
<td>_bfree:</td>
<td></td>
</tr>
<tr>
<td>_bfreeseg:</td>
<td></td>
</tr>
<tr>
<td>_bheapadd:</td>
<td></td>
</tr>
<tr>
<td>_bheapchk:</td>
<td></td>
</tr>
<tr>
<td>_bheapmin:</td>
<td></td>
</tr>
<tr>
<td>_bheapseg:</td>
<td></td>
</tr>
<tr>
<td>_bheapset:</td>
<td></td>
</tr>
<tr>
<td>_bheapwalk:</td>
<td></td>
</tr>
<tr>
<td>_bmalloc:</td>
<td></td>
</tr>
<tr>
<td>_bmsize:</td>
<td></td>
</tr>
<tr>
<td>_brealloc:</td>
<td></td>
</tr>
<tr>
<td>_exit:</td>
<td></td>
</tr>
<tr>
<td>_expand:</td>
<td></td>
</tr>
<tr>
<td>_falloc:</td>
<td></td>
</tr>
<tr>
<td>_fexpand:</td>
<td></td>
</tr>
<tr>
<td>_ffree:</td>
<td></td>
</tr>
<tr>
<td>_fheapchk:</td>
<td></td>
</tr>
<tr>
<td>_fheapmin:</td>
<td></td>
</tr>
<tr>
<td>_fheapset:</td>
<td></td>
</tr>
<tr>
<td>_fheapwalk:</td>
<td></td>
</tr>
<tr>
<td>_filbuf:</td>
<td></td>
</tr>
<tr>
<td>_flsbuf:</td>
<td></td>
</tr>
<tr>
<td>_fmalloc:</td>
<td></td>
</tr>
<tr>
<td>_fmemccpy:</td>
<td></td>
</tr>
<tr>
<td>_fmemchr:</td>
<td></td>
</tr>
<tr>
<td>_fmemcmp:</td>
<td></td>
</tr>
<tr>
<td>_fmemcpy:</td>
<td></td>
</tr>
<tr>
<td>_fmemmove:</td>
<td></td>
</tr>
<tr>
<td>_fmemset:</td>
<td></td>
</tr>
<tr>
<td>_fmsize:</td>
<td></td>
</tr>
<tr>
<td>_frealloc:</td>
<td></td>
</tr>
<tr>
<td>_freect:</td>
<td></td>
</tr>
<tr>
<td>_fsopen:</td>
<td></td>
</tr>
<tr>
<td>_fstrcat:</td>
<td></td>
</tr>
<tr>
<td>_fstrchr:</td>
<td></td>
</tr>
<tr>
<td>_fstrcmp:</td>
<td></td>
</tr>
<tr>
<td>_fstrcpyp:</td>
<td></td>
</tr>
<tr>
<td>_fstrcspn:</td>
<td></td>
</tr>
<tr>
<td>_fstrdup:</td>
<td></td>
</tr>
<tr>
<td>_fstricmp:</td>
<td></td>
</tr>
<tr>
<td>_fstrlen:</td>
<td></td>
</tr>
<tr>
<td>_fstrlwr:</td>
<td></td>
</tr>
</tbody>
</table>
_fstrncat:
_fstrncmp:
_fstrncpy:
_fstrnicmp:
_fstrncpy:
_fstrnset:
_fstrpbrk:
_fstrrev:
_fstrset:
_fstrspn:
_fstrstr:
_fstrtok:
_fstrupr:
_fullpath:
_heapadd:
_heapchk:
_heapmin:
_heapset:
_heapwalk:
_irotl:
_irotr:
_makepath:
_memavl:
_memmax:
_msize:
_ncalloc:
_nexpand:
_nfree:
_nheapchk:
_nheapmin:
_nheapset:
_nheapwalk:
_nmalloc:
_nmsize:
_nrealloc:
_nstrdup:
_pclose:
_popen:
_rotl:
_rotr:
_searchenv:
_splitpath:
_strdate:
_strerror:
_strftime:
_strtold:
_tolower:
_toupper:
abort:
abs:
adddel:
addlist:
buildckt[3]
listcc[3]
gentest
heapadd
heapchk
heapmin
heapset
heapwalk
_heapmin
_heapset
_heapwalk
_nheapadd
_nheapchk
_nheapmin
_nheapset
_nheapwalk
_rotl
_rotr
_searchenv
_splitpath
_strdate
_strerror
strftime
_strtold
_tolower
_toupper
abort:
abs:
adddel:
addlist:
buildckt[3]
listcc[3]
gentest
heapadd
heapchk
heapmin
heapset
heapwalk
_heapmin
_heapset
_heapwalk
_nheapadd
_nheapchk
_nheapmin
_nheapset
_nheapwalk
_rotl
_rotr
_searchenv
_splitpath
_strdate
_strerror
strftime
_strtold
_tolower
_toupper
abort:
abs:
adddel:
addlist:
buildckt[3]
listcc[3]
gentest
heapadd
heapchk
heapmin
heapset
heapwalk
_heapmin
_heapset
_heapwalk
_nheapadd
_nheapchk
_nheapmin
_nheapset
_nheapwalk
_rotl
_rotr
_searchenv
_splitpath
_strdate
_strerror
strftime
_strtold
_tolower
_toupper
abort:
abs:
adddel:
addlist:
buildckt[3]
listcc[3]
gentest
heapadd
heapchk
heapmin
heapset
heapwalk
_heapmin
_heapset
_heapwalk
_nheapadd
_nheapchk
_nheapmin
_nheapset
_nheapwalk
_rotl
_rotr
_searchenv
_splitpath
_strdate
_strerror
strftime
_strtold
_tolower
_toupper
abort:
abs:
adddel:
addlist:
buildckt[3]
listcc[3]
gentest
heapadd
heapchk
heapmin
heapset
heapwalk
_heapmin
_heapset
_heapwalk
_nheapadd
_nheapchk
_nheapmin
_nheapset
_nheapwalk
_rotl
_rotr
_searchenv
_splitpath
_strdate
_strerror
strftime
_strtold
_tolower
_toupper
abort:
abs:
alloca:
asctime:
assbias:
assX1:
assX2:
atexit:
atof:
atoi:
atol:
bbtrace:
bevalgate:
binconsist:
bidiolist:
bsearch:
btcont:
buildckt:
bbuildsym:
calloc:
cgets:
clearerr:
clock:
cprintf:
cputs:
cscanf:
cftime:
dellist:
difftime:
div:
ecvt:
exit:
exitefault:
faultstested:
fclose:
fcloseall:
fcvt:
fdopen:
feof:
ferror:
fflush:
fgetc:
fgetchar:
fgetpos:
fgets:
fileno:
flushall:
fopen:
fprintf:
putc:
putchar:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fputs</code></td>
<td></td>
</tr>
<tr>
<td><code>fread</code></td>
<td></td>
</tr>
<tr>
<td><code>free</code></td>
<td><code>rdfile[2]</code></td>
</tr>
<tr>
<td></td>
<td><code>buildckt[3]</code></td>
</tr>
<tr>
<td><code>freepath</code></td>
<td><code>orderfos[4]</code></td>
</tr>
<tr>
<td></td>
<td><code>exitefault</code></td>
</tr>
<tr>
<td><code>fseek</code></td>
<td><code>markinputs[3]</code></td>
</tr>
<tr>
<td><code>fsetpos</code></td>
<td><code>selpath</code></td>
</tr>
<tr>
<td><code>ftell</code></td>
<td><code>assbias[30]</code></td>
</tr>
<tr>
<td><code>fwrite</code></td>
<td><code>srchcleanup2[2]</code></td>
</tr>
<tr>
<td><code>freepath[2]</code></td>
<td><code>gentest</code></td>
</tr>
<tr>
<td></td>
<td><code>simulate[9]</code></td>
</tr>
<tr>
<td><code>fscanf</code></td>
<td></td>
</tr>
<tr>
<td><code>gbtrace</code></td>
<td><code>btcont</code></td>
</tr>
<tr>
<td><code>gcvt</code></td>
<td></td>
</tr>
<tr>
<td><code>gentest</code></td>
<td><code>main</code></td>
</tr>
<tr>
<td><code>getc</code></td>
<td></td>
</tr>
<tr>
<td><code>getch</code></td>
<td><code>rdpars</code></td>
</tr>
<tr>
<td><code>getchar</code></td>
<td></td>
</tr>
<tr>
<td><code>getche</code></td>
<td></td>
</tr>
<tr>
<td><code>getenv</code></td>
<td></td>
</tr>
<tr>
<td><code>gets</code></td>
<td></td>
</tr>
<tr>
<td><code>getw</code></td>
<td></td>
</tr>
<tr>
<td><code>gevalgate</code></td>
<td><code>gbtrace</code></td>
</tr>
<tr>
<td><code>ginconsist</code></td>
<td><code>ginconsist</code></td>
</tr>
<tr>
<td><code>gmtime</code></td>
<td></td>
</tr>
<tr>
<td><code>halloc</code></td>
<td><code>main</code></td>
</tr>
<tr>
<td><code>hfree</code></td>
<td><code>main</code></td>
</tr>
<tr>
<td><code>initglobe</code></td>
<td><code>main</code></td>
</tr>
<tr>
<td><code>initread</code></td>
<td><code>main</code></td>
</tr>
<tr>
<td><code>inp</code></td>
<td></td>
</tr>
<tr>
<td><code>inpw</code></td>
<td></td>
</tr>
<tr>
<td><code>isalnum</code></td>
<td></td>
</tr>
<tr>
<td><code>isalpha</code></td>
<td></td>
</tr>
<tr>
<td><code>isascii</code></td>
<td></td>
</tr>
<tr>
<td><code>iscntrl</code></td>
<td></td>
</tr>
<tr>
<td><code>iscsym</code></td>
<td></td>
</tr>
<tr>
<td><code>iscsymf</code></td>
<td></td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td></td>
</tr>
<tr>
<td><code>isgraph</code></td>
<td></td>
</tr>
<tr>
<td><code>islower</code></td>
<td></td>
</tr>
<tr>
<td><code>isprint</code></td>
<td></td>
</tr>
<tr>
<td><code>ispunct</code></td>
<td></td>
</tr>
<tr>
<td><code>isspace</code></td>
<td></td>
</tr>
<tr>
<td><code>isupper</code></td>
<td></td>
</tr>
<tr>
<td><code>isxdigit</code></td>
<td></td>
</tr>
<tr>
<td><code>itoa</code></td>
<td></td>
</tr>
</tbody>
</table>
kbhit:
labs:
ldiv:
listcc:
listfault:
listmem:
localtime:
lookback:
ltoa:
main:
malloc:
markinputs:
memccpy:
memchr:
memcmp:
memcpy:
memmove:
memset:
mktime:
movedata:
normalize:
onexit:
orderfos:
outp:
outpw:
panic:
 perror:
 printf:
putc:
 putch:
putchar:
putenv:
puts:
putw:
qsort:
rand:
rdfile:
rdpars:
realloc:
recon:
remove:
rename:
results:
rewind:
rmtmp:
scanf:
search:
selpath:
sensgate:
setbuf:
setvbuf:
simulate:
sprintf:
strcat:
strchr:
strcspn:
strdup:
strerror:
strftime:
strcmp:
strcmpi:
strcoll:
strcspn:
strcat:
strlen:
strlwr:
strncat:
strncpy: rdfile
strnicmp: 
strnset: 
strpbrk: 
strrchr: 
strrev: 
strstr: 
strset: 
strspn: 
strstr: 
strset: 
strspn: 
strspn: 
strset: 
strspn: 
strset: 
strspn: 
summary: main
swab: 
system: 
tempnam: 
tempnam: 
tempnam: 
tempnam: 
tempnam: 
time: main[2] header
summary search[2]
tmpfile: gentest gentest
tmpnam: gentest
ungetc:
toupper: header rdpars[2]
tzset: 
ultoa:
ungetc:
ungetch:
unlink:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
vfprintf:
unmark:
usp

VARIABLE USED BY LIST
__________
________

__amblksiz: strip[2]
__ctype: 
__doserrno: 
__fileinfo: 
__fmode: 
__iob: 
__osmajor: 
__osminor: 
__osmode: 
__psp: 

a:  
action:  
addlist:  
addnum:  
allX:  
argc:  
argv:  

b:  
bad:  
ballmatch:  
bdetermined:  
biasval:  
bind:  
bsmatch:  
bt:  
bttblock:  
bttfront:  
btlistsiz:  
bval:  
cktelem1:  
cktelem2:  
cldindex:  
conflict:  
control_ckt:  
count:  
curbias:  
curbt:  
curbtfront:  
curcolnum:  
curdist:  
curelem1:  
curelem2:  

panic[2]  
search[5]  
adddel[3]  
adddel[2]  
assX2[5]  
main[3]  
main[6]  
panic[6]  
search[13]  
assbias[8]  
search[8]  
initglobe[3]  
assbias[5]  
assbias[6]  
sensgate[8]  
search[5]  
assbias[7]  
search[17]  
gbtrace[4]  
bbtrace[4]  
initglobe[5]  
gbtrace[2]  
initglobe[5]  
bbtrace[2]  
initglobe[5]  
gbtrace[2]  
initglobe[5]  
bbtrace  
rdpars[3]  
search[3]  
(btlistsiz)  
srchassX[7]  
search[14]  
search[14]  
search[15]  
selpath[9]  
sensgate[16]  
selpath[11]  
sensgate[27]  
lookback[5]  
search[13]  
rdfile[12]  
listmem[4]  
assbias  
sensgate[10]  
assbias[9]  
srchassX[33]  
srchassX[45]  
srchassX[5]  
search[5]  
search[12]  
header[7]  
orderfos[15]  
main[5]  
initread[8]  
rdfile[14]  
normalize[7]  
listcc[10]  
summary[10]  
orderfos[24]  
markinputs[7]  
assX2[18]  
assbias[10]  
search[57]  
srchcleanup[1]  
srchcleanup[2]  
srchassX[16]  
simulate[10]  
recon[16]  
main[5]  
initread[5]  
rdfile[7]  
listcc[3]
summary[17]       orderfos[28]
recon[8]          
listcc[3]         
unmark[3]         
unmark[3]         
curfosnum:        selpath[3]    
curilindex:       srchassX[14]   
curilinel:        search[8]      srchassX[16]
simulate[6]       
simulate[3]       
curlindex:        srchassX[2]    
curline1:         summary[21]    selpath[18]
assX2[13]         search[53]
recon[9]          
freepath[10]      
curmeas:          orderfos[11]
curmindelta:      orderfos[5]
curminindex:      orderfos[7]
curnext:          sensgate[12]
assX2[10]         assbias[27]
simulate[21]      
curX:             srchassX[5]
d:                 header[3]
daylight:         
dellist:          main[3]
delnum: adddel[6]
depends: recon[6]
dummy: initread listcc[3]
elapsedtime: summary[4]
elem: listmem[2]
environ: 
errno: 
failed: search[3]
sensgate gbtrace[2]
srchassX[4] faultstested
faultline1: faultstested[40] lookback[41]
fname: header[4]
fos: buildckt[24]
fosnum: buildckt[3]
srchassX[19] faultstested
lookback[3] recon
gallmatch: assbias[8]
gb: sensgate[14] assX2[29]
srchassX[19]
gdesc: buildsym[5]
gdetermined: search[8]
gind: assbias[7]
glt2:
summary[21] buildckt[29]
bldiolist[5] orderfos[18]
gbtrace[5] gevalgate
bbtrace[5] bevalgate

summary[5] buildckt[23]
gevalgate bbtrace[3]
bevalgate ginconsist[2]

main[8]
initread[12] rdfile[19]
buildckt[36] bldiolist[9]
orderfos[32] selpath[9]
sensgate[29] gbtrace[33]
gevalgate[3] bbtrace[33]

i:
initread[12] rdfile[19]
buildckt[36] bldiolist[9]
orderfos[32] selpath[9]
sensgate[29] gbtrace[33]
gevalgate[3] bbtrace[33]
assbias[14] search[19]
srchassX umark[6]
lookback[23] simulate[7]
recon[3]

i0: lookback[3]
i1: lookback[3]
icnt: bldiolist[5]
idiot: initread[7]
iline1: recon[3]
imbecile: rdfile[4]
        buildckt[34] gbtrace[8]
        assX2[34] assbias[26]
        recon[7]
infp: main[4]
initial_attempt: selpath[3]
        bldiolist[6] orderfos
inval: initglobe[3]
inbviasval: assbias[9]
invertgate: assbias[9]
buildckt[8] orderfos[29]
selpath markinputs[7]
assX1[3] assX2[38]
assbias[35] srchassX[29]
simulate[23] recon[23]
bldcblck orderfos[9]
assX2[31] assbias

srchassX[35]


buildckt
line: addlist[2] buildckt
linel: initread[29] tcupdate[9]
list:
        simulate[2]
        simulate[2]
umfound: buildckt[18]
umg0: search[11]
umgx: search[5]
        simulate[2]
            orderfos[3]
origbt: search[32]
origelem1: assbias[3]
origelem2: assbias[4]
origin: search[37]
origindex: assX[2][4]
origline2: search[27] recon[5]
orignumlines: assX2[54] assbias[16]
outfile: main[6]
summary[76]
parfile: rdpars[4]
pathlineindex: sensgate[4]
summary[3]
pftlimit: rdpars[3]
gentest[6]

pmfail1: summary[5]

pmfail2: summary[5]

ppftol1: summary[5]

ppftol2: summary[5]

faulttested lookback[22]


srchassX

selpath[5]

prevline1: selpath[7]
sensgate[12]

prevline2: sensgate[10]

prevmin: orderfos[6]

prevnumdel: listcc[3]


preval: search[9]


psrchto1: summary[6]

psrchto2: summary[6]

ptested1: summary[8]

ptested2: summary[8]

pundet1: summary[7]

pundet2: summary[7]

q: btcont[5] gbtrace[27]

q: orderfos[29] exitefault[8]


assX2[55] srcassX[45]
lookback[38]  simulate[28]
recon[22]      
q1:  assbias[27]
q1end: assbias[14]
q1size: assbias[13]
q2:  assbias[31]
q2end: assbias[19]
q2size: assbias[17]
         (qsize)[9]  exitefault
         (q1size)  srchassX[5]
qgend: orderfos[27]  exitefault[2]
assX2[26]  srchassX[16]
lookback[17]  simulate[19]
qgend: recon[6]
qfront: orderfos[22]  exitefault[2]
qgend: recon[12]
assX2[18]  srchassX[18]
lookback[26]  simulate[21]
qgend: recon[16]
faultstested[5]

s:  strip[4]
salor0:  listfault[6]  summary
senseval: sensgate[10]
sensitized: selpath[7]
tottestvectors:  initglobe[2]  results
tp:  header[3]
tstructptr:  header[3]
tzname:
val:
value:
xorind:
xorsecond:

MACRO

-----

USED BY LIST

----

_DATE__:
__FILE__:
__LINE__:
__STDC__:
__TIME__:
__TIMESTAMP__:
BLANK:
_CLOCK_T_DEFINED:
_CONTROL:
CTYPE_DEFINED:
_DIGIT:
_DIV_T_DEFINED:
_FAR_:
_FILE_DEFINED:
FILES_INCLUDED:
_FPOS_T_DEFINED:
FREEENTRY:
FUNCTIONS_DECLARED:
_HEAP_MAXREQ:
_HEAPBADBEGIN:
_HEAPBADNODE:
_HEAPBADPTR:
_HEAPEMPTY:
_HEAPEND:
_HEAPINFO_DEFINED:
_HEAPPOK:
_HEX:
_EOF:
_EOFERR:

results
_IOFBF: header
_IOLBF:
_IOMYBUF:
_IONBF:
_IOREAD:
_IORW:
_IOSTRG:
_IOWRT:
_LOADDS_: LOWER:
_MAX_DIR:
_MAX_DRIVE:
_MAX_EXT:
_MAX_FNAME:
_MAX_PATH:
_MSC_VER:
_NFILE:
_NULLOFF:
_NULLSEG:
_ONEXIT_T_DEFINED:
_PUNCT:
_SIZE_T_DEFINED:
_SPACE: strip[2]
_STDIO_DEFINED:
_TIME_T_DEFINED:
_TM_DEFINED:
_tolower:
_toupper:
_UPPER:
_USEDENTRY:
_VA_LIST_DEFINED:
ALLOCFAIL:

AND:
ALLOCFAIL:
 BADFIELD:
 BADFORMAT:
 BADGATE:
 BUFSIZE:
 BVAL:

CHAR_BIT:
 CHAR_MAX:
GB: selpath
   btcont[2]
   assbias[9]
   srchassX[6]
sensgate[11]
   assX2[7]
   search[9]
   freepath
getc:
getchar:
GVAL:
   exitefault
   sensgate[9]
   gbtrace[6]
   assbias[6]
   srchassX[9]
   freepath[2]
   main
   panic[2]
   bldiolist[2]
   exitefault
   gbtrace[3]
   markinputs
   srchassX[2]
INOPENERR:
INPUT:
   normalize
   orderfos[2]
   sensgate[2]
   gbtrace[3]
   btraceX[3]
   assbias[2]
   main
   panic[2]
   bldiolist[2]
   exitefault
   gbtrace[3]
   markinputs
   srchassX[2]
INT_MAX:
INT_MIN:
isalnum:
isalpha:
isascii:
iscntrl:
iscsym:
iscsymf:
isdigit:
isgraph:
islower:
isprint:
ispunct:
isspace:
strip[2]
isupper:
isxdigit:
L_tmpnam:
LINECOUNT:
rdfile
panic[2]
LONG_MAX:
LONG_MIN:
M_I286:
M_I86:
M_I86LM:
max:
MAXFANIN:
iniread
buildckt
MAXFANOUT:
iniread
MAXGATES:
iniread[2]
MAXIMUM:
iniread[2]
orderfos
MAXLINE:
strip
initread[18]
rdfile[18]
rdfile
rdpars[2]
MAXLINES:
rdfile
MAXSYM:
iniread
MB_LEN_MAX:
iniread[3]
rdfile[5]
MEMERR:
null:
results buildckt[4]
bldiolist orderfos[17]
assX2[23] assbias[16]
search[14] srcchassX[21]

min:
MINIMUM:
orderfos[2]
MSDOS:
normalize sensgate[2]
NAND:
bbtrace bevalgate[2]
bbtrace bevalgate[2]
ginconsist binconsist
NCH:
initglobe[10] ginconsist
binconsist search[4]
NO_EXT_KEYS:
NOEXTKEYS:
normalize sensgate[2]
NOR:
bbtrace bevalgate[2]
bbtrace bevalgate[2]
ginconsist binconsist
NOVAL:
initglobe[2] listfault
(val)[2] gevalgate
bbtrace[7] bevalgate
assbias[18] search[18]
NULL:
rdfile[9] adddel
results header
(fp) rdpars[4]
simulate[10]
buildckt[7]
bldiolist[2]
orderfos[10]
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>simulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMP_MAX:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tooascii:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTTO:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>results[2]</td>
<td>listfault</td>
</tr>
<tr>
<td></td>
<td>rdpar[8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>buildckt[2]</td>
<td></td>
</tr>
</tbody>
</table>
|                     | gentest[9]   | (sensitized)
|                     | (initial attempt) |         |
|                     | gbtrace[4]   | gevalgate  |
|                     | bbtrace[4]   | bevalgate  |
|                     | markinputs[3]| asX2[14]   |
|                     | srchassX[12] | faulttested|
|                     | recon[9]     |            |
| UINT_MAX:           | selpath[3]   |            |
| UNDET:              | tcupdate[3]  |            |
|                     | bbtrace[2]   |            |
|                     | assX2[14]    | search[27] |
|                     | srchassX[18] | unmark[2]  |
|                     | faulttested[4]|        |
|                     | simulate[4]  |            |
|                     | gevalgate[2] |            |
|                     | bevalgate[2] | gin inconsist |
|                     | binconsistent | assbias[2] |
|                     | recon[9]     |            |

**TYPE**

---

**USED BY LIST**

_div_t:
_JHEAPINFO:
_heapinfo:
_iobuf:
_ldiv_t:
bqelem:
  btcont  gbtrace[10]
btqelem:
btrack:
  exitefault  sensgate[8]
  assX2[27]  assbias[9]
btrack:
clock_t:
div_t:
FILE:
  main[2]  results
  header  listfault
  rdpars  summary
fpos_t:
gdesc1:
  rdfile[3]  normalize
  listcc[2]  results
  header  listfault[2]
  summary[2]  buildckt
  bldiolist  orderfos[2]
  gentest  exitefault
  gbtrace[2]  gevalgate
  bbtrace[2]  bevalgate
  unmark[2]  freepath
gdesc1:
gdesc2:
  results  listfault[2]
  summary[2]  buildckt
  gevalgate  bbtrace[2]
  bevalgate  ginconsistent[2]
  unmark[2]  freepath
gdesc2:
gdesc3:
gdesc3:
inputelem: results bldiolist[2]
orderfosl search[2]
unmark faulttested

inputelem: ldesc1:
main
results rdfile
summary[2] listfault
orderfosl gentest
ginconsist binconsist
markinputs assX1[2]

ldesc1:
ldesc2:
main
buildckt[2] exitefault

ldesc2:
ldiv_t:
measelem:
onexit_t:
size_t:

main
initread[2] initglobe
normalize adddel[2]
listcc[2] results
header summary
buildckt bldiolist
orderfosl addlist
exitfault sensgate
cont gbtrace[2]
assX1 assX2
assbias search[2]
srchcleanup1 srchassX[2]
unmark freepath
faulttested lookback
simulate[2] recon

symbol:
symdesc:
time_t:

main
summary rdpars
search gentest[2]

header
va_list:
REFERENCES


