INFORMATION TO USERS

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

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

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

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

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

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

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

University Microfilms
International
300 N. Zeeb Road
Ann Arbor, MI 48106
PATEL, MAYANK RAMAN

HARDWARE COMPILER DRIVEN HEURISTIC SEARCH FOR DIGITAL IC TEST SEQUENCES

THE UNIVERSITY OF ARIZONA

M.S. 1985
HARDWARE COMPILER DRIVEN HEURISTIC SEARCH
FOR DIGITAL IC TEST SEQUENCES

by

Mayank Raman Patel

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

1985
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 the rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be 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 interest 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] 4 March 1985
FREDERICK J. HILL
Professor of Electrical and Computer Engineering
ACKNOWLEDGMENTS

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

Special thanks are due A. Mohsseni for his effort and valuable contribution, and G. R. Zou, Z. Wang, and A. Eltayeb for their assistance and helpful comments.

Indeed, I am indebted to my wife and my parents for their constant support and encouragement.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 SCIRTSS</td>
<td>1</td>
</tr>
<tr>
<td>1.2 LSSD and Other Test Generation Systems</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Review of Previous Work on SCIRTSS</td>
<td>5</td>
</tr>
<tr>
<td>2. SCIRTSS OVERALL DESCRIPTION</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Executive</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Subroutine BRIDGE</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Subroutine FLTSIM</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Subroutine NDALG</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Subroutine SEARCH</td>
<td>10</td>
</tr>
<tr>
<td>2.6 AHPL SCIRTSS Interface</td>
<td>12</td>
</tr>
<tr>
<td>3. SCIRTSS CONTROL PROGRAM</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Interfaces</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1 BRIDGE and CONTROL PROGRAM</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2 FLTSIM and BRIDGE</td>
<td>16</td>
</tr>
<tr>
<td>3.1.3 NDALG and BRIDGE</td>
<td>20</td>
</tr>
<tr>
<td>3.1.4 SEARCH and BRIDGE</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Functional Flow</td>
<td>24</td>
</tr>
<tr>
<td>4. BRIDGE</td>
<td>26</td>
</tr>
<tr>
<td>4.1 Fault Categories</td>
<td>26</td>
</tr>
<tr>
<td>4.2 Fault Priority Classes</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Fault Selection</td>
<td>29</td>
</tr>
<tr>
<td>4.4 Functional Flow</td>
<td>32</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>PAGE</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>5. DATA STRUCTURES</td>
<td>34</td>
</tr>
<tr>
<td>5.1 NRY</td>
<td>34</td>
</tr>
<tr>
<td>5.2 BRIDGE</td>
<td>38</td>
</tr>
<tr>
<td>5.3 NDALG</td>
<td>42</td>
</tr>
<tr>
<td>5.4 SEARCH</td>
<td>44</td>
</tr>
<tr>
<td>6. EXTENDING PREVIOUS WORK</td>
<td>49</td>
</tr>
<tr>
<td>7. SCIRTSS USER INTERFACE</td>
<td>53</td>
</tr>
<tr>
<td>7.1 General Parameters</td>
<td>53</td>
</tr>
<tr>
<td>7.2 Second Level Parameters</td>
<td>54</td>
</tr>
<tr>
<td>7.3 SEARCH Guiding Input Vectors</td>
<td>55</td>
</tr>
<tr>
<td>7.4 Interactive Interface</td>
<td>56</td>
</tr>
<tr>
<td>7.5 User Defined Heuristic Component</td>
<td>56</td>
</tr>
<tr>
<td>7.6 Design Rules in SCIRTSS</td>
<td>58</td>
</tr>
<tr>
<td>7.6.1 U-AHPL Control Functions</td>
<td>58</td>
</tr>
<tr>
<td>7.6.2 Bidirectional Bus Considerations</td>
<td>59</td>
</tr>
<tr>
<td>7.6.3 Circuit Size</td>
<td>59</td>
</tr>
<tr>
<td>8. EXAMPLES</td>
<td>61</td>
</tr>
<tr>
<td>8.1 BYTADDR Circuit</td>
<td>61</td>
</tr>
<tr>
<td>8.2 DUPCHK Circuit</td>
<td>64</td>
</tr>
<tr>
<td>8.3 MDPCHK Circuit</td>
<td>67</td>
</tr>
<tr>
<td>9. CONCLUSION</td>
<td>71</td>
</tr>
<tr>
<td>APPENDIX A: SCIRTSS PROGRAMS</td>
<td>73</td>
</tr>
<tr>
<td>A-1 SCIRTSS Hierarchical Structure</td>
<td>74</td>
</tr>
<tr>
<td>A-2 SCIRTSS COMMON Blocks Files</td>
<td>78</td>
</tr>
<tr>
<td>A-3 Listing of Defines</td>
<td>79</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>83</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SCIRSS Flow Chart</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Compiler Driven SCIRSS</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>SCIRSS Functional Flow</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Fault Priority Classification</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>BRIDGE Functional Flow</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Array NRY(1:NFO)</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Array NRY(NVECT, MVPC)</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Array NRY(LNKO:NRYDIM)</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>SEARCH Node Structure</td>
<td>46</td>
</tr>
<tr>
<td>8.1</td>
<td>BYTADDR U-AHPL Description</td>
<td>62</td>
</tr>
<tr>
<td>8.2</td>
<td>BYTADDR CLUS Description</td>
<td>63</td>
</tr>
<tr>
<td>8.3</td>
<td>DUPCHK U-AHPL Description</td>
<td>66</td>
</tr>
<tr>
<td>8.4</td>
<td>MDPCHK U-AHPL Description</td>
<td>68</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Extensions to Previous SCIRTSS</td>
<td>52</td>
</tr>
<tr>
<td>8.1 Summary of SCIRTSS Statistics</td>
<td>64</td>
</tr>
<tr>
<td>8.2 Summary of SCIRTSS Results</td>
<td>65</td>
</tr>
<tr>
<td>8.3 MDPCHK Circuit Results</td>
<td>69</td>
</tr>
<tr>
<td>8.4 New and Old SCIRTSS Runs on MDPCHK</td>
<td>70</td>
</tr>
</tbody>
</table>
ABSTRACT

This thesis describes an automatic test generation system, SCIRTSS Version 4.0 (Sequential Circuit Test System). The test generation system is driven by the hardware compiler AHPL, a Hardware Programming Language, and an intelligent heuristic-based Search for test vector generation.

The major objective of this research was to develop a fault selection process and the control program for the SCIRTSS system. The control program controls the parallel fault simulator, the test sequence search program, and the fault selection program. The fault selection program selects faults for propagation or sensitization based on a fault classification system and converts the results of the parallel fault simulator to the format required by the search program.

This system is compared with its predecessor, SCIRTSS Version 3.0. This system is more user friendly (highly interactive) and portable than the old version. Several circuits were tested under SCIRTSS 4.0 and the results are listed in this paper. The results one of the circuits tested under SCIRTSS 4.0 are compared with the results of the same circuit tested under SCIRTSS 3.0.
CHAPTER 1

INTRODUCTION

With the advent of VLSI (Very Large Scale Integration) digital Integrated Circuits (ICs) containing thousands of gates, it is becoming more difficult to test the ICs completely since the IC has only a few hundred inputs and outputs, and therefore no additional observable test points. Exercising the function of the highly sequential circuit usually requires a huge amount of test sequences and a lot of time on the tester. Stuck-at fault testing is desirable but requires the designer to come up with a test sequence to detect desired portion of all stuck-at faults. An automatic test generation system can obtain a nearly optimum test sequence for detecting these stuck-at faults.

1.1 SCIRTS5

Sequential Circuit Test System (SCIRTS5) is an automatic test generation system which takes a functional description of a digital sequential circuit described in U-AHPL [1]. U-AHPL is a register transfer type language which can be used to describe synchronous and asynchronous sequential circuits. After the designer has completed functional simulation, the U-AHPL can be compiled to a gate-level description to prepare for SCIRTS5. To run
SCIRTSS, the user needs to create a parameter file containing circuit-dependent information and heuristic weights for guiding the search. SCIRTSS runs until the time limit, specified by the user, is reached, or all the detectable faults are detected, or some error occurs. The user can get a print out and analyze the results. SCIRTSS can optionally list the faults detected on each input sequence applied or a summary containing the entire input sequence and a list of all undetected faults and the total percentage faults detected. Usually, the user makes several SCIRTSS runs, adjusting different parameters on each run to detect all the faults in the circuit. To continue with a single circuit with multiple runs, the user can save the state of the circuit at the end of a run and restore that circuit state at the beginning of the next run.

SCIRTSS is an opportunistic system in that, every time an input is applied to test a specific fault, the entire circuit is simulated to find all the possible faults that could be detected with that input sequence. This process is called parallel fault simulation. Only stuck-at-one or stuck-at-zero faults and only a single fault active at any time are considered in SCIRTSS.

1.2 LSSD and Other Test Generation Systems

SCIRTSS is a sequential test generation system which does not require any overhead in hardware, unlike LSSD (Level
Sensitive Scan Design) [2] based test generation systems which require the designer to follow strict design rules and have considerable overhead in hardware and number of circuit inputs and outputs.

LSSD-based design systems enforce strict design rules for testability. Instead of master-slave edge-triggered latches, level-sensitive polarity-hold latches are employed in a LSSD design. Each edge-triggered latch is replaced by a pair of polarity hold-latches (called a SRL, Shift Register Latch) clocked with two non-overlapping clocks for hazard-free operation. Also, the first latch (L1) is fed to the inputs of the second latch (L2) and the output of the second latch is used as the latched signal in the rest of the circuit. This is done so all the SRLs in the circuit can be connected to form a shift register. During test mode, the SRLs are in a shift register mode and any pattern can be shifted in the L2 latches and can provide additional inputs to the rest of the combinational logic of the circuit. Also the L1 contents can be shifted out to observe the values of all the signals going to all the SRLs, thus reducing the test generation problem from a sequential one to a purely combinational one. The D-Algorithm is well known for generating test patterns for detecting faults in a combinational circuit.

The Table Generator Test Generation Algorithm [3] extends the LSSD approach with a partial truth table for
every node (signal) in the circuit such that the input vectors contained in that partial table will drive that node to a logic 0 and logic 1 value. The table also stores the input vectors necessary to sensitize a path from that node to an output of the circuit. The tables are built using a two-pass simulation process involving a forward drive from inputs to outputs and a backward drive from outputs to inputs. The test generation process simply selects two input vectors from the partial table that will sensitize the node and propagate that fault to an output. Inputs and outputs are considered to be the circuit inputs and outputs and the L1 and L2 LSSD latches.

SCIRTSS is an automatic test generation system based on structured functional description language and heuristic-driven test generation algorithms. Another system that is similar to SCIRTSS in that respect is HITEST system [4]. HITEST is a knowledge-based interactive test generation and simulation system. HITEST offers fault modelling for MOS circuitry, knowledge-based heuristics to constrain its search for a possible input sequence, and user interaction with difficult test generation problems. The designer uses three different languages to specify the functional description, the waveforms necessary to drive circuit to a known state, and a gate-level circuit description.
1.3 Review of Previous Work on SCIRTSS

SCIRTSS 3.0 was developed to work on relatively large circuits described in AHPL, a subset of U-AHPL. But it is not capable of handling VLSI circuits since it can only have a maximum of sixty flip-flops and a few hundred gates. It also suffers from inaccurate good and faulty circuit simulations. It is available as a batch process only and is not easily portable since much of it is written in CDC CYBER FORTRAN and ASSEMBLY languages.

One of the primary objectives of this study is to increase the circuit size that SCIRTSS can handle. In addition, an accurate three-value functional simulator is used for SEARCH. The CONTROL PROGRAM, BRIDGE, and SEARCH are written in RATFOR (a preprocessor of FORTRAN) to make the system easily portable. Only a few routines are written in VAX-11 ASSEMBLY language for speeding up the Parallel Fault Simulator. Some errors in the D-Algorithm borrowed from SCIRTSS 3.0 were fixed and the entire D-Algorithm is written in VAX-11 FORTRAN. Finally, the SCIRTSS 4.0 is a much more interactive and user-friendly system.
CHAPTER 2

SCIRTSS OVERALL DESCRIPTION

This paper deals with the CONTROL PROGRAM and the BRIDGE subroutine of the SCIRTSS system. The SCIRTSS automatic test generation program consists of the following:

1. CONTROL PROGRAM
2. BRIDGE subroutine
3. FLTSIM subroutine
4. NDALG subroutine
5. SEARCH subroutine

2.1 Executive

The CONTROL PROGRAM calls all other subroutines to detect faults and also allow the user to change parameters. It also displays percentage total faults detected on every loop of SCIRTSS and acts upon error conditions. In the basic SCIRTSS loop the CONTROL PROGRAM calls the BRIDGE subroutine to select faults for propagation or sensitization, then the CONTROL PROGRAM calls the SEARCH subroutine to find an input sequence to propagate or sensitize that fault and then calls the FLTSIM subroutine to identify all the faults that are detected by that input sequence [5]. The SCIRTSS flow chart is shown in Figure 2.1.
Figure 2.1: SCIRTSS Flow Chart
One of the modifications to this basic loop deletes duplicate start nodes for fault propagation. The second modification makes SCIRTSS interactive. The user is asked to enter file names and other parameters in interactive mode. The user is asked on every loop of SCIRTSS to continue or stop SCIRTSS. The user is also allowed to change parameters through a menu-driven query system. CONTROL PROGRAM is discussed in more detail in CHAPTER 3.

2.2 Subroutine BRIDGE

Subroutine BRIDGE takes results from the FLTSIM and displays them and compares the final states of SEARCH and FLTSIM to ensure that the two simulators are in step. Then BRIDGE sets up the start nodes for fault propagation mode or goal nodes for sensitization mode for subroutine SEARCH.

BRIDGE also calls subroutine NDALG to find test vectors for sensitizing gate faults. BRIDGE converts NDALG test vectors into a set of goal nodes for SEARCH. Subroutine BRIDGE is discussed in more detail in CHAPTER 4.

2.3 Subroutine FLTSIM

Subroutine FLTSIM is the parallel fault simulator. FLTSIM is responsible for maintaining the list of all detected and undetected faults. As the name implies, FLTSIM, using the fault-injection gate-level simulator, simulates all the faulty states of the circuit in parallel and is thus able to find all detected faults by a particular input sequence.
The input sequences are specified by the user as initial input sequences or generated randomly by the TASMBLE subroutine [5,6] or generated by subroutine SEARCH.

FLTSIM uses the elemental logic simulator FSIM for three-value good and faulty simulation. The three logic values are 0, 1, and U. Zero and one are the normal logic values and U represents an unknown value. Subroutines TASMBLE and ASM3 read in the gate-level circuit description, assign simulation order and also assign the fault priority to all the gates and flip-flops. Subroutine FSIM can simulate AND, OR, NAND, NOR, and XOR gates and D, SC, and JK flip-flop.

The FLTSIM subroutine was adopted for the VAX version of SCIRTSS by Zou Guang-Ran [5] from the CYBER version [5,7].

2.4 Subroutine NDALG

The NDALG subroutine is used in SCIRTSS to generate test vectors for sensitizing gate faults since the D-Algorithm will generate an input vector to sensitize a path from the fault location to a circuit output for any given fault in a combinational circuit. The D-Algorithm will generate a set of test vectors if and only if a sensitization path exists, otherwise it will return no test vectors.

The D-Algorithm basically finds a test vector to propagate a selected fault by tracing forward to a circuit
output or a flip-flop since each flip-flop is considered to be a pseudo input or output. Once a fault propagation input vector is determined, D-Algorithm tries to obtain a sensitization input vector to sensitize the fault, that is, to set inputs to a value needed to make the fault line assume the opposite value of the stuck-at value of the fault. Once the sensitization vector is determined, it is merged with the propagation input vector to obtain a final test input vector. If the sensitization and propagation input vectors collide (require different values for same inputs), D-Algorithm tries another path for propagation and sensitization until it has exhausted all possible paths, or runs out its time limit set by BRIDGE, or memory space is exceeded. In any case, if any test vectors are found, D-Algorithm returns distinct test vectors to BRIDGE.

The NDALG is written entirely in VAX FORTRAN and was adopted for the VAX version of SCIRTSS by Wang [8].

2.5 Subroutine SEARCH

Subroutine SEARCH [9,10,11] is a heuristic-based algorithm to obtain test input sequences to either propagate a stored fault or sensitize a particular fault. Subroutine SEARCH includes a three-value functional simulator that uses the same executable tables that the HPSIM4 [12] two-value functional simulator uses.
For fault propagation mode, SEARCH receives the present good state of the circuit and several start nodes. Each start node is a faulty state of the circuit associated to a particular fault stored in a flip-flop. SEARCH takes a start node and attempts to propagate the fault to a circuit output. Using calculated heuristic function values, SEARCH goes from one state to another. SEARCH calls the functional simulator twice, one for the good state of the circuit and the second time for the faulty state of the circuit to go to the next state. SEARCH maintains two sets of nodes, different states of the circuit. One set, SETB, contains all the expanded nodes after applying the data and control input vectors. SEARCH chooses a node with the lowest heuristic function value from SETB and adds it to the other set, SETA, where all the nodes in the current propagation are maintained. After every node is generated, SEARCH checks to see if the fault effect is observable on an output. When the fault effect is observable on an output, SEARCH stops and generates a set of input sequences it took to propagate that fault [5,9].

For fault sensitization mode, SEARCH starts with the present good state of the circuit, the start node, and a set of goal nodes, each corresponding to a distinct flip-flop or gate fault. To sensitize the selected fault SEARCH drives the circuit to the goal node state. SEARCH uses the functional simulator to simulate the good state of the
circuit from start state to goal state, and SETA and SETB nodes with heuristic values to obtain an optimum input sequence set.

The SEARCH subroutine and the three-value functional simulator were written by Alaa Mohsseni [9] for the VAX version of SCIRTSS.

2.6 AHPL SCIRTSS Interface

The circuit to be tested is described in U-AHPL and is functionally simulated using the HPSIM4 [12] functional simulator. To prepare for test generation the user first compiles the U-AHPL circuit description into executable tables for the SEARCH functional level simulator using STAGE 1 compiler. The output of this compiler is run through second and third STAGE compilers in one run and a gate list of the network is generated for the FLTSIM parallel fault simulator and NDALG. The executable tables and the gate-level network description are inputs to the SCIRTSS program as shown in Figure 2.2.
Figure 2.2: Compiler Driven SCIRTSS
CHAPTER 3

SCIRTSS CONTROL PROGRAM

SCIRTSS is made of many subroutines (Appendix A). The main program (CONTROL PROGRAM) calls various major subroutines, TASMBLE, FLTSIM, BRIDGE, and SEARCH [13]. Each of these subroutines in turn call other subroutines to perform their function. This chapter will describe the interfaces between the major subroutines and BRIDGE since BRIDGE and the CONTROL PROGRAM were written by the author. The functional flow of the CONTROL PROGRAM is described at the end of the chapter.

3.1 Interfaces

The CONTROL PROGRAM communicates to all the major subroutines through COMMON data variables. Using COMMONs to pass parameters back and forth is an efficient way to make the programs more modularized and still avoid the penalty of passing large numbers of variables.

The array NRY is used by all the subroutines and is in COMMON area:

COMMON ARRAY NRY(MAXNRY)

where MAXNRY is the size of the array. The size of NRY can be set by changing the value of MAXNRY in file DEFNS.RAT and recompiling the program.
3.1.1 BRIDGE and CONTROL PROGRAM

Subroutine BRIDGE and the CONTROL PROGRAM communicate via COMMONs and the variable INEW which is passed to BRIDGE with a value and BRIDGE returns it as a result to the CONTROL PROGRAM. Variable INEW is set to the number of faults detected for a particular loop of SCIRTSS. If the SEARCH mode was fault propagation and no new faults were detected and there was no SEARCH errors, INEW is set to -1 to indicate to BRIDGE to make the faults that were selected for propagation ineligible until the end of the SCIRTSS run. BRIDGE returns a value of 0 in INEW if there were no errors. A negative value of INEW is returned by BRIDGE if the BRIDGE loop limit (unsuccessful BRIDGE attempts to select eligible faults for propagation or sensitization) is exceeded. A positive value is returned in INEW if BRIDGE detects that all the undetected faults have been tried twice and still are undetected. SCIRTSS stops with an error in these last two conditions.

Only the variables used in the BRIDGE subroutines from the following COMMONs are described in detail. COMMON A is used by BRIDGE to get circuit parameters and other user supplied parameter values.

COMMON /A/ iw(60), iz(121), nrff, nexin, krnch, nrydim, limloop, nrand, ndrand, kswitch, tstart, tlimit
LIMLOOP maximum number of consecutive non-productive SCIRTSS loops allowed.

KSWITCH optional switches record where a bit of value 0 is off and a bit of value of 1 is on for a switch condition. Bit position 1 of KSWITCH is the MSB and position 32 is the LSB. Bit positions 27 to 32 represent switch(1) to switch(6), respectively.

TSTART real variable which holds the start time in seconds for SCIRTSS relative to the system timer.

TLIMIT time limit in seconds specified by the user for SCIRTSS.

COMMON /RETA/ kretain, krestor

KRETAIN flag for retaining the good and all faulty states of the circuit at the end of a SCIRTSS run.

LRESTOR flag for restoring the state of the circuit that was saved by a previous run of SCIRTSS.

3.1.2 FLTSIM and BRIDGE

Subroutine FLTSIM provides BRIDGE with the good and all the faulty states of the circuit and also a vector of faults that are detected. The TASMBLE subroutines provide BRIDGE with the priority class of each element and its associated faults. BRIDGE, FLTSIM, and TASMBLE use the following COMMONs. Note that only the variables used in BRIDGE subroutines are described in detail.
COMMON /A/ iw(60), iz(121), nrff, nexin, krunch, nrydim,  
limloop, nrand, ndrand, kswitch, tstart, tlimit

IW(60) used as a temporary array to pass parameters between subroutines.

NRFF total number of flip-flop in the circuit, both data and control flip-flops.

NEXIN number of external inputs to the circuit.

NRAND maximum number of SEARCH start nodes for fault propagation or start nodes for sensitization to be generated by BRIDGE.

NDRAND maximum number of D-Algorithm output vectors allowed.

COMMON /F/ newflt, iflfnf, nvect(6), mvice(6), mvff(8), mxpc,  
zz, mask, za, kdiagnx, mptr, nrpg, eltyp(20),  
maxtype, idff, jxor, jmacro, zpf(27), nrpg, icnt,  
ityp, ifan, ilnk, impi, ipfv, iout, iorder, lnko,  
kgu, konst0, konst1, nfo, zl, ngprpg, nrpgf, nmax,  
maskg, nfound, limit, zb(2), nlist, nl, nlist, ipnt,  
nzabb, ze(7), ifv(7), nrft, nrout, iotpt, nrft,  
ibit, icw, nrpgsv, zc, lnkosv, jstart, jend

NEWFLT pointer to array NRY where a vector of NRPG words is reserved for use by BRIDGE and FLTSIM for keeping a temporary fault vector.

IFLNFND pointer to array NRY where the vector of faults detected resides.
NVECT(6) six pointers to array NRY where the fault vectors used are:

(1) - vector of faults attempted on last loop of SCIRTSS.

(2) - SEARCH graph level exceeded error faults, and SCIRTSS non-productive loop faults, and FLTSIM and SEARCH final states different error faults.

(3) - SEARCH SETB empty error faults vector.

(4) - fault propagation made ineligible faults due to faulty states having control states with value unknown faults.

(5) - maximum number of SEARCH SETA or SETB nodes exceeded faults and SEARCH simulator call limit exceeded faults.

(6) - faults currently ineligible for D-Algorithm sensitization.

MVPC(6) six pointers to array NRY where priority class masking vectors are stored.

(1) - priority class 0 masking vector pointer.

(2) - priority class 1 masking vector pointer.

(3) - priority class 2 masking vector pointer.

(4) - priority class 3 masking vector pointer.

(5) - UNUSED

(6) - UNUSED
MXPC maximum fault priority class number. For example, if MXPC is 4 then the priority classes are 0, 1, 2, and 3. The fault priority classes are described in detail in CHAPTER 4.

ICNT total number of elements in the circuit, gates and flip-flops.

KGW word number in the fault vector where the good state of that element is stored in the least significant bit.

NRPGF number of words used for flip-flop faults masks in a vector.

NFOUND current total number of faults detected.

NROUT number of outputs of the circuit.

NRFLT total number of faults in the circuit.

NRPGSV number of words in a fault vector.

LINKOSV pointer to array NRY where a list of pointers is stored where each pointer points to a vector of the good and faulty state of an element.

COMMON /S/ ns(8), indx, indxb, marka, markb, omega, defh, ivrfnd, nlimit, iprnt, nrnod, hmode, wt(16), jclass, ngoals, nrstart, schmod, mff(8), ksrcerr, nodrset(8), nresets, insave(8), nsg(4)

MARKA pointer to array NRY where BRIDGE uses temporary vectors.

NRNOD number of control states in the circuit.
3.1.3 NDALG and BRIDGE

BRIDGE calls subroutine NDALG to get a set of test vectors for sensitizing a gate fault. First BRIDGE sets up the element number, the input or output involved, and the stuck-at value of the fault. Subroutine NDALG returns results in the arrays NRY and CNR. Array CNR is a byte array where the different tests generated by NDALG are stored.

COMMON /LOGI/ CNR(MAXCNR)

Where MAXCNR is the size of the array and defined in file defns.rat. CNR is a byte array.

COMMON /A/ iw(60), iz(121), nrff, nsexin, krnch, nrydim, limloop, nrand, ndrand, kswitch, tstart, tlimit

IW(1) gate number specified by BRIDGE on which the fault occurs.

IW(2) the input number on which the fault occurs. iw(2) is set to zero for output stuck at faults.

IW(3) fault type, 0 for stuck-at-0 fault and 1 for stuck-at-1 fault.

IW(4) number of test vectors returned by NDALG.

IW(5) number of test vectors requested by BRIDGE.

IW(6) number of elements in the circuit partition for the fault involved. Returned by NDALG.

IW(7) pointer to list in array CNR of element numbers in the original circuit corresponding to the element number of the partitioned circuit. Element j of
the partition corresponds to element \( NRY(iw(7)+j) \)
in the original circuit.

**nw(8)**
pointer to list in array CRN of tests returned by NDALG. Test 1 is returned in \( CRN(iw(8)+1) \) through \( CRN(iw(8)+iw(6)) \). The next test is at \( CRN(iw(8)+iw(6)+1) \), etc.

**nw(9)**
NDALG time limit set by BRIDGE for NDALG.

### 3.1.4 SEARCH and BRIDGE

Subroutine BRIDGE and subroutine SEARCH communicate via the COMMONS SCH1, SCH2, SCH3, B, and S. The COMMONS are used to pass the SEARCH mode, number of start or goal nodes and the start or goal nodes.

**COMMON /SCH1/**
- setb(2,MAXSTB),
- seta(2,MAXSTA),
- sucnod(4,MAXGN),
- setblk(MAXBNDS),
- setalk(MAXANDS),
- lastlkb, lastlka, valnoda

**VALNODA**
number of valid nodes in SETA array when subroutine PNODES is called to print the SETA nodes.

**COMMON /SCH2/**
- minheu(WC00002),
- heurfn, ndffb, nroutb, nexinb,
- nodesiz, offset, nbussb

**NDFFB**
number of bytes reserved for data flip-flops in the node structure.

**NROUTB**
number of bytes reserved for outputs in the node structure.

**NEXINB**
number of bytes reserved for inputs in the node structure.
NODSIZ   size of the nodes in bytes.

COMMON /SCH3/ success, tvisit, nodbyt, omgahd, psswb, psswp,
pssud, fpswp, fpsud, fpsuf, csvisit(MAXMOD,MAXCS),
acsptr, noutptr, nodlvl, maxten, scherr, strtin,
lastin, ffbool(MAXGNDS), fomrit(MAXGNDS), avemrt,
maxmrt, modlcnt

ACSPTR   byte number of the number of active control states.
Set by subroutine BLDMAP.

SCHERR   SEARCH error number:
0 - no error
1 - graph level exceeded
2 - SETB empty
3 - SEARCH simulator call limit exceeded
4 - maximum number of SETA nodes exceeded
5 - maximum number of SETB nodes exceeded

COMMON /B/ gnnodet2,MAXGN), fnnode(2,MAXFN), memap(MAXMEM),
eximap(MAXINP), outmap(MAXOUT), bitobyt(MAXBVY, 2),
busmap(MAXOUT), stkfip(MAXNODS), stkffm(MAXNODS),
stktyp(MAXNODS)

GNNODE   2-byte array for the good network node. GNNODE(1,i)
is for the unknown value of the elements and
GNNODE(2,i) is for the known value of the elements.

FNNODE   2-byte array for the faulty network node.
FNNODE(1,i) is for the unknown value of the
elements and FNNODE(2,i) is for the known value of
the elements. MAXFN is ten times as large as MAXGN
since the maximum number of start nodes or goal
nodes that BRIDGE can select is ten.

**BITOBYT**

is a two-dimensional array where BITOBYT(i,1) is a
pointer and BITOBYT(i,2) is a mask array. The
pointer array is used to point, for each
U-AHPL-declared element in the circuit, to the
SEARCH node structure. The mask array is the
Corresponding valid mask array for each element in
the byte node structure. Each array is two bytes
wide.

**STKFFP**

2-byte pointer where each location is a pointer to
the start nodes generated by BRIDGE for SEARCH for
propagation mode. The pointer is zero for all
stored faults except for stored flip-flop faults.
This is done so SEARCH can preserve the stored
fault effect after each simulation.

**STKFFM**

2-byte mask array corresponding to the STKFFP array
for each start node.

**STKTYP**

2-byte fault type array corresponding to the STKFFP
array for each start node. STKTYP is 0 for
stuck-at-0 faults and 1 for stuck-at-1 faults.

**COMMON /S/**

ns(8), indxa, indxb, marka, markb, omega, defh,
ivrfnd, nlimit, iprnt, nnnod, hmode, wt(16),
jclass, nrgoals, nrstart, schmod, mff(8), ksrcerr,
nodrset(8), nresets, insave(8), nsg(4)

**NLIMIT**

SEARCH simulator call limit set by BRIDGE.
**NRGOALS** number of goal nodes for embryonic sensitization mode or D-Algorithm sensitization mode. Set by BRIDGE.

**NRSTART** number of start nodes for fault propagation mode. Always one for sensitization modes. Set by BRIDGE.

**SCHMOD** SEARCH mode set by BRIDGE. For fault propagation mode SCHMOD is 0, for embryonic sensitization mode SCHMOD is 1, and for D-Algorithm sensitization mode SCHMOD is 2. Refer to CHAPTER 4 for a detailed description of each SEARCH mode.

### 3.2 Functional Flow

Since SCIRTSS is written in RATFOR, the CONTROL PROGRAM is much more readable than FORTRAN. Also, CONTROL PROGRAM was written with comments to explain the functions of the code in subroutines. SCIRTSS functional flow is described in Figure 3.1.
initialize timer and circuit parameters
read general param. and other optional param. files
if any initial input sequence
    call FLTSIM to apply it
if switch(4) set then stop
repeat for ever {
    mark program time
    print faults found
    if all faults found then exit repeat loop
    continue?
    if no then exit repeat loop
    change parameters?
    if yes then call change parameters menus
    if productive loop then reset loop counter
    else {
        increment loop counter
        if loop limit exceeded then
            print error message and exit repeat loop
        if previous SEARCH propagation and no SEARCH errors
            print err. msg. and notify BRIDGE via IHEW = -1
    }
call BRIDGE
mark BRIDGE and total time
if BRIDGE returns error then
    print error message and exit repeat loop
if SCIRTSS time limit exceeded then exit repeat loop
if propagation mode then
    delete any duplicate start nodes
print BRIDGE results
call SEARCH
if any SEARCH errors then {
    dump last SETA nodes to a file?
    if yes then
        dump the SETA nodes to a user specified file
} else
    call FLTSIM
} # end repeat loop
print SCIRTSS results
if retain flag set then
    retain state of the circuit to a user specified file
end

Figure 3.1: SCIRTSS Functional Flow
CHAPTER 4

BRIDGE

The function of subroutine BRIDGE is to select faults for propagation or sensitization. Based on the faults selected, BRIDGE sets up the proper start nodes or goal nodes for propagation or sensitization, respectively. Since the faulty state of the circuit is stored in FLTSIM format and the nodes for SEARCH are to be in its format, BRIDGE also converts from one format to another.

4.1 Fault Categories

BRIDGE separates faults into three basic categories for fault selection:
1. Faults stored in flip-flops (from here are on these will be referred to as stored faults).
2. Flip-flop faults.
3. Gate faults.

For a fault to be considered to be a stored fault, the first category, its effect must be observable at the output of the flip-flop. Only the stored faults are eligible for propagation, that is, the stored fault's effect is propagated to a circuit output. For each stored fault selected for propagation by BRIDGE, a start node is generated for SEARCH which simulates the faulty circuit for an input
sequence tree and tries to observe the fault effect on an output. Each start node corresponds to a faulty state of the circuit specified by FLTSIM for that stored fault.

In the second category are flip-flop faults. Flip-flop faults are modeled as the flip-flop output stuck-at-0 or stuck-at-1. Only flip-flop faults which are not stored in any flip-flops, even the flip-flop which is the location of the fault, are eligible for embryonic sensitization. For each fault selected for embryonic sensitization, a goal node is specified by BRIDGE for SEARCH. A goal node consists of the state of the circuit at which the particular fault will be sensitized. For this, all the flip-flops are set to don't-care value and the flip-flop which contains the fault is set to the opposite value of the stuck-at signal.

Category three faults are eligible for D-Algorithm sensitization in which only gate faults which are not stored in any flip-flops are considered. The gate faults are input or output stuck-at-0 or stuck-at-1 faults. For a gate fault selected for D-Algorithm sensitization, BRIDGE sets up the following before calling NDALG:

- the gate number on which the fault occurs.
- the input number of the gate on which the fault occurs or 0 for output of the gate.
- fault type, 0 for stuck-at-0 and 1 for stuck-at-1.
- number of test vectors requested by BRIDGE.
NDALG return the test vectors and a list of the elements that were in the circuit partition for the fault involved. Each test vector specifies the state of the elements involved in the partition. BRIDGE takes the values of the flip-flops and circuit inputs specified by NDALG and sets all other flip-flops and circuit inputs to don't-cares. BRIDGE builds a goal node for each test vector for SEARCH.

### 4.2 Fault Priority Classes

For an opportunistic fault selection process, a fault priority classification is employed. The user supplies the priorities of circuit flip-flops. An integer value is assigned to any or all of the flip-flops as follows:

- 0 - easy
- 1 - normal
- 2 - difficult
- 3 - very difficult

The values above correspond to whether it is expected to be easy, normal, difficult, or very difficult to propagate a fault stored in the particular flip-flop to a circuit output. The default value is zero for any flip-flops not specified by the user.

During initialization, the program (subroutine ASM3 in TASMILE [6]) starts with the flip-flop priorities specified by the user and classifies each individual fault.
in the circuit according to the algorithm [5] described in Figure 4.1.

```
for each unspecified flip-flop {
    classify it in class 0
}
for each control flip-flop in class 0 {
    re-classify it to class 1
}
for each gate {
    classify it to class 3
}
for each gate {
    for each gate in its fanout (recursively) {
        find the gate with minimum priority class
        assign the gate of the input that priority class
    }
}
for each priority class {
    allocate a vector in array NRY
}
for each element {
    put the output faults into the vector
    corresponding to the priority class of that element
}
for each gate {
    for each input fault of the gate {
        put the input fault in the vector corresponding
        to the priority class which is the maximum of the
        class of that gate and the class of the elements
        connected to that input
    }
}
end
```

Figure 4.1: Fault Priority Classification

4.3 Fault Selection

The fault selection process is an opportunistic one where difficult faults are selected first for propagation and easy faults are selected for sensitization. This means that
the first step is to select difficult faults stored in flip-flops from which it is easy to propagate a fault. First priority class 3 faults stored in priority class 0 flip-flops are selected. If there are none eligible then priority class 2 faults stored in priority class 0 flip-flops are selected and so on for priority class 1 and priority class 0 faults stored in priority class 0 flip-flops. If there are no stored faults in priority class 0 or 1 flip-flops, priority class 0 or 1 flip-flop faults are selected for embryonic sensitization. If there are no eligible priority class 0 or 1 flip-flop faults, faults stored in priority class 2 or 3 flip-flops are selected for propagation. If there are no eligible stored faults in priority class 2 or 3 flip-flops, priority class 2 or 3 flip-flop faults are selected for embryonic sensitization. If there are no eligible stored or flip-flop faults, an easy gate fault is selected for D-Algorithm sensitization.

Flip-flop faults selected for embryonic sensitization and gate faults selected for the D-Algorithm sensitization are re-classified to priority class 3 to ensure that sensitized faults stored in flip-flops are selected first for propagation.

For any mode, propagation or sensitization, if there is a SEARCH error or the final states of SEARCH and FLTSIM do not match, the faults that were selected by BRIDGE are made ineligible for selection by adding them to ineligible
faults vectors. These ineligible faults are made eligible for selection at the end of the SCIRTSS run. If any errors occur during this portion of the run, some parameters like SEARCH simulator call limit are increased and these faults are tried only one more time. If any errors occur during the second try, BRIDGE return with an error and SCIRTSS stops.

Instead of selecting a fixed fault selection process, the user is allowed to modify this fault selection process by specifying it in the second level optional parameters file. The array PCORDER in COMMON /PCO/ is used to specify the fault selection order. The array PCORDER is of size 12 and the values are 1, 2, 3, 4, -1, -2, -3, -4, and 0 where the values correspond to:

1 select faults stored in priority class 0 flip-flops for propagation
2 select faults stored in priority class 1 flip-flops for propagation
3 select faults stored in priority class 2 flip-flops for propagation
4 select faults stored in priority class 3 flip-flops for propagation
-1 select priority class 0 flip-flop faults for embryonic sensitization
-2 select priority class 1 flip-flop faults for embryonic sensitization
-3 select priority class 2 flip-flop faults for embryonic sensitization
-4 select priority class 3 flip-flop faults for embryonic sensitization
0 select gate faults for D-Algorithm sensitization

The selection order is from PCORDER(1) through PCORDER(12).

For an opportunistic fault selection process, the default value of the PCORDER array is:

```
PCORDER()

| 1 | 2 | -1 | -2 | 3 | 4 | -3 | -4 | 0 | 0 | 0 | 0 |
```

4.4 Functional Flow

The BRIDGE functional flow is described in Figure 4.2.
make failed faults ineligible
get stored faults and good network state
if no SEARCH error then
    if SEARCH and FLTSIM final states differ then
        make the faults selected last time ineligible
repeat for ever {
    get stored fits in all FFs of all priority classes
    for PCORDER(1) thru PCORDER(12) {
        if PCORDER(i) is positive then {
            if there are any stored faults in priority class
                (i-1) FFs, randomly select NRAND faults from that
                class and generate start nodes for each fault.
        }
        else {
            if any undetected faults in priority class (-i-1)
                FFs, randomly select NRAND faults from that class
                and generate goal nodes for each fault
        }
        if any faults selected then exit repeat loop
    }
    if no eligible stored nor flip-flop faults then {
        for priority class 0 to MXPC
            if any undetected gate faults of that class then
                call NDALG and gen. goal nodes on test vectors
            if no eligible fits for D-Alg. sens. then {
                if BRIDGE unsuccessful loop limit exceeded then
                    return with error
                if 1st failure to select any faults then {
                    set 1st failure flag, double SEARCH sim. limit
                    make NVECT(4,5,6) faults eligible
                }
                else if 2nd failure to select any faults then {
                    set 2nd failure flag, double SEARCH sim. limit
                    make all ineligible faults eligible
                }
                else return with fatal error
        next iteration of repeat loop
    }
}
exit repeat loop
} # end repeat loop
end

Figure 4.2: BRIDGE Functional Flow
CHAPTER 5

DATA STRUCTURES

The data structures used in the CONTROL PROGRAM and the subroutines are discussed in this chapter. The array NRY which is common to all subroutines is described first, followed by the data structures used in BRIDGE and then the data structures used between BRIDGE and NDALG and between BRIDGE and SEARCH.

5.1 NRY

Array NRY is of size MAXNRY by 32 bits (VAX word size). It is used to store the fault priority classes, the circuit description, element types, and fanin and fanout lists of the elements, see Figure 5.1. It is also used to store such things as the good and all faulty states of each flip-flop and the detected fault vector, see Figure 5.2. BRIDGE uses NRY to keep a set of vectors of ineligible faults and also for storing temporary fault vectors during fault selection. NDALG makes use of the circuit topology stored in NRY to keep a partition of the circuit that it is processing. SEARCH uses NRY to pass the input sequence it generates to FLTSIM, see Figure 5.3.
Figure 5.1: Array NRY(1:NFO)
<table>
<thead>
<tr>
<th>NRY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(1)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(2)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(3)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(4)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(5)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← NVECT(6)</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(1) p.c. 0 masking</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(2) p.c. 1 masking</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(3) p.c. 2 masking</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(4) p.c. 3 masking</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(5) unused</td>
</tr>
<tr>
<td>NRPG words</td>
<td>← MVPCC(6) unused</td>
</tr>
</tbody>
</table>

Figure 5.2: Array NRY(NVECT, MVPC)
Figure 5.3: Array NRY(LNKO:NRYDIM)
5.2 BRIDGE

BRIDGE uses COMMON BRG to keep its local variables and arrays that are used by its different subroutines.

COMMON /BRG/ GBIT(MFPPO), FBIT(MFPPO), lpc(6), kprop, nodal, mrnd, mk, flag19, flag20, firstflg

GBIT GBIT is a byte array used to store the current good state of the circuit. MFPPO is maximum the number of flip-flops plus maximum number of circuit inputs plus maximum number of circuit outputs. MFPPO can be changed in file defns.rat. The value of MFPPO is 1400.

FBIT FBIT is a byte array used to store the faulty state of the circuit associated with a stored fault selected for propagation, the goal state for a flip-flop fault selected for embryonic sensitization, or the goal state for a gate fault selected for D-Algorithm sensitization.

LPC LPC(1) through LPC(4) are pointers to vectors in NRY where the stored faults in priority class 0 through 3, respectively, are stored by BRIDGE for the present faulty state of the circuit.

KPROP pointer to NRY to a vector of stored faults of all priority classes eligible for fault propagation.

NODALG pointer to NRY where a vector of faults ineligible for D-Algorithm sensitization are stored.
MRAND number of faults to be selected randomly. Specified by the user.

MK variable to save the pointer value MARKA during BRIDGE.

FLAG19 is set if first loop of BRIDGE where no eligible faults were found for propagation or sensitization.

FLAG20 is set if second loop of BRIDGE where no eligible faults were found for propagation or sensitization.

FIRSTFLG flag is set when BRIDGE is called first time and is reset afterwards.

For the good state of the circuit, array GBIT is used to store the good value of each flip-flop. For flip-flop n, the present good state of the circuit is obtained from the array NRY as shown below:

```
NRY

LNKO+n  Pointer to the fault vector for flip-flop n.

NRY(LNKO+n)

NRY(LNKO+n+KGM)

NRY(LNKO+n+NRPG+KGM)

unknown value

known value
```

For flip-flop n, the present good state of the circuit is obtained from the array NRY as shown below:
The good state is saved in array GBIT. The control state flip-flop values are stored first and then the data flip-flop values are stored in array GBIT as shown:

```
1
GBIT
```

- Control flip-flops

```
NRNOD
```

- Data flip-flops

```
NRFF
```

The values stored in GBIT for each flip-flop are derived from the good known and the unknown states of each flip-flop as shown below:

<table>
<thead>
<tr>
<th>Known</th>
<th>Unknown</th>
<th>Flip-flop value</th>
<th>GBIT(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>'0'</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>'U'</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>'1'</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

For a stored fault in flip-flop n, the faulty state of the circuit is obtained from array NRY at pointer LNKO. The fault mask vector, a single bit is one in the entire NRPG words, is used to get the value of each flip-flop's faulty known and unknown values from the corresponding known and unknown vectors, each NRPG words long, as shown below:
The faulty state is saved in array FBIT. The control state flip-flop values are stored first and then the data flip-flop values are stored in array FBIT as shown:

```
<table>
<thead>
<tr>
<th></th>
<th>FBIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

NRNOD |

NRFF  |

Control flip-flops

Data flip-flops

<-- LNK0+n  Pointer to the fault vector for flip-flop n.

<-- NRY(LNK0+n)

<-- known faulty value of flip-flop n

<-- NRY(LNK0+n+NRPG)

<-- unknown faulty value of flip-flop n
The values stored in FBIT for each flip-flop are derived from the faulty known and the unknown states of each flip-flop as shown below:

<table>
<thead>
<tr>
<th>Known</th>
<th>Unknown</th>
<th>Flip-flop value</th>
<th>FBIT(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>'0'</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>'U'</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>'1'</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>invalid</td>
<td></td>
</tr>
</tbody>
</table>

For the embryonic sensitization mode, an unstored flip-flop fault is selected. BRIDGE uses the array FBIT to specify which flip-flop fault is to be sensitized. All the flip-flops are set to don't-care (1) values except for the flip-flop that has the fault. That flip-flop FBIT value is 0 if the fault is a stuck-at-one, or FBIT value is 1 if the fault is a stuck-at-zero.

5.3 NDALG

BRIDGE and NDALG communicate via arrays NRY and CNR. CNR is a byte array used by NDALG to return the test vectors required to sensitize the gate fault selected. The test vectors are element values specified by NDALG. Since NDALG only works on a part of the circuit that is affected by the fault in question, it returns in NRY a list of elements involved in the partition. The FLTSIM given element numbers are given in NRY at location given given by IW(7). The number of elements in the partition is given by IW(6) as shown below:
BRIDGE converts the element values in the partition to FBIT representation and assigns a don't-care to the flip-flops and external inputs of the circuit not in the partition. The test vectors are stored in array CNR as shown below:

The following conversion table is used to convert from NDALG element value to FBIT element value.
5.4 SEARCH

BRIDGE sets up the start and goal nodes for SEARCH in the GNODE and FNODE arrays. The two-dimensional GNODE and FNODE arrays are equivalent to IGNODE and IFNODE two byte arrays as shown:

\[ \text{GNODE}(2, \text{MAXGN}) = \text{IGNODE}(\text{MAXGN}) \]

\[ \text{GNODE}(1, i), \text{GNODE}(2, i) \]

\[ \begin{array}{c}
\text{IGNODE}(i) \\
\end{array} \]

\[ \begin{array}{c}
\text{8 bits} \quad \text{8 bits} \\
\end{array} \]

\[ \quad \text{16 bits} \]

Similarly:

\[ \text{FNODE}(1, i), \text{FNODE}(2, i) \]

\[ \begin{array}{c}
\text{IFNODE}(i) \\
\end{array} \]

\[ \begin{array}{c}
\text{8 bits} \quad \text{8 bits} \\
\end{array} \]

\[ \quad \text{16 bits} \]

In either of the arrays, a location is used to store values of up to eight elements. For element \( n \):

\[
\begin{array}{c|c|c}
\text{unknown value} & \text{known value} & \text{logic value} \\
0 & 0 & '0' \\
0 & 1 & 'U' \\
1 & 1 & '1' \\
\end{array}
\]

unknown or don't-care

The node, either start or goal, has the same structure described in Figure 5.4.
For the good network node, the array GBIT is converted to GNNODE. The two dimension array BITOBYT is used to convert from GBIT to GNNODE. BITOBYT(i,1) is a 16-bit pointer array to point to the proper location in the SEARCH node.

BITOBYT(i,2) is a 16-bit mask array for locating the element inside the two bytes of the SEARCH node. The mask values for element n are given below:

<table>
<thead>
<tr>
<th>GBIT(n)</th>
<th>BITOBYT(n,2) (in binary)</th>
<th>IGNODE(BITOBYT(n,1)) (in binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00100000 00100000</td>
<td>xx0xxxxxx xx0xxxxxx</td>
</tr>
<tr>
<td>1</td>
<td>00100000 00100000</td>
<td>xx0xxxxxx xx1xxxxxx</td>
</tr>
<tr>
<td>2</td>
<td>00100000 00100000</td>
<td>xx1xxxxxx xx1xxxxxx</td>
</tr>
</tbody>
</table>

The array BITOBYT has a location for every data flip-flop, every external input, and every external output. Values for all data flip-flops, external inputs, and external outputs are converted in the same way. For control state flip-flops the value is converted to SEARCH node differently for propagation and sensitization modes.

For fault propagation mode, only the control states that are active (logic value of 1) are used in the start node. At ACSPTR in the start node the number of active control states is stored followed by the active control state codes. The array MODNUM is used to get the active control state codes. MODNUM contains a value for each control state in the circuit. Since SCIRSS can have multiple modules and FLTSIM does not distinguish between modules, FLTSIM orders the
Memory section
(NDFFB)

External Inputs section
(NEXINB)

External Outputs section
(NROUTB)

No. of Active Control States

Active Control State Codes
.
.

Unused

Predecessor Node Number

Node Level

Heuristic Value

Figure 5.4: SEARCH Node Structure
control states as they are declared in U-AHPL description. However, SEARCH requires the active control states in each module. Therefore, MODNUM contains the value of one thousand times the module number plus the control state number within that module. For example, if control state two in module three and control state one in module four were active, a section of start node would appear as:

```
Start Node

ACSPTR ->
  2
  3002
  4001
  0
  0
```

For embryonic or D-Algorithm sensitization modes, control states with don't-care values are ignored. At ACSPTR in the goal node, the number of control states with logic values of 0 or 1 is stored. Control states having a value of logic 1 are specified by the same type of active control code as for the propagation mode start nodes. Control states having a value of logic 0 are specified by the same code as for the logic 1 value active control codes except that the codes are negated. For example, control state four in module two has value of logic 1 and control state one in module one
has value of logic 0, a section of sensitization goal node would appear as:

<table>
<thead>
<tr>
<th>Goal Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSPTR</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2004</td>
</tr>
<tr>
<td>-1001</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

For fault propagation mode, the start nodes are given in array IFNODE. First NODSIZ words are for start node number one and the next NODSIZ words are for start node two and so on. Array IGNODE contains the good state of the circuit in one node. Before the start nodes are given to SEARCH, after being generated by BRIDGE, any duplicate start nodes are deleted from array IFNODE by shifting other distinct nodes up in the array.

For embryonic and D-Algorithm sensitization modes, the goal nodes are arranged in similarly to the start nodes in fault propagation mode in array IFNODE. The start node for SEARCH is provided in array IGNODE.
CHAPTER 6

EXTENDING PREVIOUS WORK

Old SCIRTSS (version 3.0) is available on the CDC CYBER computer. It can be run only in batch mode and is limited in the circuit size it can handle for VLSI applications.

One of the objectives of this study was to increase the circuit size that new SCIRTSS (version 4.0) can handle. The total number of flip-flops in a circuit has been increased from 60 to 999. The total number of gates in a circuit has been increased from 500 to 10,000. For VLSI applications, the number of circuit inputs and outputs has been increased from 60 each to 200 each.

Some of the other objectives included multiple module capability. Old SCIRTSS can handle only one module where new SCIRTSS handles up to ten modules. Also new SCIRTSS can have multiple active control states simultaneously where old version can have single active control state simultaneously.

SCIRTSS 4.0 conforms to U-AHPL syntax, that is, functions as combination logic units are available compared to no combinational logic units and subset of U-AHPL syntax for old SCIRTSS.
Since the SEARCH functional simulator in new SCIRTSS is more accurate than the three different simulators that are used by SEARCH in old SCIRTSS, new SEARCH has less errors than the old version. New SEARCH also preserves the fault effect on a flip-flop fault stored in a flip-flop. Old SEARCH does not do this and consequently is not as efficient for fault propagation. Also BRIDGE removes any duplicate start nodes for fault propagation so SEARCH does not waste time working on a start node that failed propagation once. Old SCIRTSS did not remove duplicate start nodes and therefore, if an error occurred on a start node, SEARCH will work on the next start node specified by BRIDGE which could be a duplicate of the first one and SEARCH will get the same error on that start node, a waste of effort.

For initial input sequence specification, old SCIRTSS is cumbersome. New SCIRTSS accepts initial input sequences the old way and also like the communication section of the U-AHPL simulator HPSIM4.

SCIRTSS 3.0 is written in CDC FORTRAN and Assembly language and depends on its 60-bit word length. SCIRTSS 4.0 CONTROL PROGRAM, BRIDGE, and SEARCH are written in RATFOR, FLTSIM and NDALG are written in VAX FORTRAN. Since VAX has a 32-bit word length, new SCIRTSS is more portable than the old one, even though a small number of functions in FLTSIM are written in VAX Assembly language.
Old SCIRTSS runs in batch mode only. New SCIRTSS runs in batch and interactive modes. In interactive mode, SCIRTSS 4.0 is very user friendly. The user is able to decide on every loop of SCIRTSS whether to continue running SCIRTSS or not. The user also has the option of changing some of the parameters and debug print flags on every loop of SCIRTSS. SCIRTSS results are printed and displayed with symbol names used in the U-AHPL description so that the user does not have to go back to the U-AHPL description to refer to symbol names. The symbol names of flip-flops, inputs and outputs are displayed with their respective values.

A list of SCIRTSS 4.0 extensions to and advantages over SCIRTSS 3.0 are summarized in table Table 6.1.
Table 6.1: Extensions to Previous SCIRTSS

<table>
<thead>
<tr>
<th>Item</th>
<th>SCIRTSS 3.0</th>
<th>SCIRTSS 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of flip-flops</td>
<td>60</td>
<td>999</td>
</tr>
<tr>
<td>Maximum number of gates</td>
<td>500</td>
<td>10000</td>
</tr>
<tr>
<td>Maximum number of inputs</td>
<td>60</td>
<td>200</td>
</tr>
<tr>
<td>Maximum number of outputs</td>
<td>60</td>
<td>200</td>
</tr>
<tr>
<td>Maximum number of modules</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Combination Logic Units</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No. of Active CSs Allowed</td>
<td>Single</td>
<td>Multiple</td>
</tr>
<tr>
<td>Reduction of Duplicate Start Nodes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Search Functional Simulator</td>
<td>Inaccurate</td>
<td>Accurate</td>
</tr>
<tr>
<td>Interactive Mode</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>User Friendly Input/Output</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>System Portability</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
</tbody>
</table>
SCIRTSS user interface is mainly a question and answer type. The user is queried for information and decisions. Whenever possible, for an incorrect response the user is asked the query again while giving appropriate messages on the incorrect response. The SCIRTSS parameters are divided into two sections, a general parameter section and second-level optional parameters section. The user can also optionally specify SEARCH guiding input vectors interactively or in a file. The user has the option of defining his own heuristic component for guiding SEARCH via a user-defined heuristic subroutine.

7.1 General Parameters

SCIRTSS general parameters must be specified by the user in the input file. The general parameters include such parameters as the time limit for SCIRTSS and FLTSIM and SEARCH parameters. Heuristic weights and the initial input sequence are also specified in this file. The initial input sequence can also, optionally, be specified using a communication section file. See SCIRTSS User's Manual for a detailed description of the general parameters and the
parameters of the communication section file. The general parameters file contains:

Line 1: Title
Line 2: The flag of good or fault simulation
Line 3: Optional switch(I)
Line 4: Heuristic weights
Line 5: Search parameters
Line 6: Priority classes
Line 7: Initial values of data flip-flops
Line 8: Simulation parameters
Line 9: Initial input sequence

7.2 Second Level Parameters

As the name implies, the second level parameters are optional and the user does not need to specify them at all. If, however, the user wants to override the default values of any or all of the second level parameters, he must specify values for all of them in a file and use that file for specifying them. Second level parameters include debug print flags and fault selection order and other flags. The second level parameter file contains:

Line 1: Debug Mode Flag
Line 2: Print Maps Mode
Line 3: Print NRY(1:MARKA)
Line 4: Print Input Sequence (NRY)
Line 5: Bridge Debug Mode
Line 6: Print New Faults Debug Mode
Line 7: Print Bit Nodes Debug Mode
Line 8: Print BRIDGE Results Flag
Line 9: Print Good Network Final State Check
Line 10: Flip-Flop Fault Selection Order
Line 11: Print SEARCH Intermediate Nodes
Line 12: Print SEARCH Final Sequence Nodes
Line 13: Fault Gone Flag
Line 14: User Heuristics Subroutine Flag
Line 15: SEARCH Functional Simulator Ordering Flag
Line 16: Print SEARCH error SETA flag


7.3 SEARCH Guiding Input Vectors

For SEARCH guiding input vectors, SCIRTSS automatically generates control input vectors from the U-AHPL executable tables [9]. The user can optionally specify data input vectors (D.I.V.) for guiding SEARCH. Once the user has entered the D.I.V.s interactively, they can be saved in a file for later runs. Next time, the user needs to specify that file name for reading in D.I.V.s and does not have to enter them manually, unless of course the user wants to specify different D.I.V.s. Refer to the SCIRTSS User's Manual for a detailed description of entering D.I.V.s.
7.4 Interactive Interface

On every loop of SCIRTSS, the user is asked if he wishes to stop or continue and also if he wishes to change any parameter values. The changing of the parameters is menu-driven. The user is presented with the main SCIRTSS Debug Parameters Menu. From this menu the user can select one of four other menus or return to SCIRTSS. The four menus are:

- Executive (CONTROL PROGRAM) Debug Menu
- FLTSIM Debug Menu
- BRIDGE Debug Menu
- SEARCH Debug Menu

Each of these menus displays a selection list of debug parameter names and their present values. The user can select from this menu a parameter to be changed and will be asked to specify the new value or null for no change. After this the same menu is displayed again with the new value for the parameter selected. The user can repeatedly change parameters in this menu or return to main menu for selecting other functional area menus or return directly to SCIRTSS. See SCIRTSS User's Manual for a detailed description of each menu and its associated parameters.

7.5 User Defined Heuristic Component

The user-supplied heuristic subroutine, HEUSUB, helps in directing the search towards the goal node. This
could be done by controlling the heuristic value of the successor node. Therefore some useful information about the successor node is available for the user to enable the writing of an effective HEUSUB routine. The meanings of the variables passed in the HEUCOM are:

- **NGPRED** The number of predecessor good control states.
- **PRDGSC** The array of predecessor good control states.
- **NFPRED** The number of predecessor faulty control states.
- **PRDFCS** The array of predecessor faulty control states.
- **NGSUCS** The number of successor good control states.
- **SUCGCS** The array of successor good control states.
- **NFSUCS** The number of successor faulty control states.
- **SUCFCS** The array of successor faulty control states.
- **UHEURF** The initial successor node heuristic value. The new value should also be stored this variable.
- **USCHMD** The search mode.
- **UNDLVL** The successor node level in the graph.
- **SUCXIN** The applied external inputs of the node. Bytes allocated as declared. MSB in a byte is the MSB of the external input.
- **SUCMEM** The successor node data memory byte array. Column 1 and 3 are the good and faulty known values. Column 2 and 4 are the good and faulty unknown values. Bit zero in a byte is the least significant memory bit.
7.6 Design Rules in SCIRTSS

There are three design rules in SCIRTSS which the user should follow in designing with U-AHPL. These rules must be met before the user can run SCIRTSS. These rules govern U-AHPL control function expressions, bidirectional buses (EXBUSES) and the maximum circuit size allowed.

7.6.1 U-AHPL Control Functions

In the following, a list of permitted expressions for the condition functions of the branch statement are listed.
1. All the -ELEMENT statements.
2. The AND and OR reduction operators and the nesting of such operators.
3. Expressions with a product of sum forms.
The branch control expression should simplify to a sum of products expression for the control input vectors.

It is expected that the user would avoid trivial expressions such as (A\&-A). The user should also use sum-of-products expressions such as (-A\&-B) rather than -(A+B). The user should insure that the memory elements used in the conditional branches are not initialized to an unknown value (U), since SEARCH will ignore a successor node that has an unknown control state if a U propagates to such a memory. Refer to SCIRTSS User's Guide [14] for a detailed description of U-AHPL control functions.
7.6.2 Bidirectional Bus Considerations

Since SCIRTS simulators do not model tri-state values, SCIRTS does not directly support bi-directional buses. But bidirectional buses (EXBUSES) can be modeled using EXINPUTS, BUSES, and EXOUTPUTS facilities in U-AHPL. The EXOUTPUTS facility is assigned the value of the BUSES facility after ENDSEQUENCE. The EXINPUT facility is used on the right-hand side of the expressions and the BUSES facility is used on the left-hand side of the expressions in the control states. For a detailed description of bidirectional bus considerations refer to SCIRTS User's Guide [14].

7.6.3 Circuit Size

The user should insure that the maximum number of flip-flops in the circuit (all memory and control flip-flops of all modules) does not exceed 999. The maximum number of gates allowed in a circuit is 10,000, and the maximum number of external inputs and outputs is 200 each. The maximum number of modules allowed in a circuit is ten.

Even if the user's circuit does not exceed the above circuit limits, it is possible for SCIRTS to reach its internal storage limit. For example, to simulate a circuit with the maximum allowed size, FLTSIM alone requires \((2 \times 999) + (6 \times 10,000)\) = 61,998 bits to represent all the flip-flop and gate faults assuming that the circuit is made up of 10,000 two-input gates. Each flip-flop has two
faults (output stuck-at-zero and stuck-at-one) and each two-input gate has six faults (each input stuck-at-zero and stuck-at-one and gate output stuck-at-zero and stuck-at-one). The 61,998 bits require 3,784 32-bit words of storage to describe a three-valued faulty state of a single memory element. To describe the faulty state of the entire circuit (all flip-flops) for parallel fault simulation, 3.78 million words are required. If FLTSIM saves multiple faulty states during the fault propagation mode to remove any useless trailing input sequences, it is obvious that the SCIRTSS system would require a huge amount of storage.
CHAPTER 8

EXAMPLES

The results of running the three circuits described in this chapter illustrate the capability of SCIRTSS to handle large and complex sequential circuits efficiently. The three circuits considered are the BYTADDR circuit, the DUPCHK circuit, and the MDPCHK circuit.

8.1 BYTADDR Circuit

The circuit shown in Figure 8.1 is a byte processing circuit. It reads from or writes to a 16X16 random access memory that is embedded in the circuit. The combinational logic units (CLU) used in this circuit are described in Figure 8.2. This is a relatively large circuit with 585 gates and 283 flip-flops (see Table 8.1), 256 of which are contributed by the RAM. The circuit contains a 4-bit counter, a traditionally hard to test function. To sensitize the faults in the MSB of the counter, the counter must be allowed to count up to that bit pattern, requiring a large number of input sequences to test those faults. Since random input sequences usually detect the easy faults, a random input sequence of 50 clocks was applied, which detected 55 percent of the faults. Three more SCIRTSS runs were used to detect the harder faults.
MODULE: BYTADDR.

EXINPUTS: ADDRESS[5]; READ; WRITE; DATAIN[16]; CLOCK.
EXOUTPUTS: DATAOUT[16]; READY.
MEMORY: ADDRREG[5]; DATAREG[16]; M<16>[16].
BUSES: ADDBUS[4]; DBUS[16]; DCDBUS[16].
CLUNITS: DCD[16] <= DCDER[4].

BODY SEQUENCE: CLOCK.

1 ADDBUS=ADDRESS[0:3];
   DATAREG[0:7]*(READ+WRITE) <=
   (DBUS[0:7]!DBUS[8:15]!DATAIN[0:7]) *
   (READ~ADDRESS[4]+WRITE~ADDRESS[4],
   READADDRESS[4],WRITE~ADDRESS[4]);
   DATAREG[8:15]*(READ+WRITE) <=
   (DBUS[8:15]!DATAIN[0:7]!DATAIN[8:15]) *
   (READ,WRITEADDRESS[4],WRITE~ADDRESS[4]);
   ADDRREG[0:3] <= (INC(ADDBUS)!)ADDBUS]*(READ,WRITE);
   ADDRREG[4]<=ADDRESS[4];
   => (READ~WRITE,RED,WRITE)/(1,2,4).
2 DATAOUT=DATAREG; READY=~ADDRREG[4];
   ADDBUS=ADDRREG[0:3];
   => (~ADDRREG[4])/(1).
3 DATAOUT=DATAREG; READY=\1\;
   => (1).
4 ADDBUS=ADDRREG[0:3]; ADDRREG[0:3] <= INC(ADDBUS);
   M*DCDBUS <= DATAREG; READY=~ADDRREG[4];
   DATAREG[0:7] <= DATAIN[8:15];
   => (~ADDRREG[4])/(1).
5 ADDBUS=ADDRREG[0:3];
6 ADDBUS=ADDRREG[0:3];
   M*DCDBUS <= DATAREG; READY=\1\;
   => (1).

ENDSEQUENCE
CONTROLRESET(1);
DCDBUS=DCD(ADDBUS); DBUS=M*DCDBUS.
END.

Figure 8.1: BYTADDR U-AHPL Description
CLU : INCER(X) [I].
INPUTS : X[I].
OUTPUTS : Y[I].
CTERMS : RESULT[I]; CARRY[I-1].

BODY

CARRY[I-2], RESULT[I-1] = X[I-1], ¬X[I-1];
FOR J = I-2 TO 0 CONSTRUCT
RESULT[J] = CARRY[J] ⊕ X[J];
IF J <> 0 THEN CARRY[J-1] = CARRY[J] & X[J] FI
ROF;
Y = RESULT.
END.

CLU : DCDER(IN) [I].
INPUTS : IN[I].
OUTPUTS : OUT[2-I].
CTERMS : RESULT[2-I].

BODY

FOR M = 0 TO (2-I)-1 CONSTRUCT
RESULT[M] = &/TERM(M; IN)
ROF;
OUT = RESULT.
END.

Figure 8.2: BYTADDR CLUs Description

All 32 possible combinations were specified for the 5-bit ADDRESS external inputs for data input vectors to assure that SCIRTSS can access all the locations for reading and writing the RAM. Except for the 16-bit DATAIN external inputs, only the all zeros combination was used as a data input vector to reduce the number of successors generated from control state one. Different data input vectors were applied for the second, third, and the fourth run to detect the rest of the faults in the circuit. In the final two runs, the DATAIN values of all zeros and all ones were used for data
Table 8.1: Summary of SCIRTSS Statistics

<table>
<thead>
<tr>
<th></th>
<th>BYTADDR</th>
<th>DUPCHK</th>
<th>MDPCHK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Gates</td>
<td>585</td>
<td>161</td>
<td>56</td>
</tr>
<tr>
<td>Number of Flip-Flops</td>
<td>283</td>
<td>300</td>
<td>46</td>
</tr>
<tr>
<td>Number of Inputs</td>
<td>23</td>
<td>33</td>
<td>6</td>
</tr>
<tr>
<td>Number of Outputs</td>
<td>17</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Number of Faults</td>
<td>3116</td>
<td>1079</td>
<td>240</td>
</tr>
<tr>
<td>Faults found</td>
<td>3110</td>
<td>1079</td>
<td>238</td>
</tr>
<tr>
<td>% Faults Found</td>
<td>99.8</td>
<td>100</td>
<td>99.2</td>
</tr>
<tr>
<td>Sequence Length</td>
<td>329</td>
<td>486</td>
<td>200</td>
</tr>
</tbody>
</table>

input vectors to detect all the parallel faults and at the same time not specify all possible combinations of DATAIN. SCIRTSS detected 99.8 percent of the detectable faults in the circuit with four runs and an input sequence of 329 clocks. The circuit statistics are given in Table 8.1 and the results are summarized in Table 8.2.

8.2 DUPCHK Circuit

The duplicate character checker circuit, shown in Figure 8.3, is another relatively large circuit with a large number of flip-flops and is highly sequential in nature.
Table 8.2: Summary of SCIRTSS Results

<table>
<thead>
<tr>
<th>Circuit</th>
<th>BYTADDR</th>
<th>DUPCHK</th>
<th>MDPCHK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial input sequence</td>
<td>50</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Manual input sequence</td>
<td>0</td>
<td>392</td>
<td>0</td>
</tr>
<tr>
<td>No. of Subsequences</td>
<td>93</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>No. of Sen. Searches</td>
<td>22</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Faults found by SEARCH</td>
<td>1395</td>
<td>1005</td>
<td>208</td>
</tr>
<tr>
<td>Faults found (Sens.)</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Faults/subsequence</td>
<td>15</td>
<td>201</td>
<td>11</td>
</tr>
<tr>
<td>Faults/Sens. seq.</td>
<td>0</td>
<td>0</td>
<td>0.67</td>
</tr>
<tr>
<td>Faults found last input</td>
<td>924</td>
<td>295</td>
<td>174</td>
</tr>
<tr>
<td>% found last input</td>
<td>66</td>
<td>29</td>
<td>73</td>
</tr>
</tbody>
</table>

As a 32-bit word is read-in, it is compared with the previous eight 32-bit words saved in the embedded RAM in the circuit. With 33 external inputs and two external outputs, a test output TOUT was added to detect some faults otherwise undetectable due to circuit characteristics.

Only data input vectors of all zeros and all ones were applied for 32-bit CHAR external input. Without any initial input sequence in the first run of SCIRTSS, SEARCH detected 92 percent of the faults with an input sequence of 92 clocks. With only five subsequences, all propagation, an
average of 201 faults were detected per subsequence. The second and the third runs were manual input sequences of length 392 combined which detected the rest of the detectable faults in the circuit.
With 300 flip-flops, DUPCHK circuit was used to prove the capability of SCIRSS to handle large circuits. The final example, MDPCHK, was run to compare SCIRSS results with the old SCIRSS.

8.3 MDPCHK Circuit

The MDPCHK circuit, Figure 8.4, is a smaller modified version of the previously-mentioned duplicate character checker. The CHAR size is 4-bits instead of 32-bits, but there is an additional RESET external input. This external input, when activated, sets the circuit to control state one from any other control state, making random input sequences ineffective and more difficult for SEARCH. Since the memory flip-flop Y gets the result of the comparison of characters in control states three and four, and Y is then loaded with the flip-flop ZR in control state five only, it is difficult to get a stored fault to an external output. The results for the 22 subsequences are summarized in Table 8.3. The four SEARCH failures are due to the circuit behavior.

For example, a stored fault effect in flip-flop Y in control state 2 disappears when Y is loaded with a value of zero. The MDPCHK is the same circuit that Carter [5] used as Circuit 8. Comparing Table 5 in Carter [5] and Table 8.3, it is obvious that SEARCH in old SCIRSS was inaccurate in fault propagation and had five SEARCH errors due to the functional simulator inaccuracies. New SCIRSS
Figure 8.4: MDPCHK U-AHPL Description

had four SEARCH failures but all were due to circuit behavior and not an inaccurate functional simulator. New SCIRTSS detected the same percentage faults as the old version with
a smaller number of subsequences, a smaller number of sensitization searches, and a smaller number of input sequence clocks. The comparison is summarized in Table 8.4.

Table 8.3: MDPCHK Circuit Results

<table>
<thead>
<tr>
<th>Subsequence No.</th>
<th>Length</th>
<th>SEARCH Mode</th>
<th>Graph Size</th>
<th>Detected Faults</th>
<th>Last Total</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>FPS</td>
<td>97</td>
<td>63</td>
<td>30</td>
<td>Initial Sequence</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>FPS</td>
<td>107</td>
<td>24</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>FPS</td>
<td>157</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>FPS</td>
<td>107</td>
<td>45</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>FPS</td>
<td>107</td>
<td>8</td>
<td>8</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>FPS</td>
<td>107</td>
<td>0</td>
<td>0</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>FPS</td>
<td>107</td>
<td>0</td>
<td>0</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>FPS</td>
<td>1643</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>FPS</td>
<td>1240</td>
<td>9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>FPS</td>
<td>1075</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>FPS</td>
<td>1095</td>
<td>1</td>
<td>1</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>FPS</td>
<td>1025</td>
<td>1</td>
<td>1</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>FPS</td>
<td>1241</td>
<td>1</td>
<td>1</td>
<td>SEARCH Failure</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>FPS</td>
<td>1155</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>DSS</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>FPS</td>
<td>507</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DSS</td>
<td>748</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>DSS</td>
<td>757</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Where: FPS - Fault Propagation Search

DSS - D-Algorithm Sensitization Search
Table 8.4: New and Old SCIRTSS Runs on MDPCHK

<table>
<thead>
<tr>
<th></th>
<th>New SCIRTSS</th>
<th>Old SCIRTSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Faults Found</td>
<td>99.2</td>
<td>99.2</td>
</tr>
<tr>
<td>Search Failures</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>No. of Subsequences</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>No. of Sen. Searches</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>Sequence Length</td>
<td>200</td>
<td>222</td>
</tr>
</tbody>
</table>
CHAPTER 9

CONCLUSION

This paper developed a fully automatic, user-friendly automatic test generation system. This system employs a heuristic-based SEARCH system that is driven by a hardware compiler. The system is capable of handling VLSI digital integrated circuits with up to a thousand memory elements and up to ten thousand gates. SCIRTSS produces efficient test sequences for large circuits in a reasonable time.

SCIRTSS 4.0 is a more user-friendly, more portable, and more accurate test generation system than its predecessor. The designer is able to avoid testing problems encountered on the manufacturing floor by including comprehensive test generation early in the design cycle with SCIRTSS. There are a couple of improvements and extensions to SCIRTSS for future work that can enhance SCIRTSS even more.

An improvement to SCIRTSS can be made by making the NDALG subroutine more intelligent. For typical U-AHPL circuits with a single active control state at any time, NDALG returns some vectors requiring multiple active control states to sensitize a gate fault. NDALG subroutine could be modified to discard such test vectors while it is in process.
of generating them, depending on a flag set by the user signifying whether the circuit under test is a single active control state circuit or not. Since NDALG generates multiple test vectors, some of which usually require a single active control state, SCIRTSS is not limited by the present NDALG, but NDALG can be made more efficient by not generating test vectors with multiple active control states for a circuit with only single active control state capability.

As the use of MOS technology increases, accurate fault modeling for MOS circuits becomes essential. Since the parallel fault simulator in SCIRTSS models stuck-at-0 and stuck-at-1 faults only, it is not sufficient to test MOS circuits. An accurate MOS fault simulator could be used to verify the effectiveness of SCIRTSS-generated input sequences in testing MOS circuits. To incorporate MOS fault modeling in SCIRTSS, the FLTSIM subroutine would have a major impact since it is the fault simulator program. The NDALG subroutine would not be effected as much as the FLTSIM subroutine since it deals with circuit element function and element values. The SEARCH subroutine would have minimal impact since it employs a technology-independent functional simulator. Since the BRIDGE subroutine must use the results of the FLTSIM subroutine, the fault format conversion subroutines would have to be modified. The fault priority selection subroutines would not be affected. The interactive user interface should not be affected at all.
APPENDIX A

SCIRTSS PROGRAMS

The SCIRTSS hierarchical structure is shown in terms of subroutines calling other subroutines with details for the CONTROL PROGRAM and the BRIDGE subroutines. Also the names of the files that contain the COMMON blocks and the complete listing of all the RATFOR defines used in SCIRTSS are listed.
A-1 SCIRTSS Hierarchical Structure

SCIRTS
  INITIME
  SECOND
  BEGIN
  ETIME
  SECOND
  FLTSIM
  PNRY
  ETIME
  ASK
  CHORDR
  MCHGPA
  BRIDGE
  ETIME
  RMVND
  DUPMOD
  UNMACH
  BRGRSL
  PMODES
  HTOB
  PINSEQ
  SEARCH
  SCHRSL
  PINSEQ
  ETIME
  FLTSIM
  FINISH
  UPDIMP
  CLFIL
  PRESLT
  SECOND
  LSTFLTS
  BINHEX
  CLFIL
BEGIN
  OPFIL
  ASK
  SCRPRM
  DSLPFL
  MCHGPA
  RSTORS
  CLFIL
  UPDSDT
  SVTBPW
  ERRMSG
  BLDMAP
  BLDBMP
  INTLIZ
  ASK
  OPFIL
  INVECT
  DMPDIV
  CLFIL
  CNTFNC
  INITIAL
  TASMBLE
  INTLCS
  ASK
  PMAP

MCHGPA
  ASK
  PMENLN
  GETSEL
  CPMENA
    PMENLN
    GETSEL
    GETNVL
    CHORDR
  CPMENB
    PMENLN
    GETSEL
  CPMENC
    PMENLN
    GETSEL
    GETNVL
  CPMEND
    PMENLN
    GETSEL
    GETNVL
BRIDGE
PMFLT
FINSRT
GETFLT
GENGND
PRLPC
PRMVPC
PNVECT
PVars
COMPS
HTOB
PROPFL
PNFLT
STFLTS
FFFLTS
UNDTGF
FAILFL

STFLTS
PNFLT
RANSLC
CHKSND
LSTBITS
PMFLT
LSTFLTS
GENSTR
LSTBITS
GENFND

FFFLTS
GFFFLT
PNFLT
RANSLC
PNFLT
LSTFLTS
GENFGL
LSTBITS
GENFND
A-2  SCIRTSS COMMON Blocks Files

COMALL.RAT
COMENM.RAT
TEMP.RAT
DIVFLG.RAT
COMPL.RAT
COMBRG.RAT
COMMB.RAT
XBO00.RAT
SHARE.RAT
PNTERS.RAT
PERM.RAT
SCRDAT.RAT
A-3 Listing of Defines

#FILE NAME: DEFNS.RAT

DEFINE(XB023L,40)  #SIZE OF NEST LIST
DEFINE(XB023C,08)

DEFINE(XB024L,360) #SIZE OF CEQ & CEQREF LISTS
DEFINE(XB024W,1080)
DEFINE(XB024C,03)

DEFINE(XB025L,300) #SIZE OF ORT1 LIST
DEFINE(XB025W,600)
DEFINE(XB025C,02)

# INTEGER CONSTANTS
DEFINE(WC0002,02)
DEFINE(WC0004,04)
DEFINE(WC0008,08)
DEFINE(WC0010,10)
DEFINE(WC0015,15)
DEFINE(WC0020,20)
DEFINE(WC0025,25)
DEFINE(WC0032,32)  #NO. OF BITS/COMPUTER WORD
DEFINE(WC0035,35)
DEFINE(WC0050,50)
DEFINE(WC10E3,1000)
DEFINE(WC10E4,10000)
DEFINE(WC10E5,100000)
DEFINE(WC10E6,1000000)

# STAGE 1 TABLES
DEFINE(XB001L,150) #SIZE OF SDT LIST
DEFINE(XB001W,23)  #NUMBER OF WORDS
DEFINE(XB001C,10)  #NUMBER OF COLUMNS

DEFINE(XB002L,60)  #SIZE OF LRT LIST
DEFINE(XB002W,05)
DEFINE(XB002C,03)

DEFINE(XB003L,150) #SIZE OF SRT LIST
DEFINE(XB003W,35)
DEFINE(XB003C,09)

DEFINE(XB004L,250) #SIZE OF QTABLE LIST
DEFINE(XB004W,65)
DEFINE(XB004C,04)

DEFINE(XB005L,50)  #SIZE OF SQRT LIST
DEFINE(XB005W, 17)
DEFINE(XB005C, 06)

DEFINE(XB007L, 400) #SIZE OF TOTS LIST
DEFINE(XB007W, 107)
DEFINE(XB007C, 04)

DEFINE(XB013L, 20) #SIZE OF SYSTAB LIST
DEFINE(XB013W, 04)
DEFINE(XB013C, 13)

DEFINE(XB014L, 148) #SIZE OF THUNK LIST
DEFINE(XB014W, 147)
DEFINE(XB014C, 04)

DEFINE(XB015L, 108) #SIZE OF PARAM LIST
DEFINE(XB015W, 107)
DEFINE(XB015C, 02)

DEFINE(XB016L, 108) #SIZE OF ARG LIST
DEFINE(XB016W, 107)
DEFINE(XB016C, 01)

DEFINE(XB018L, 110) #SIZE OF PINTAB LIST
DEFINE(XB018W, 11)
DEFINE(XB018C, 01)

DEFINE(XB019L, 50) #SIZE OF REF LIST
DEFINE(XB019W, 23)
DEFINE(XB019C, 07)

DEFINE(XB020L, 50) #SIZE OF FOR LIST
DEFINE(XB020W, 23)
DEFINE(XB020C, 06)

DEFINE(XB021L, 50) #SIZE OF IF LIST
DEFINE(XB021W, 23)
DEFINE(XB021C, 07)

DEFINE(XB022L, 20) #SIZE OF PULSE LIST
DEFINE(XB022W, 05)
DEFINE(XB022C, 01)

DEFINE(XB030L, 150) #TRUE SIZE OF SYMBOL TABLE
DEFINE(XB030W, 600) #TOTAL WORDS = 150*4
DEFINE(XB030C, 04)

DEFINE(XBPW, 8)
DEFINE(MODTYP, 81)
DEFINE(CLUTY1, 641)
DEFINE(CLUTY2, 642)
DEFINE(FNRTY1, 781)
DEFINE(FNRTY2, 782)
DEFINE(CLUDEC, 151)
DEFINE(FNRDEC, 152)
DEFINE(MEMORY, 125)
DEFINE(EXBUS, 122)
DEFINE(EXINPU, 123)
DEFINE(INPUT, 124)
DEFINE(OUTPUT, 126)
DEFINE(BUS, 121)
DEFINE(LABEL, 116)
DEFINE(EXOUT, 128)
DEFINE(XXX, 999)

# SEARCH CONSTANTS AND DIMENSIONS
#
DEFINE(FPROMOD, 0)  #FLT PROP. MODE.
DEFINE(EMBRMOD, 1)  #EMB. SENS. MODE.
DEFINE(DALGMOD, 2)  #D-ALG SEN. MODE.
DEFINE(MAXBND, 2000)  #MAX. # OF SETB NODES
DEFINE(MAXAND, 2000)  #MAX. # OF SETA NODES
DEFINE(MAXSTB, 2000000)  #MAX. SETB STORAGE POOL
DEFINE(MAXSTA, 2000000)  #MAX. SETA STORAGE POOL
DEFINE(MAXLVL, 1023)  #MAX. SEARCH GRAPH LEVEL
DEFINE(MAXNODS, 10)  #MAX. # OF NODES
DEFINE(MAXGN, 1410)  #MAX. WORDS/GOOD NETWORK NODE
DEFINE(MAXFN, 14100)  #(MAXGN * MAXNODS) =
# MAX WORDS/FAULTY
# NETWORK NODE

DEFINE(DUMMY, 9999)
DEFINE(SVTSIZ, 2500)  #SIZE OF SIM. OSVT & NSVT ARYS
DEFINE(MAXMOD, 10)  #MAX # OF UAHPL MODS. IN SCIRTSS
DEFINE(MAXCS, 200)  #MAX # OF CONTROL FFS
DEFINE(MAXMEM, 1000)  #MAX # OF MEMORY FFS
DEFINE(MAXINF, 200)  #MAX # OF EXTERNAL INPUTS
DEFINE(MAXOUT, 200)  #MAX # OF OUTPUTS
DEFINE(MAXACS, 5)  #MAX # OF ACTIVE C.S. BRIDGE!!
DEFINE(MAXSYM, 20000)  #MAX # OF DATA INP. VEC. LIST
DEFINE(MAXCHAR, 10)  #MAX # OF DIV. CHARACTER ARRAY
DEFINE(DIVBY, 6000)  #MAX # OF BYTES OF D.I.V.
DEFINE(MAXBYT, 14000)  #MAX SIZE OF BITOBYT ARRAY

# BRIDGE DEFINES
#
DEFINE(MAXNRY, 1000000)  # MAX NRY SIZE
DEFINE(MAXPC, 6)  #MAX PRIORITY CLASS
DEFINE(HDALTLT, 40)  #MAX DALG TIME LIMIT IN SECONDS
DEFINE(LDALTLT, 10)  #MIN DALG TIME LIMIT IN SECONDS
DEFINE(EDALT, 4)  #ERR DALG TESTS REQUESTED
DEFINE(MBUPLPS, 10)  #MAX BRIDGE UNPRODUCTIVE LOOPS
DEFINE(MAXFF,999) # MAX # OF FFS
DEFINE(MAXGATES,9999) # MAX # OF GATES
DEFINE(MAXEXIS,200) # MAX # OF EXTERNAL INPUTS
DEFINE(EXCLOCK,11000) # EXTERNAL CLOCK INPUT
DEFINE(FEXIN,11001) # FIRST EXTERNAL INPUT #
DEFINE(LEXIN,11200) # LAST EXTERNAL INPUT #
DEFINE(FEXOUT,11201) # FIRST EXTERNAL OUTPUT #
DEFINE(LEXOUT,11400) # LAST EXTERNAL OUTPUT #
DEFINE(MAXCNR,2000000) # MAX SIZE OF CNR ARRAY
DEFINE(MAXSKDS,10) # MAX # OF START NODES
DEFINE(MAXGNDS,10) # MAX # OF GOAL NODES
DEFINE(MFPIPO,1400) # MAXFF + MAXEXIS + MAXOUT

DEFINE(CHKCSX,1) # CHECK FOR CS VALUE UNKNOWN
DEFINE(NCHKCSX,0) # DON'T CHECK FOR CS VALUE UNKNOWN

DEFINE(MAXPCORD,12) # MAX P/C ORDER ARRAY SIZE

DEFINE(SWITCH4,4) # SWITCH 4 FLAG

# NODES DEFINES
DEFINE(BRSL,0) # PRINT BRIDGE RESULTS MODE
DEFINE(SSEN,1) # PRINT SEARCH SEN. SETA MODE
DEFINE(SPROP,2) # PRINT SEARCH PROP. SETA MODE
DEFINE(SESSEN,3) # PRINT SEARCH ERROR SEN. SETA MODE
DEFINE(SESEP,4) # PRINT SEARCH ERROR PROP. SETA MODE
DEFINE(MAXCOLS,10) # MAX # OF COLUMNS TO PRINT IN NODE
DEFINE(MAXROWS,100) # MAX # OF ROWS OF MAXCOLS TO PRINT
DEFINE(MAXLEN,100) # MAX LINE LEN FOR CA ARRAY

#
REFERENCES


