M83/27
151 pages

VICTOR: GLOBAL REDUNDANCY IDENTIFICATION AND TEST GENERATION

by

I. Ratiu

Memorandum No. UCB/ERL M83/27

9 May 1983

(cover)

VICTOR: GLOBAL REDUNDANCY IDENTIFICATION AND TEST GENERATION

by

Ion Ratiu

Memorandum No. UCB/ERL M83/27

9 May 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

VICTOR: Global Redundancy Identification and Test Generation for VLSI Circuits

Doctor of Philosophy | Ion Mircea Ratiu | Electrical Engineering and Computer Sciences

Abstract

During the generation of tests for a digital circuit, most of the effort is spent on just a few faults. For some of these faults, even an exhaustive search cannot find a test, because no test exists; the fault is redundant, and the effort has been wasted. For some others, a test can be found only after much computation, but most test generation procedures allocate fixed resources —computer time and memory —per fault and may stop the computation before a test is found. These hard-to-test, but detectable faults are likely to be considered undetectable, hence lumped with the redundant faults, and effort has been wasted again. Therefore, efficient test generation for a digital circuit requires advance knowledge of the redundant and the hard-to-test irredundant faults.

This report describes VICTOR (VLSI Identifier of Controllability, Testability, Observability, and Redundancy), a linear complexity method for global redundancy identification and test generation for scan-testable VLSI circuits. In four passes through the circuit fault list, VICTOR identifies all redundant and hard-to-test irredundant faults in a general combinational circuit and generates test vectors for most irredundant faults, which are then collapsed and the corresponding test vectors are compacted. The complexity of the algorithm and of the data structure grows linearly with circuit size and primary input count. Several circuit examples are analyzed to illustrate the operations in the algorithm.

The program implementation of VICTOR consists of approximately 4300 lines of ANSI FORTRAN 77.

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to his research advisor, Prof. Donald O. Pederson, for guidance and support, and to Constantin C. Timoc for countless discussions and expert technical hints in the area of testing throughout the development of this dissertation. He also gratefully acknowledges advice and suggestions from Paul Bardell (IBM), Thomas Williams (IBM), Kenneth Parker (Hewlett-Packard), Shigeru Takasaki (NEC), Ray Mercer (University of Texas, Austin), Vishwani Agrawal (Bell Laboratories), Erwin Trischler (Siemens), Predrag Kovijanic (Sperry), Alberto Sangiovanni-Vincentelli, and Antony Fan (University of California, Berkeley).

Support for the research presented in this thesis has been received from the CMOS IC Design Department and the Computer-Aided Design Department of Bell Laboratories and is gratefully acknowledged. The author wishes to thank Eric Iwersen, Hermann Gummel, Ajoy Bose, Bernard Murphy, and Wesley Grant for their encouragement and support.

The author thanks his parents, Rodica and Mircea, for infinite patience and understanding during his graduate years.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Automated generation of tests solves the testing problem for small and medium size digital circuits. For large sequential circuits, however, most such techniques are too expensive, since they require vast amounts of data processing and data storage [Goel 81].

Current very-large-scale-integrated (VLSI) circuits comprise upward of 10,000 logic gates. Automated test generation for a sequential circuit of this complexity is simply not feasible. Moreover, most test generation procedures have difficulties handling such complex circuits, even if scan-path techniques [Williams 73] [Funatsu 75] [Eichelberger 77] [Koenemann 79] [Ando 80] [Mercer 81] are employed and the circuit becomes scan-testable as a combinational circuit.

When generating tests for a digital circuit, most of the effort is spent on just a few faults. For some of these faults, even an exhaustive search cannot find a test, because no test exists; the fault is redundant [Breuer 76], and the effort has been wasted. For some others, a test can be found only after much computation, but most test generation procedures allocate fixed resources — computer time and memory — per fault and may stop the computation before a test is found. These hard-to-test, but detectable faults are likely to be considered undetectable, hence lumped with the redundant faults, and effort has been wasted again. Therefore, efficient test generation for a digital circuit requires advance knowledge of the redundant and the hard-to-test irredundant faults.

Several methods for the identification of redundancy in combinational circuits have been published [Yau 71] [Dandapani 74] [Lee 74] [Smith 79] [Si 78]. However, the methods either apply to special classes of circuits only, or are as complex as the fault detection problem itself.

Recently, a set of fast, heuristic approaches for identifying potential testing difficulties in a digital circuit — collectively called testability analysis — have gained much attention. However, testability analysis employs overly simplified models for sequential circuits, hence the testability estimates for sequential circuits are inherently erroneous. To facilitate computations, testability analysis approaches assume input signals to a circuit element to be independent of each other. Unfortunately, this simplifying assumption proves to be an Achilles' heel, since testability analysis consistently identifies redundant faults as testable, irredundant faults, and thus the method defeats its purpose [Ratiu 82] [Agrawal 82].

This report describes VICTOR (VLSI Identifier of Controllability, Testability, Observability, and Redundancy), a *linear complexity method for global redundancy identification and test generation for scan-testable VLSI circuits*. Two theoretical results are presented: signal dependence is equivalent to convergent fanout, and redundancy implies convergent fanout. Based on these theorems, potentially redundant faults are introduced as the set of redundant and hard-to-test irredundant circuit faults, and a method for their identification is described. For the rest of the faults, the irredundant ones, tests are generated, and the test vectors are collapsed and compacted.

Chapter 2 reviews redundancy and testability analysis. First, the basic concepts in digital testing and testing issues specific to VLSI circuits are described. A detailed analysis of redundancy and its identification methods

follows, and testability analysis, a fast, heuristic approach for identifying potential testing difficulties, is introduced. The various testability analysis methods are extensively reviewed.

Chapter 3 establishes the theoretical foundation and the strategy in VICTOR, the proposed method for redundancy identification and test generation. After defining the circuit and fanout node models, combinational circuits and redundant faults are classified based on fanout convergence. An equivalence theorem for convergent fanout and dependent signals, and an implication theorem for redundant faults and convergent fanout are stated and proven. The strategic goal for VICTOR is sketched out as the development of a global, linear estimation tool that relies on the risk of convergence and evaluates the primary input dependencies of every node in the circuit.

Chapter 4 presents the VICTOR algorithm and details its four steps. The VICTOR testability primitives —pattern, risk, and size —are introduced as the primary input dependencies of a circuit node, the risk of convergence for these dependencies, and the number of such dependencies, and the pattern operations — selection and intersection — are defined. Then, circuit levelizing, controllability calculation, observability calculation, and test generation and redundancy identification, are described and illustrated on a small circuit example. An analysis of the algorithm complexity closes the chapter.

Chapter 5 explains the VICTOR program implementation. The program structure, the files attached during program execution, and the structure of the circuit and fault data are described in detail and exemplified. The reason for choosing ANSI FORTRAN 77 and specific language use in program VICTOR towards program portability are presented.

Chapter 6 deals with the evaluation of the performance of VICTOR, i.e., method correctness and program performance. Method correctness is shown on some pathological circuits small enough to be intuitive, and on a 4-bit ALU, which is analyzed in detail. For program performance, run time measurements for program VICTOR are given on an industrial circuit example.

Chapter 7 summarizes the VICTOR approach, shows its strengths and weaknesses, and suggests directions for future research.

Seven appendices are included in this report. A short history of the development at Berkeley of the FORTRAN 77 version of the SANDIA SCOAP testability analysis program is given in Appendix 1. Appendix 2 lists the library of predefined cells. The input data and the VICTOR analysis results for the 74181 4-bit ALU are presented in Appendix 3, and the circuit description of the industrial example analyzed in Chapter 6 is listed in Appendix 4. Appendix 5 contains the data initialization routine for program VICTOR, and Appendix 6 lists the source code for the entire program.

CHAPTER 2

REDUNDANCY AND TESTABILITY ANALYSIS

2.1. INTRODUCTION

When generating tests for a digital network, a disproportionately small fraction of the faults is responsible for most of the test generation effort. At times, an exhaustive search cannot find a test for a fault; no test exists, the fault is redundant, and the effort has been wasted. Some other faults, although detectable, require an inordinate amount of computation during test generation. Since most test generation procedures limit the resources — computer time and memory — spent on detecting a fault, such hard-to-test faults may not be detected, hence they may be lumped together with the redundant faults as "undetectable" faults. Thus, efficient test generation in a digital network requires advance knowledge of redundant and hard-to-test faults, therefore methods for redundancy identification or heuristic methods capable of identifying potential testing difficulties must be used.

This chapter presents some of the basic concepts in digital testing and analyzes the testing issues specific to a VLSI environment. Then, redundancy in digital networks is reviewed, and techniques for redundancy identification in combinational networks are presented. A fast, but approximate method to identify potential testing difficulties, testability analysis, is introduced, and the testability various approaches published in the past ten years are reviewed.

2.2. DIGITAL TESTING OVERVIEW

A comprehensive review of the testing issues is given in the references [Muehldorf 81]. In this section, the concepts, terms, and notations used in digital testing are briefly reviewed, and the complexity of the fault detection problem is presented. Two different solutions to the testing problem, design for testability and testability analysis, are introduced.

2.2.1. Basic Concepts in Digital Testing

The process of detecting and identifying the causes of incorrect circuit operation is called *testing*. When detecting a circuit malfunction, relevant circuit information is processed from "four universes of discourse, arranged in an ascending order toward the user: (1) physical; (2) logical; (3) informational; and (4) external, or user's universe" [Avizienis 82]. Each of these four universes has its own rules and terminology for the undesired event, i.e., the disruption that produces unexpected and unwanted behavior of the system as perceived by the user: *failure* in the physical universe, *fault* in the logical universe, *error* in the informational universe, and *crash* in the user's universe. The simplest universe to deal with is the logical one, since a variable can have only one out of two possible values, 0 or 1, although it reflects a variety of failures and causes sundry errors and crashes. In this logical universe, the undesired event is a fault for which simple rules of Boolean algebra apply. Depending on circuit representation, logic switch or logic gate, the appropriate fault model is either a switch fault or a logic fault.

The circuit model used most often in testing consists of logic gates and signal lines connected via primary inputs and primary outputs to the outside world. In the *stuck* or *stuck-at* fault model, logic gates always operate

correctly, but signal lines to and from the gates may remain fixed unintentionally at a constant logic value (0 or 1). Thus, for an arbitrary signal line K in the circuit, two faults are possible, K stuck-at-0 and K stuck-at-1, with the notation K/0 and K/1, respectively. A widely used model in testing is the *single-stuck* fault model, in which the circuit contains at most one stuck-at fault. In the remainder of this report, the single-stuck fault model is employed.

A *test* for a fault in a circuit is a sequence of logic 0 and 1 values applied to the primary inputs that cause at least one erroneous primary output value. The set of logic 0 and 1 values applied at the same time makes up a test pattern or a test vector; a test usually comprises several patterns. Two faults are said to be *equivalent* if any test which detects one of them detects the other one as well, and no test distinguishes between the two. If no test exists for a fault, the fault and the circuit to which it belongs to are called *redundant*. If a test exists for every fault in a circuit, both the faults and the circuit are called *irredundant*.

A widely used measure of test generation proficiency is the test set *fault coverage*, which gives the percentage of detected faults out of all faults in the circuit. Due to undetectable faults, the highest attainable fault coverage for a redundant circuit is always less than 100%. Since the number of possible faults depends exclusively on network topology, fault coverage does not indicate how much of the irredundant part of a circuit has been tested if the circuit is redundant.

A test for a specific fault detects many other faults on its path from the primary inputs to the primary outputs, hence the effect of a test on the faults of a circuit must be evaluated. The procedure, called *fault simulation*, also

aids in finding tests for isolating the fault in the circuit *(fault location)* and in tracking down a fault based on specific erroneous output values *(fault diagnosis)*.

2.2.2. Complexity of the Test Problem

Given an arbitrary combinational circuit with a total of p gates and primary inputs, does there exist a test generation algorithm that can compute a test for any detectable fault in p^r operations, where r is a finite constant? If no such algorithm exists, the problem is *NP-complete*.

By linking the single fault detection problem to classic combinatorial problems in complexity theory e.g, the traveling salesman problem, Ibarra and Sahni [Ibarra 75] prove that the testing problem of combinational circuits is NP-complete. Moreover, identifying single redundant faults in a combinational circuit is an NP-complete problem also. To compound issues even further, most useful circuits are not combinational, but sequential in nature, since they contain feedback loops, hence are finite state machines. As is shown later in this chapter, the complexity of the testing problem for sequential circuits is an exponential function of the testing problem for combinational circuits.

The test problem is not NP-complete for any combinational circuit. For instance, ripple-carry adders and decoders, which are known to be easily testable, and other 2-level monotone/unate circuits are testable in polynomial time [Fujiwara 82].

A frontal attack on the testing problem using automated test generation (ATG) and fault simulation has worked well in the past yet has become prohibitively expensive or infeasible for current circuit complexity. Goel reports

more than 23 hours of CPU time on a 370/168 computer system for the test generation and fault simulation of a 50,000 gate circuit [Goel 81].

2.2.3. Design for Testability and Testability Analysis

As an alternative to the traditional approach of testing after design completion, *design for testability* addresses the testing problem *during* design by building testability into the circuit by design. The approach resembles preventive medicine; it does not represent a panacea, but it may shrink the test problem to a manageable size. The price paid for testability by design is usually additional hardware and sometimes lower performance. Hence, design constraints must be weighed carefully before deciding on any design for testability technique. Three well-known techniques published extensively in the past are: level-sensitive scan design (LSSD) [Eichelberger 77], signature analysis (SA) [Frohwerk 77], and built-in logic block observation (BILBO) [Koenemann 79]. For a review of design for testability techniques, see a paper by Williams and Parker [Williams 82].

Different from design for testability, *testability analysis* is a fast method for approximating the difficulty in detecting circuit faults before generating the test patterns. The resulting information serves as a guide for circuit redesign for testability and as a good starting point for test generation and fault simulation. A later section in this chapter presents reviews the various testability analysis approaches.

2.3. VLSI TESTING ISSUES

In addition to the problems in testing large digital networks, new challenges arise for the testing of very large scale integrated (VLSI) circuits. Con-

straints imposed by the VLSI environment require the use of different fault models, yet the traditional single-stuck fault model can be still used. Due to the lack of feasible methods and tools, the testing of sequential VLSI circuits is extremely difficult. Therefore, design techniques that allow sequential circuits to be tested as combinational ones have gained acceptance. With this assumption, the testing of VLSI can be regarded as the problem of testing very large but purely combinational networks.

2.3.1. The VLSI Environment Constraints

The essence of the constraints imposed by the very-large-scale-integrated (VLSI) environment to design and testing alike is chip complexity. Integrated circuits with half a million devices or ten thousand logic gates on a silicon chip are currently manufactured. If chip density continues to grow at the same pace as during the past decade, it will double every year or year and a half, and so will the associated testing difficulties.

Several problems arise from this steadily growing chip complexity. First, how does one access tens of thousands of gates through a number of pins typically limited to less than hundred? Even exotic packaging techniques achieve less than two hundred external input/output connections per chip [Collins 82]; therefore, information flow through a chip resembles two funnels joined mouth to mouth: a few dozen input pins — thousands of internal signal lines — a few dozen output pins. Second, are most signals on a chip still independent of each other? As is shown in Chapter 3, signal convergences imply dependent signals; hence, the double-funnel effect causes many such signals dependencies. Third, do finite-state machines still have a number of states that can be considered practically finite? Out of the thousands of pos-

sible states of current 16 or 32 bit computer architectures [Blume 83], only a few states are assigned for system operation. Most states should never be reached during normal operation, thus the risk for unassigned states and illegal operation codes to occur increases with chip complexity.

2.3.2. The Need for New Fault Models in VLSI

New fault models are required for VLSI because of developments specific to the VLSI mainstream technology, MOS (Metal-Oxide-Semiconductor):

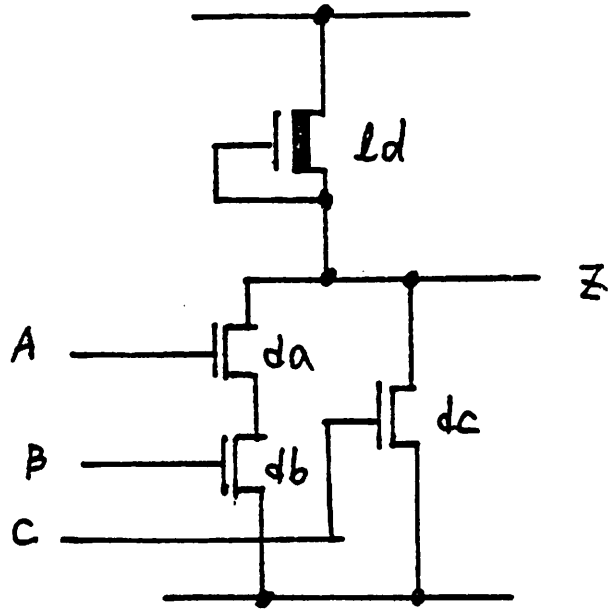
(1) merged MOS logic does not map into the traditional elementary logic gate representation (i.e., AND, OR, NOT, NAND, NOR), and

(2) unconventional failure modes have become statistically significant for MOS technologies.

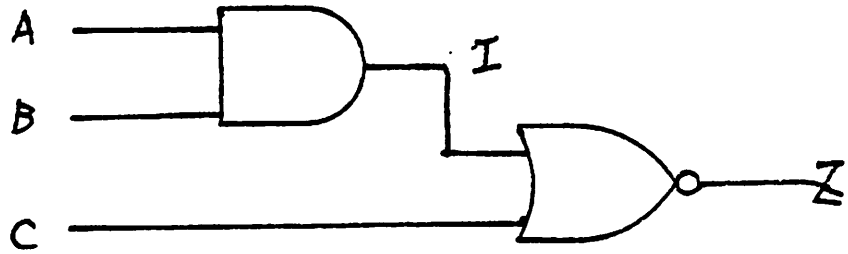
To illustrate the first argument, Figure 2.1 depicts a merged MOS logic circuit and an elementary AND/NOR gate level circuit, both implementing the same logic function

$$Z = (AB + C)'.$$

In the MOS circuit of Figure 2.1(a), the drivers *da* and *db* share the load *ld* with the driver *dc*, which makes it impossible to assign *ld* to either just the AND gate or just the NOR gate of Figure 2.1(b). Also, node I in the gate circuit lacks an homologue in the MOS circuit; hence, stuck-at faults do not model the physical MOS circuit failures. Although both circuits implement the same function, any attempt to establish a structural analogy between the two is contrived, since the gate circuit lacks enough resolution. For the required level of detail, a complete fault analysis of such merged logic must resort to device models instead of gate models [Hayes 82] [Bose 82].



(a)



(b)

Figure 2.1 AND-OR-INVERT Cell z=(ab+c)'.
(a) NMOS circuit implementation.
(b) Logic gate representation

Several failure modes that have been ignored in the past without affecting the chip fault coverage are prevalent in today's silicon implementations. The risk of multiple failures in a half-a-million device chip is real, cannot be neglected, and can only increase with chip density. With device geometries scaling down, in due time, crosstalk among "electronically adja-

cent" circuit parts is likely to cause the same pattern sensitivity faults that plague high-density memory chips. Unfortunately, established memory testing techniques dealing with this aspect, e.g., GALPAT, do not apply directly to general-purpose circuit chips [Breuer 76].

A prevalent cause of circuit failure in CMOS is a transistor open or short circuit. The corresponding fault models, stuck-off and stuck-on, cannot be handled easily by traditional testing approaches; a stuck-at fault model requires almost ten gates [Wadsack 78], whereas the global current-sensing technique [Levi 81] only detects the presence of the failure and not its location. Moreover, stuck-on and stuck-off faults cause sequential circuit behavior due to the charging time of the stray output line capacitance. Techniques for their detection exist though: two vectors are assigned per fault, with the first one applied such that charge/discharge delays are controlled to a known state, and the second one applied for fault detection [Timoc 82]. A test set capable of detecting all stuck-off and stuck-on faults is also guaranteed to detect all single stuck-at faults.

2.3.3. The Viability of the Single-Stuck Fault Model

Is the single-stuck fault model of any use in VLSI? Experimental data on field reject rates shows that if 90-95% of the single-stuck faults are detected, then most other fault types are detected as well. (Exceptions are faults that cause sequential behavior, such as bridging faults and CMOS stuck-on and stuck off faults.) The stuck-at fault model is based on Boolean algebra, therefore it has a simple structure, is computationally efficient, and is of general use. Finally, for a combinational circuit with q nodes, only $2q$ single-stuck faults are possible (K/0 and K/1 for each arbitrary node K), which implies

linear model complexity. If multiple stuck faults are analyzed, the complexity is an exponential function ($3^g - 1$) of the circuit size [Hayes 71].

Recent findings justify the choice of the single-stuck fault model. Most failures for cascode emitter-coupled logic (CECL) and bipolar circuits can be modeled as stuck-at faults [Beh 82]. Carter identifies certain techniques for single-stuck fault detection that find a high percentage of the multiple stuck faults [Carter 82].

2.3.4. The Testing of Sequential VLSI Circuits

In a widely used testing model for sequential circuits, the circuit is transformed into an equivalent set of combinational circuits for which tests are generated [Breuer 76]. For this transformation, the sequential circuit is represented using the Huffman model, which consists of a combinational circuit block, a feedback path, and a register in the feedback path. The equivalent combinational circuit is obtained by breaking up the feedback path and replicating the combinational block and the register for each state of the finite state machine that the sequential circuit implements. Of course, a k-bit register generates 2^k states, hence the complexity of the equivalent combinational model and of its accompanying computations is exponential compared to the complexity of the initial combinational block.

In spite of powerful computer aids for testing, it is unlikely that an ATG will achieve a reliable fault coverage higher than 90% for complex sequential circuits exceeding 5000 logic gates [Bottorff 80]. Moreover, a study [Jensen 82] of the commercially available ATG programs (LASAR, TEGAS-5, and HILO-2) finds that highly sequential networks, even for small circuits, require extensive computer resources.

2.3.5. The Level-Sensitive Scan Design Technique

A solution which eliminates the sequential testing problem altogether is to design the circuit such that all machine states can be easily controlled or observed, i.e., easily set or checked by breaking up the feedback paths. Out of several such methods published in the past [Williams 73] [Funatsu 75] [Eichelberger 77] [Koenemann 79] [Ando 80] [Mercer 81], the Level-Sensitive Scan Design approach has gained acceptance with many mainframe computer and system manufacturers.

Level-Sensitive Scan Design (LSSD) [Eichelberger 77] is a design for testability technique which allows full combinational testing of a sequential circuit, be it a chip, card, subsystem, or full system, by imposing a set of design rules following two concepts:

(1) all internal storage elements (other than memory) have to function also as shift registers, and

(2) circuit operation must not depend on rise time, fall time, or minimum delay of the separate circuits.

LSSD is a well established technique by now. Over the years, it has influenced fault diagnosis methods [Arzoumanian 81] and automatic test generation procedures for a variety of environments: logic masterslices [Lowden 79], LSI chips and printed circuit boards [Bottorff 79], multiple chip VLSI packages [Goel 82b], and large systems [Bottorff 77 & 81].

The LSSD approach is undergoing steady development; recently, a low overhead variation of LSSD particularly suited to VLSI chips has been reported [DasGupta 82]. Also, in spite of the associated overhead, industry acceptance of LSSD and of similar scan design techniques is growing.

2.4. REDUNDANCY IN DIGITAL NETWORKS

A concept related to fault testing is redundancy, the property of a system to operate correctly if part of it is deleted. In this section, redundancy is defined, the reasons for the use of redundancy in system and logic design are given, and some of the proposed techniques to identify single redundancies in a logic network are reviewed. None of these techniques is feasible for the identification of all redundancies in a general combinational circuit.

2.4.1. Definition of Redundancy

The IEEE Standard Dictionary of Electrical and Electronics Terms lists four definitions of redundancy according to the its meanings in different fields: information theory, transmission of information, power systems, and reliability. The broadest one, the definition for reliability, describes redundancy as "the existence of more than one means for performing a given function." This definition, applied to digital circuits and logic functions, is general enough to encompass most other definitions found in literature and is used in the remainder of this report.

2.4.2. The Rationale Behind Redundancy

The chief reason to introduce redundancy in a system is to render it impervious to failure. In order to meet stringent specifications of reliability, availability, and maintainability, designers of military [Bernhard 81], space [Williams 81], and communication systems have traditionally employed redundancy.

The low yield problem plaguing high-density memory chips has induced various semiconductor companies to consider redundant design as a way to

increase chip manufacturability. Nowadays, redundancy can be found in a variety of random access and read-only memory chips [McKenny 80] [Kitano 80] [Mano 80]. Moreover, redundancy in a design benefits both the yield and the field reliability of large chips [Cliff 80].

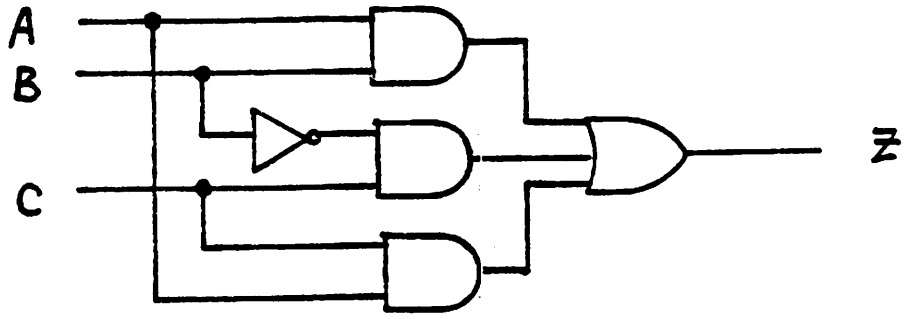
What is the rationale of building in redundancy for an average industrial chip design which does not target fault tolerance or high reliability? As mentioned previously, redundancy represents a safety margin for design, and many practical circuits contain redundancy [To 73]. Also, redundancy can eliminate logic hazards and sometimes simplifies circuit structures [Si 78].

To illustrate the point, the two redundant networks in Figure 2.2 are analyzed. The first one, shown in Figure 2.2(a), implements the logic function

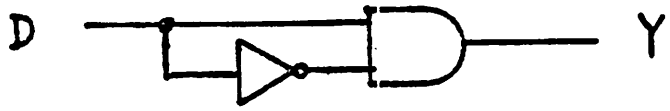
$$Z = AB + B'C + AC$$

in which minterm AC is redundant, i.e., it does not affect function Z. (Note that the input and output stuck-at-0 faults of the lower AND gate are redundant.) If the lower AND gate is deleted, then for A=C=1, any logic change of B causes a static 1-hazard [Breuer 76] at output Z. However, the redundant term AC preserves a logic 1 at Z during any transition on B and thus eliminates the hazard. Eichelberger and Williams apply a similar technique to achieve level-sensitive latches for the LSSD scheme [Eichelberger 77].

The second example, Figure 2.2(b), is a phase splitter feeding a two input AND gate with D and D'. Since the output Y is always 0, fault Y/0 and the two stuck faults at the root of fanout node D, D/0 and D/1, are redundant. Somewhat modified, this scheme is used widely in high-speed circuitry to drive both load and driver devices and therefore shorten the switching time [Blume 83].



(a)



(b)

Figure 2.2 Redundant circuit examples.
(a) Hazard-free implementation
(b) Phase splitter

2.4.3. Methods for Redundancy Identification

By definition, redundant faults in a network cannot be detected. Their identity is usually not known before test generation; hence, much effort (about 90%) is wasted in the futile attempt to generate a test for the redundant faults. Various approaches for redundancy identification (RI) in a combinational network have been proposed, but most deal with particular circuit configurations and cannot be expanded. A brief review of the published work follows.

A simple algorithm to identify redundant variables of a combinational logic function with the goal of achieving a simple realization has been

developed by Yau and Tang [Yau 71]. The method relies on the minterm expansion of the function and some manipulation of the binary and decimal number representation of minterms. No extension of the procedure is given for the identification of redundancies that are embedded in the circuit.

Dandapani and Reddy [Dandapani 74] show that the RI method which considers all combinations of inputs in a general circuit is computationally impractical. Instead, they analyze restricted prime-tree networks, which are specially connected tree networks consisting of AND, OR, and NOT gates. An algorithm to design such prime trees is given, and it is shown that prime trees are easy to test for redundancy.

Lee and Davidson [Lee 74] develop a simple, necessary, and sufficient test for RI in a tree-type NAND network and generalize the result for multi-output NAND, AND-OR, and NOR trees. General, nontree networks are handled by converting them into an equivalent tree form; however, the transformation may map a single connection onto several connections in the equivalent tree form, and the single redundancy test is changed into a much more difficult multiple redundancy test. Moreover, a counter-example to the proposed transformation has been found by Smith [Smith 78 & 79].

Si [Si 78] proposes to use dynamic testing for redundancy identification. His method relies on Clegg's structure and parity-observing output function (SPOOF) [Clegg 73], which he modifies to handle delay information. The technique finds some statically undetectable faults, but does not guarantee the identification of all redundancies and requires vast amounts of memory and computation.

2.5. TESTABILITY ANALYSIS OF DIGITAL NETWORKS

In contrast to the algorithmic redundancy identification methods presented before, a host of simple and fast heuristic methods have been developed that single out the potential testing difficulties in digital networks. These methods, collectively called testability analysis, usually employ simplified circuit and fault models and rely on heuristics. Over thirty papers on testability analysis have been published during the past ten years, as shown in the chronology of publications given in Table 2.1. In the following review, they are grouped into investigative studies, early research work, gate-level analysis, functional-level analysis, hybrid circuit analysis, probabilistic approach, and algebraic approach.

2.5.1. Investigative Studies

Keiner and West [Keiner 77] introduce testability as a subset of maintainability and develop a framework for the derivation of testability measures to

1. [Rutman 72]	17. [Kovijanic 81]
2. [Stephenson 74]	18. [Takasaki 81]
3. [Stephenson 76]	19. [Dunning 81]
4. [Dejka 77]	20. [Longendorfer 81]
5. [Keiner 77]	21. [Akers 82]
6. [Breuer 78]	22. [Bardell 82]
7. [Dussault 78]	23. [Hess 82]
8. [Danner 79]	24. [Fong 82a]
9. [Goldstein 79]	25. [Goel 82a]
10. [Grason 79]	26. [Savir 82]
11. [Kovijanic 79]	27. [Menzel 82]
12. [Longendorfer 79]	28. [Fung 82]
13. [Breuer 79]	29. [Fong 82b]
14. [Goldstein 80]	30. [Agrawal 82]
15. [Bennets 80]	31. [Berg 82]
16. [Susskind 81]	32. [Ratiu 82]

Table 2.1 Chronology of Testability Analysis Publications.

assist the design engineer in producing supportable systems. In their view, testability should be a design parameter instead of a design goal, and an optimum approach to the measurement of testability must rely on the synergism between design for testability techniques and proven measurement capabilities of existing computer-aided design (CAD) tools.

In the quest for a formalized theory of testing, Dejka [Dejka 77] studies the use of circuit complexity as a measure of its testability. Gate count, number of primary inputs, and controllability and observability (as defined for linear sequential machines in control theory) are considered.

Investigating the testability of printed circuit boards, Danner and Consolla [Danner 79] establish a list of 56 testability circuit factors, that range from the use of clocks and functional partitioning of circuits to the content of unusual discrete components and warm-up time. The approach yields board testability ratings that are empirical, but correlate well with experimental results.

2.5.2. Early Work

A heuristic fault measure aiding fault detection in sequential networks is introduced by Rutman. The approach models fault detection as a decision tree corresponding to the path-sensitizing algorithm [Armstrong 66] and relies on the similitude of this tree and the decision trees found in game-playing algorithms. The cost of setting or resetting a circuit node is calculated as the sum of the node costs for the nodes constrained to set or reset the given node; a factor for the level number (see Chapter 4 for circuit levelizing) is then added to the result. Costs for combinational and sequential elements are evaluated alike. The node cost, a positive integer, serves as a measure of

conflict risk in a test generation procedure reminiscent of the D-algorithm [Roth 66]: drive a D into a particular element, drive a D forward, justify a line setting. The process ends when either a test has been computed, a test is judged impossible (the tree has been exhaustively searched), or the program runs out of time or memory space.

The concepts employed by Rutman have proven successful over the years. A sophisticated branching heuristic in the decision tree is largely responsible for the success of the PODEM-X ATG program [Goel 81]. Chess playing algorithms in COPTR, a testability analysis preprocessor for the TEGAS ATG program, speed up the latter by a factor of ten [Kirkland 83].

Stephenson and Grason [Stephenson 74 & 76] introduce the concept of a testability measure and create an independent design tool for large circuits that can be specified at the register-transfer level. A numerical controllability and observability is assigned to every node as a number between zero and one that estimates how easily nodes within the circuit can be controlled from the primary inputs and observed from the primary outputs, respectively. Controllability and observability propagates through a circuit according to a set of combining rules and a transfer factor for each circuit component. The transfer factor of a combinational or sequential component is obtained by lumping its input-output mappings to a single number; a default factor of 0.5 is assumed if no such information exists. The approach relies more on connectivity than function and is computationally simple. Grason [Grason 79] reports a program implementation, TMEAS, that includes a powerful postprocessor for analyzing the generated testability data.

In a proposal for an ATG package for the 80s, Breuer [Breuer 78 & 79] proposes two preprocessing concepts, cost and rate analysis, that should reduce

test generation time by two or three orders of magnitude. The cost analysis uses Rutman's cost function of controlling a node to logic 0 or 1 and applies the same method to calculate the node observability. In the rate analysis, the maximum rate of change on a line, expressed as a sequence of logic 0 and 1 values at the output of a sequential block, is evaluated. Since the rate analysis precludes a wrong choice of the initialization sequence, it reduces substantially the test generation effort for sequential circuits.

2.5.3. Gate-Level Analysis

A gate-level approach based on Breuer's cost and rate analysis, SCOAP [Goldstein 79 & 80] is probably the best-known testability analysis program. SCOAP calculates for each node three combinational measures, 0-controllability, 1-controllability, and observability. For sequential elements, both sequential and combinational measures are computed: the sequential controllability and observability represents the number of time frames required to reach a given node condition, and the combinational controllability and observability represents the number of constrained nodes for each time frame. Simplified sequential models are used, and the different sets of node constraints for each time frame are merged to a single set of constraints. The algorithm handles feedback loops and is at worst quadratic with circuit size. The circuit model consists of cells, logic gates and simple blocks of gates, which must be predefined in a cell library containing a matrix-like binary encoding of the cell controllability and observability equations. The encoding of a cell is a formidable task: seven hundred binary terms for a simple AND-OR-INVERT block have been reported [Trischler 81].

The SCOAP program is publicly available: the FORTRAN 66 original from the SANDIA National Laboratories, and an ANSI FORTRAN 77 version, written by the author, from the University of California at Berkeley (see Appendix 1) Several research papers listed in Table 2.1 stem from SCOAP; the gate-level approaches are described below, while the functional-level approaches are presented later in this chapter.

Hess [Hess 82] and Berg [Berg 82] have taken the Berkeley SCOAP version and, by adding powerful input and output data processing capabilities, have turned it into an industrial-grade testability analysis package for CMOS gate arrays. Menzel [Menzel 82] has proposed and implemented a bidirectional model in SCOAP. Goel [Goel 82a] has started from SCOAP, has enhanced the sequential model considerably by taking into account individual constraints for each time frame, and has build the algorithm upon selective trace; the resulting program yields more accurate sequential values and executes ten times faster.

Another major testability analysis approach based on Breuer's cost and rate analysis is Kovijanic's TESTSCREEN [Kovijanic 79]. Although developed independently, TESTSCREEN and SCOAP are similar, but for a different modeling of the combinational circuits. TESTSCREEN does not automatically increase the controllability when traversing a circuit component, and it takes into account the gate and primary input count. The TESTSCREEN analysis has been expanded to include an estimate of the number of stuck faults and a global testability measure for the whole circuit [Kovijanic 81] [Dunning 81]. The global figure of merit results from a weighted sum of various circuit indicators such as fanout, latches, primary input/output count, gate count, etc..

A final gate-level approach, CAMELOT [Bennets 80], employs the same method of calculating the controllability as TMEAS, but refines the gate-transfer factor and the dependence from the gate inputs by including information about the cell function. Although clocking information is taken into account, CAMELOT encounters computational difficulties with large sequential circuits.

2.5.4. Functional-Level Analysis

Takasaki [Takasaki 81] proposes an approach to functional-level analysis consisting of two steps: a SCOAP evaluation of all functional blocks at the gate level, and the calculation of the functional testability values for the input/output nodes of the functional blocks. The first step consists of a SCOAP analysis for each functional block. Then, controllability and observability values internal to the block are discarded; each block pin is assigned a weighting factor that, together with the number of available pins, enters a norm-like calculation for the functional controllability and observability of the entire block. The analysis results in three testability numbers attached to each pin of the functional blocks.

Recognizing the prevalence of bus architectures in current designs, Fong [Fong 82a & 82b] introduces a data-path controllability and observability along with the usual SCOAP measures. To keep the computations simple, he assumes that for a data path with n branches, the 2^n possible states are uniformly distributed, hence equally likely to occur. The method needs no gate-level representation of the functional blocks.

2.5.5. Hybrid Circuit Analysis

An extension of the testability analysis domain to the device level is hybrid circuit analysis, which uses a hybrid representation composed of logic gates and circuit devices for the circuit.

Longendorfer [Longendorfer 79 & 81] defines a testability measure based on graph theory alone but adjusts the results by empirically penalizing large sequential depth, redundancy, and large circuit blocks. A connectivity matrix of the transistor-level circuit is required to compute the reachability and reaching matrices, which, in turn, generate the desired controllability and observability values.

2.5.6. Probabilistic Approach

Two testability analysis approaches using probability theory have been reported. The first employs information entropy in a chip, while the second develops upon error latency.

Dussault [Dussault 78] sets his testability measure in the domain of information theory and denotes it as the mutual information between the circuit inputs and outputs. Controllability and observability are defined as the inverse of the conditional output/input and, respectively, input/output entropy. In his view, testability analysis should extract as much test data as possible from the circuit, yet should stop short of generating the tests. The algorithm requires much computational effort and memory space. Fung and Fong [Fung 82] expand Dussault's approach to the functional level under assumptions similar to the ones outlined earlier [Fong 82a & 82b].

Bardell [Bardell 82] uses error latency in a combinational circuit as a testability measure. Relying on the initial work on signal probability [Parker 75a

& 75b], he expands the techniques for computing signal probabilities to calculate fault detection probabilities for random pattern testing. Bardell deals with reconvergent fanout in a systematic way and classifies combinational circuits based on reconvergence. However, the approach has difficulties with large circuits because the symbolic data manipulation in the algorithm requires massive computation and data storage.

2.5.7. Algebraic Approach

The application of Boolean algebra for testability analysis has received some attention in the two papers reviewed below. Both handle only combinational circuits and cannot be expanded easily to analyze sequential circuits.

In the first paper [Susskind 81], Susskind assigns a controllability and an observability connotation to the two parts of the Boolean difference. The approach is complete and theoretically consistent, but because the Boolean difference generates all possible tests for every fault in the circuit, large amounts of computation and storage are required, even for circuits consisting of only a few dozen gates.

In the second paper [Akers 82], a powerful logic structure [Akers 76] is employed to count the number of tests that detect each fault. The measure of testability Akers and Krishnamurthi propose is a lower bound on the number of tests necessary to meet a prespecified set of test requirements for the whole logic circuit. The necessary number of tests per circuit is counted and is propagated through the network. If reconvergent fanout is present, it is taken into account, and a near-optimal partitioning into fault-equivalence classes is evaluated. The method involves propagating local effects and requires at most four passes through the circuit.

2.5.8. A Critical View

Each of the testability analysis approaches reviewed above presents a solution to the problem of estimating the potential testing difficulties. The various approaches differ in domain and method, but share two basic problems:

(1) sequential circuits are not analyzed correctly, and

(2) redundant faults are not identified.

In order to keep computations simple, the testability analysis approaches dealing with sequential circuits employ overly simplified models, which lose the essence of sequential behavior — the state transition — due to low resolution. However, an appropriate model, e.g. the Huffman model presented before, is not feasible, since it requires vast computational effort and data storage.

Only two approaches [Bardell 82] [Susskind 81] identify redundancy, but at a high cost in computation and storage requirements. Both methods take into account *all* signal dependencies in the entire network, in contrast to the other methods that assume all signals to be independent. (In Chapter 3, the necessity of the dependent signal assumption is proven.)

Simple circuits have been reported [Savir 82] [Ratiu 82] that are easy to control, easy to observe, but comprise untestable (redundant) faults. However, testability analysis approaches that ignore signal dependencies predict good testability for such redundant faults.

Experimental data supporting the preceding results has been presented by Agrawal and Mercer [Agrawal 82]. They view the testability measure as a statistical estimator and calculate the correlation between its capability to predict which individual faults can be detected and the the fault data

(obtained through test generation and fault simulation) for a large chip. The resulting coefficient of correlation is always less than 0.4 and shows that a level of resolution exists for which a testability measure can provide useful information, but below which predictions are erroneous.

For VLSI circuits, the published testability analysis approaches do not provide reliable fault data to aid in test generation. However, if used interactively during design, testability analysis identifies *some* of the potentiai testing difficulties and, therefore, constitutes a valuable tool to educate designers about testing.

CHAPTER 3

THEORETICAL FOUNDATION AND STRATEGY IN VICTOR

3.1. INTRODUCTION

The theoretical foundation and the strategy for VICTOR, a new approach for providing reliable fault data for VLSI circuits, are presented. Models for the logic circuit and the fanout node are defined, combinational circuits and redundancies are classified, and two theorems on the links among convergent fanout, dependent signals, and redundant faults are introduced. Finally, the strategy in the design of the VICTOR approach to global redundancy identification and test generation is outlined.

3.2. LOGIC CIRCUIT AND FANOUT NODE MODELS

The circuit and fanout node models and the terminology used in VICTOR [Ratiu 82] are introduced.

3.2.1. Logic Circuit Model

A logic circuit is a network of cells interconnected via unidirectional links, called nodes, in dialogue with the outside world through its primary inputs (PIs) and primary outputs (POs). Each cell in the network is a functional block that performs a predefined set of logic operations and has its own cell inputs and outputs. Cell operations, when combined according to the interconnecting circuit topology, yield the logic function implemented by the circuit as a whole. Every link to a cell represents the location of a stuck-at

fault, and every node connecting exactly two cells corresponds to two equivalent faults, an output fault for the driving cell and an input fault for the driven cell.

3.2.2. Fanout Node Model and Terminology

A link that fans out to several destinations is called a *fanout node* and consists of a *root* (stem, origin) and its corresponding *branches*. The fanout of a node to n places "means the reproduction of that atom [node] in n distinct physical positions" [Kuck 78]. Although fanout root and branches represent the same (electric) circuit node, faults located on root and branches are not equivalent. A test that detects a root fault also detects at least one branch fault, yet a local analysis cannot identify the branch; a test that detects a branch fault must also detect the root fault, but rarely detects some of the other branch faults. Therefore, a correct fault analysis of a fanout node requires explicit faults for the fanout root and the branches.

Since most description languages of logic circuits assign only one name per circuit node, the same name designates root and branches of a fanout node. To distinguish among these different fault locations, the root is assigned the name of the fanout node, and each branch is assigned a composite node name of the form *root-pin-output*. Root is the original circuit node name (the fanout root), while pin is the number of the input pin and output is the name of the first output node of the cell connected to the fanout branch (an ordering of the cell inputs and outputs is assumed). Figure 3.1 illustrates the described fanout naming convention.

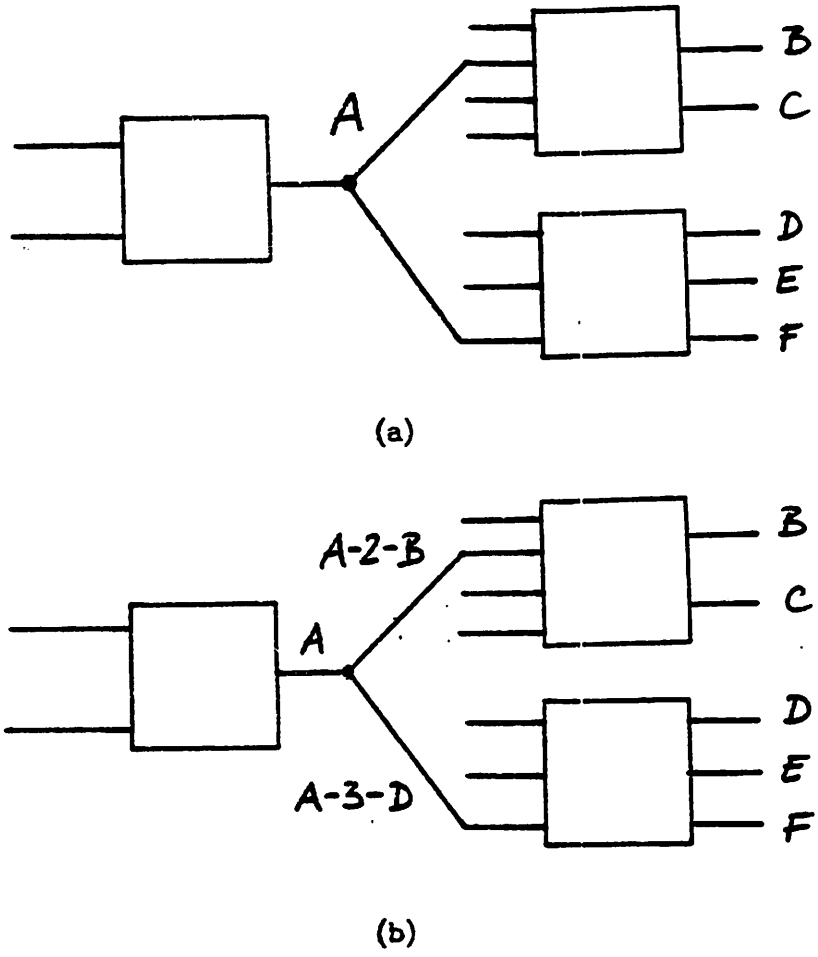


Figure 3.1 Fanout node naming convention.
(a) Original name for fanout node A
(b) Composite names for the branches of fanout node A

3.3. CIRCUIT AND REDUNDANCY CLASSIFICATION

3.3.1. Classification of Combinational Circuits

A classification of combinational networks based only on fanout topology is proposed. Instead of using the traditional terms of nonreconvergent and reconvergent fanout [Armstrong 66] [Schertz 72], the terms divergent and con-

vergent fanout are used for this purely topological classification. Three classes of combinational circuits are distinguished: fanout-free circuits, divergent fanout circuits, and convergent fanout circuits.

Fanout-free circuits do not contain any fanout nodes and have the simplest structure. A typical example is the AND-OR-INVERT cell of Figure 2.1, in which each logic gate has exactly one immediate successor. Only few large fanout-free circuits have any practical value.

Divergent fanout circuits contain fanout nodes whose branch successors always diverge, i.e., for all fanout nodes, no two successors of a node ever serve as inputs to the same cell. Since divergent fanout circuits do not include loops (sets of branches forming a closed path), divergent fanout circuits are tree networks [IEEE 77] and the general properties of tree structures apply [Aho 74] [Knuth 73]. Although more important than the fanout-free circuits, divergent fanout circuits are limited to special applications that require tree-like structures, e.g., error correction circuitry.

Convergent fanout circuits contain fanout nodes that have at least two convergent branch successors, i.e., at least one fanout node exists whose successors serve as inputs to the same cell. By definition, convergent fanout circuits include loops, allow arbitrary fanout topology and are, therefore, general, nontree networks. Most useful digital circuits and practically all VLSI circuits belong to this category; hence, any circuit investigation that ignores convergent fanout ignores an essential circuit aspect and produces incorrect results.

3.3.2. Classification of Redundant Faults

By definition, a fault is redundant if no test exists that can detect it. An arbitrary node K is called *totally redundant* if both faults K/0 and K/1 are redundant; if one of the two faults is irredundant, K is called *partially redundant*. Three classes of redundant faults in a circuit can be distinguished: uncontrollable faults, unobservable faults, and untestable-controllable-observable faults.

Uncontrollable faults cannot be provoked from the PIs, i.e., no pattern exists that can set or reset the corresponding node to the correct logic value. Accordingly, uncontrollable stuck-at-0 faults are 1-uncontrollable, and uncontrollable stuck-at-1 faults are 0-uncontrollable. For example, node K in Figure 3.2 is always at logic 0, since A-1-K and J always carry A and A'. Fault K/0 cannot be provoked; therefore, fault K/0 is 1-uncontrollable, and node K is partially redundant.

Unobservable faults cannot be propagated to the POs, i.e., no pattern exists that can monitor the corresponding node from the POs. In Figure 3.3, a

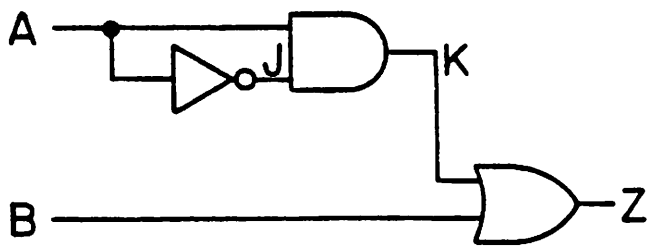


Figure 3.2 Redundant circuit containing 1-uncontrollable fault K/0.

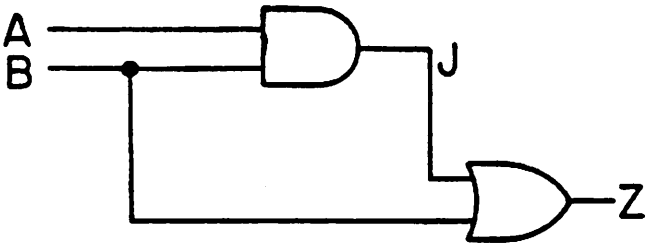


Figure 3.3 Redundant circuit containing unobservable faults A/0 and A/1.

fault on node A propagates to the output Z if and only if both B-2-J=1 and B-2-Z=0, but these conditions require node B to be simultaneously at logic 0 and 1, which is impossible. Therefore, both faults A/0 and A/1 are unobservable, and node A is totally redundant.

Untestable-controllable-observable faults can be provoked from the PIs, can be propagated to the POs, but no pattern exists that can both provoke and propagate the faults at the same time. When trying to match the two patterns, all combinations lead to PI conflicts. Schneider's example [Schneider 67], shown in Figure 3.4, contains two faults, B-1-K/0 and C-2-K/0, that are both controllable and observable, but untestable. Pattern 0000 is the only one that propagates the faults to the POs, and it cannot provoke either fault, although both faults are easily controllable.

A fault is redundant if and only if it is uncontrollable, unobservable, or untestable-controllable-observable. Uncontrollable redundancies are easiest to identify, since they involve just one PI function each. More effort is needed for the identification of unobservable redundancies, because establishing a propagation path to the POs involves a series of control conditions on the nodes adjacent to the path. A global evaluation of the circuit is needed to

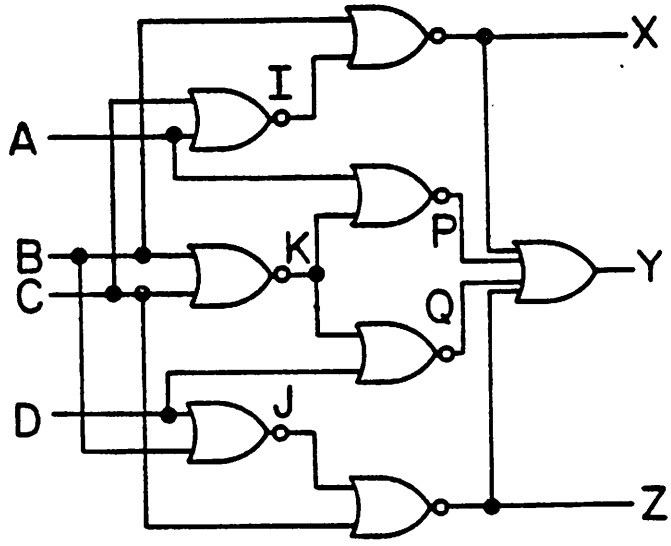


Figure 3.4 Redundant circuit containing untestable-controllable-observable faults B-1-K/0 and C-2-K/1.

identify the untestable-controllable-observable redundancies; shortcuts performing a partial analysis cannot find these redundancies. Therefore, systematic iterative attempts are required to match patterns for possible control and monitor paths through the entire circuit.

3.4. EQUIVALENCE AND IMPLICATION THEOREMS

An equivalence theorem between convergent fanout and signal dependencies and an implication theorem on the necessity of convergent fanout for redundancy are introduced in this section.

3.4.1. Signal Dependence and Convergent Fanout

Within the circuit, fanout branches supply internal signals in the same manner as the primary inputs supply external signals to the entire chip.

Both fanout branches and primary inputs represent the skeleton of the logic network and are commonly referred to as *checkpoints* [Breuer 76]. Their importance in combinational circuit testing has been recognized by Schertz and Metze [Schertz 72], who prove in their checkpoint theorem that any test that detects single (multiple) stuck faults on all checkpoints detects all single (multiple) faults in the circuit.

Every checkpoint is the root of a tree with a set of dependent branches and branch successors; new dependencies emerge when trees converge. When are two inputs to an arbitrary circuit cell dependent on each other?

> **Equivalence Theorem:** In a combinational network, inputs to a cell in the network depend on each other if and only if they belong to a convergent fanout tree, i.e., they stem from the same fanout node. ▪

> **Proof** (by definition): If some circuit branches depend on each other, they must belong to the same fanout tree, else they are independent. Since they are cell inputs, the fanout tree is convergent by definition. The opposite implication follows directly from the definition of a convergent fanout and the properties of a tree. ▪

If two cell inputs depend on each other, a condition imposed on one input affects the other input also. The Equivalence Theorem specifies the relation between function and structure in a general combinational network and guarantees that a complete structural (functional) analysis of dependencies is still complete if parts of it use the function (structure) as shown by the theorem.

3.4.2. Redundant Faults and Convergent Fanout

The relation between convergent fanout and redundant faults has been analyzed first by Armstrong when introducing the single-path sensitization method for test generation [Armstrong 66]. His method deals with two cases:

> *Case 1:* All reconverging fanout paths between a specified fanout node and a specified node of reconvergence have the same Inversion Parity.
>
> *Case 2:* Not all these paths have the same Inversion Parity. The Inversion Parity of a reconverging fanout is defined to be the number of inversions, modulo 2, along the path between the specified fanout node and the specified node of reconvergence.

If a test is applied that simultaneously sensitizes two converging fanout paths, Armstrong finds only the second case to entail redundant faults, since the output of the node of convergence does not change its value; therefore, the effect of the fault cannot propagate beyond it.

An example illustrating Armstrong's findings is shown in the circuit in Figure 3.2 which contains two paths with unequal inversion parity, <A, A-1-K, K> and <A, A-1-J, J, K>. As shown before, node K is 1-uncontrollable, because it does not change its value from logic 0 regardless of any logic change on node A, the originating fanout node (node A is totally redundant).

Attempts to expand Armstrong's inversion parity criterion for redundancy identification have failed [Ratiu 81], because inversion parity is neither necessary, nor sufficient to cause redundancy. For example, the circuit in Figure 3.3 contains the totally redundant node A, even though the two convergent fanout paths, <B, B-2-J, J, Z> and <B, B-2-Z, Z>, have equal inversion parity. On the other hand, the irredundant circuit in Figure 3.5 (a particular implementation of an XOR gate), contains two convergent fanout paths, <A, A-1-M, M, Z>, <A, A-1-K, K, N, Z>, and <B, B-1-J, J, M, Z>, <B, B-2-N, N, Z>, with unequal inversion parity, but no faults are redundant.

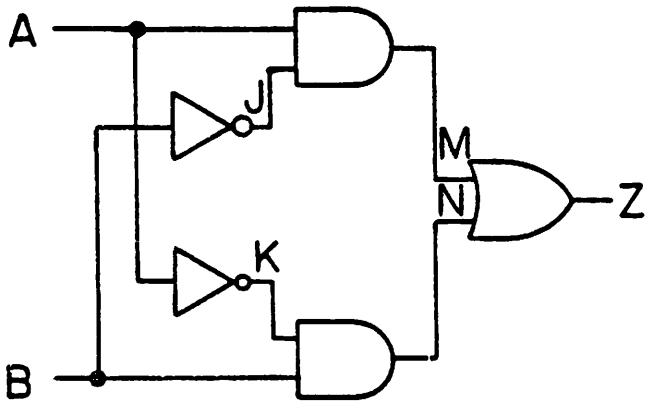


Figure 3.5 Irredundant circuit containing multiple convergences.

In general, when multiple convergence occurs at the input of a cell, nothing can be inferred regarding the redundant nature of the faults at the originating fanout node or after the point of convergence. In Schneider's example, Figure 3.4, the four input signals converge at the output NOR gate Y, and all have unequal inversion parity along the different paths, yet only faults B-1-K/0 and C-2-K/0 are redundant.

Since inversion parity is neither necessary, nor sufficient to cause redundancy, its use for redundancy identification (RI) is limited. However, convergent fanout is related to redundant faults.

> **Implication Theorem:** A fault is redundant if all paths from the primary inputs to the primary outputs that include the fault traverse a convergent fanout circuit. ▪

Proof (by contradiction): A path from the PIs to the POs traverses a convergent fanout circuit if and only if either all control paths are convergent, or all monitor paths are convergent, or the control and monitor paths are divergent separately, but convergent if taken together as a whole path. In any case, dependent signals arise (as proven in the Equivalence Theorem). The assumption is made that signals are independent, i.e., the path does not traverse a convergent fanout circuit, and redundant faults exist. Then, a change in any of the signals uniquely affects the circuit function and, by definition, the circuit cannot contain redundant faults; the initial assumption has been contradicted. Therefore, the path must traverse a convergent fanout circuit for the fault to be redundant. ∎

The Implication Theorem states that redundancy requires convergence: no fault can be redundant if a divergent fanout path exists along which the fault location can be controlled and observed. The opposite implication is false, however, since not every convergence causes redundancy. As was shown before, the XOR gate of Figure 3.5 contains a double convergence at the output gate, but no redundancies are present in the circuit.

3.5. VICTOR STRATEGY

After having laid out the theoretical foundation, the strategy for redundancy identification and test generation in VICTOR is presented.

3.5.1. Goal and Feasibility Conditions

The goal in the design of VICTOR (VLSI Identifier of Controllability, Testability, Observability, and Redundancy) has been a fast method for producing reliable fault data for VLSI circuits. To attain this goal, VICTOR must meet several feasibility conditions.

Linear complexity. Tailored to the fast-growing VLSI chip complexity, computer-aided testing (CAT) tools must involve a number of variables and of operations that are both linear or near-linear functions of the chip complexity; the complexity of the data base and the algorithm is then said to be *linear* or *near-linear*. As was shown in Chapter 2, the testing of sequential circuits employs a circuit model that grows exponentially with circuit size, but by incorporating a scan-path, the testing can be performed on an equivalent linear combinational model of the original circuit, i.e., the circuit becomes scan-testable. To fulfill the linear-complexity requirement, VICTOR handles only combinational circuits, hence only scan-testable VLSI circuits. However, current trends in VLSI design show a growing willingness of circuit designers to incorporate such scan-paths (see Chapter 2).

Appropriate fault models. The tools involved in the testing of a circuit (testability analyzer, redundancy identifier, automated test generator, and fault simulator) must use the same fault model in order to form a useful test system. For the reasons outlined in Chapter 2, VICTOR uses the single-stuck fault model described above (fanout root and branches of a fanout node represent separate fault locations).

Dependent signal assumption. Signals must be assumed dependent on each other and all dependencies must be taken into account. Both the author's experience with SCOAP and the analysis of SCOAP by Agrawal and

Mercer [Agrawal 82] show that testability analyzers employing the independent signal assumption do not recognize redundant faults. Furthermore, the theorems outlined above prove that a necessary condition to identify redundancy is to consider all signal dependencies.

3.5.2. Global Linear Estimation

A novel feature in VICTOR is the linear and global approach to redundancy identification. Rather than analyzing faults in succession, as is done in conventional automatic test pattern generation (ATG), VICTOR sweeps through the entire circuit a few times and processes all faults. A fixed number of passes is needed, regardless of circuit complexity, and no iterations or backtracks are involved.

All necessary signal dependencies in the circuit are expressed as functions of PIs only in order to keep the data base linear. (The PI count of a VLSI chip is, owing to packaging constraints, a weak function of circuit complexity.) However, the information on signal dependencies is not approximate, but exact and complete.

3.5.3. Control, Monitor, and Test Patterns

Signal dependencies on a node are expressed as PI *patterns*, i.e., strings of logic values composed of one value for each PI. Modeled after the classification of redundancies introduced earlier, three patterns are calculated for each fault: control, monitor, and test pattern.

The *control pattern* of a node is a collection of sufficient PI conditions to provoke a fault on that node. Since two faults are possible, stuck-at-0 and stuck-at-1, two control patterns exist, *set pattern* and *reset pattern*.

The *monitor pattern* of a node is a collection of sufficient PI conditions to propagate a fault on that node to the POs. Finding a possible monitor pattern for a node is inherently more difficult than finding the appropriate set or reset pattern, because the monitor conditions are the intersection of all control conditions that sensitize a path to the POs.

Finally, the *test pattern* of a node is a collection of sufficient PI assignments that simultaneously provoke (from the PIs) and propagate (to the POs) a fault on that node. Thus, the desired test pattern results from the merger of control and monitor pattern: reset and monitor pattern for a stuck-at-1 fault, and set and monitor pattern for a stuck-at-0 fault.

3.5.4. The Risk Measure Heuristic

The underlying heuristic in VICTOR is the risk of convergence, the selection criterion among different possible patterns. The Implication Theorem shows that convergence leads to conflict in the calculation of the test patterns for redundant faults, but also for some irredundant faults. Faults for which conflicts arise during test generation are called *potential redundancies* and comprise the truly redundant faults and some hard-to-test irredundant faults, called *false redundancies*. If the pattern corresponding to the lowest risk of convergence is selected, the risk of conflict is minimized, hence a selection of patterns based on the lowest risk of convergence minimizes the number of false redundancies during test generation. Applied successively to each cell during a circuit pass, the minimum risk selection tries to find a conflict-free test pattern for every fault; a global optimum is approximated through a succession of local optimizations, a standard CAD technique. Some false redundancies remain (a linear approach cannot solve an NP-complete

problem), but their number has been minimized.

In existing ATG programs, redundancies, true or false, are the cause of the many iterations that degrade program performance. The amount of effort per fault (iteration count or computer time) is limited in most ATG programs, and the test generation for the fault is stopped once this limit has been reached. Without further probing, false redundancies cannot be separated from the genuine ones; hence, the fault may or may not be redundant, situation which the choice of the term *potential* redundancy tries to suggest.

Two types of errors can occur in redundancy identification: (1) redundant faults are falsely predicted irredundant, and (2) irredundant faults are falsely predicted redundant. The first type renders the RI technique useless and is, therefore, catastrophic (such errors occur in the testability analysis methods described in Chapter 2). The second type produces false redundancies and is acceptable, as long as it does not contaminate the major part of the estimation.

The approach to redundancy identification in VICTOR resembles statistical estimation techniques, where absolute precision matters less than knowledge of error behavior and error bounds. In VICTOR, the lowest risk heuristic biases the estimate such that no errors of the first type ever occur at the cost of many errors of the second type. VICTOR identifies all redundancies but its RI estimate includes many false redundancies that must be dealt with later.

CHAPTER 4

VICTOR ALGORITHM

4.1. INTRODUCTION

The basic algorithm used in VICTOR consists of four steps: circuit levelizing, controllability calculation, observability calculation, and redundancy identification and test generation. Two testability operations, pattern selection and intersection, and two testability measures, risk and size, are employed [Ratiu 82]. To illustrate the succession of operations in VICTOR, a small circuit example is analyzed. The chapter ends with an analysis of the linear complexity of the algorithm.

4.2. VICTOR TESTABILITY PRIMITIVES

Before detailing the VICTOR algorithm, the testability primitives underlying its operations are defined. For the purpose of redundancy identification and test generation, the circumstances under which a node is set, reset, or monitored are characterized by a a testability triplet of pattern, risk, and size. The pattern expresses exact primary input (PI) conditions, while risk and size are estimates of the risk of convergence conflict and pattern count, respectively. Two pattern operations, selection and intersection, are defined.

4.2.1. The Set, Reset, and Monitor Pattern

Given a circuit and its primary inputs (PIs) and primary outputs (POs), three patterns are defined for an arbitrary node V in the circuit:

- **set pattern,** V_s: a sufficient collection of PI conditions to force node V to logic 1.
- **reset pattern,** V_r: a sufficient collection of PI conditions to force node V to logic 0.
- **monitor pattern,** V_m: a sufficient collection of PI conditions to sensitize a path from node V to the POs.

Node patterns represent PI signal dependencies expressed as an ordered string of logic symbols. Patterns contain exactly one symbol for each primary input; consequently, patterns are of equal length for a given circuit. Currently, four-valued logic is implemented, i.e., a symbol may take one of four logic values (but VICTOR may handle any finite multiple-valued logic).

0	logic 0 assignment
1	logic 1 assignment
x	no assignment (don't care)
#	conflicting logic 0 and logic 1 assignment (clash)

The circuit in Figure 4.1 has two primary inputs only, A and B; thus, all node patterns consist of two symbols, the first corresponding to A, and the second to B. The primary inputs themselves have the following control patterns:

$$A_s = 1x, \quad A_r = 0x; \qquad B_s = x1, \quad B_r = x0.$$

The structure of a PI control pattern for any circuit is unique and consists of a 0 or 1 symbol in the position of the respective primary input and an x symbol in all other positions. Monitor patterns for any circuit contain only x symbols, since primary outputs are by definition observable and, therefore, do not impose PI constraints.

Usually, internal nodes can be set, reset, or monitored in a variety of ways; hence, many patterns of one type exist. For instance, the three patterns 01, 11, and x1 are valid set patterns for node Z. An internal node has exactly one pattern if and only if it depends on all primary inputs, i.e., the pattern consists exclusively of 0 and 1 symbols.

A clash in a pattern highlights a PI convergence conflict and thus a potential redundancy. For example, K=1 requires both A-1-K=1 and J=1, but these conditions are equivalent to A=0 and A=1, a conflicting requirement. In Chapter 3, node K has been shown to be partially redundant (1-uncontrollable); hence, the clash in the set pattern K_s=#x corresponds to a true redundancy. Condition Z=1 requires either K=1 or B=1, and the two set patterns for node Z are #x and x1; the fault on node Z can be considered a potential redundancy if pattern #x is chosen. Such a decision is in general far less obvious and affects the patterns of many succeeding nodes. If, for instance, node Z is deeply embedded in the network and pattern #x is chosen, many successors of node Z would be identified incorrectly as potential redundancies; a false redundancy contaminates other patterns in the circuit.

4.2.2. The Risk and Size Testability Measures

Except for the patterns, the testability of each node is characterized by two positive integers, a risk and a size testability measure. The former expresses the risk of convergence associated with each pattern, hence the risk of conflict, and the latter estimates the total number of patterns that exist per node. The risk and size measures are defined below for an arbitrary node V in the circuit (the evaluation of the measures is presented later in this chapter).

Risk Measures

- **set risk,** $risk(V_s)$: weighted sum of constrained checkpoints on the chosen path from the PIs to set node V, for pattern V_s.
- **reset risk,** $risk(V_r)$: weighted sum of constrained checkpoints on the chosen path from the PIs to reset node V, for pattern V_r.
- **monitor risk,** $risk(V_m)$: weighted sum of constrained checkpoints on the sensitized path from the node V to the POs for pattern V_m.

Size Measures

- **set size,** $size(V_s)$: number of patterns V_s that can set node V.
- **reset size,** $size(V_r)$: number of patterns V_r that can reset node V.
- **monitor size,** $size(V_m)$: number of patterns V_m that can monitor node V.

4.2.3. Pattern Selection

As shown before, the many ways to set, reset, or monitor a node produce a multitude of patterns for the given action. If propagated through the network, these patterns multiply rapidly from node to node and computations as well as data storage per node become unwieldy. Therefore, one pattern is chosen among the various patterns of a certain type, and this pattern alone enters further computations. The operation is called *pattern selection*, and the question mark has been chosen as the selection operator owing to its everyday connotation.

Definition: Given two patterns, P and Q, the selection of P and Q produces a pattern S equal to the lowest-risk, lowest-level pattern between P and Q. The selection of P and Q is written as follows:

$$S = P\ ?\ Q = ?\ (P,Q)$$

The risk and size of the resulting pattern S are:

$$risk(S) = risk(P)\ \ or\ \ risk(Q), \quad size(S) = size(P) + size(Q)$$

The selection operation implements the minimum risk strategy outlined in Chapter 3 by using the risk testability measure. Whenever a choice exists, the pattern with the lowest risk of convergence is selected, and the chance for potential redundancies is minimized. In case of a tie, the lowest-level pattern is selected (levels are introduced in the next section of this chapter), and the choice implements a shortest-path heuristic. If the tie persists, the first pattern to enter the operation is chosen. Pattern selection always generates a unique result, and the operation is commutative, associative, and any pattern of risk 99999 acts as identity element. The selection heuristics strongly influence the algebraic structure.

The risk measure of the selected pattern becomes the risk measure of the result. To evaluate the size measure of the result, the individual sizes of P and Q must be combined to reflect the increased pattern count. Since signal dependency is handled correctly by the patterns, signals are assumed independent to simplify the size calculation which then becomes a simple addition.

Example: Let P, Q, and R be the patterns below.

$$P=0xx\#,\ \ r(P)=99999,\ \ s(P)=0,\ \ level=2$$

$$Q=x1x0,\ \ r(Q)=32,\ \ s(Q)=9,\ \ level=6$$

$$R=0x10,\ \ r(R)=32,\ \ s(R)=5,\ \ level=4$$

Then,

$$S=P\ ?\ Q=Q=x1x0,\ \ r(S)=r(Q)=32,\ \ s(S)=s(P)+s(Q)=0+9=9$$

$$T=Q\,?\;R=R=0x10,\quad r(T)=r(R)=32,\quad s(T)=s(Q)+s(R)=9+5=14$$

During every selection, pattern information is lost, but the pattern considered most likely to succeed is propagated; hence, the damage is heuristically confined. No new patterns are generated, because existing patterns are steered to successive nodes, and therefore the selection operation is independent of the logic structure.

4.2.4. Pattern Intersection

To set, reset, or monitor a node sometimes requires the simultaneous action of two or more node patterns. For example, to observe one of the inputs of a two-input AND gate requires that the output be monitored *and* the other input be set, i.e., the monitor pattern of the input under investigation is a combination of the output monitor pattern and the other input reset pattern. The operation defining such a combination of patterns is called *pattern intersection*, and the exclamation mark has been chosen as the intersection operator owing to its everyday imperative connotation.

Definition: Given two patterns, P and Q, the intersection of P and Q produces a pattern I, the symbols of which are obtained by combining the two homologous symbols in P and Q according to the following combination table:

!	0	1	x	#
0	0	#	0	#
1	#	1	1	#
x	0	1	x	#
#	#	#	#	#

The intersection of P and Q is written as follows:

$$I = P \,!\, Q = !\,(P,Q)$$

If the resulting pattern I includes a clash (#), then the risk and the size of the resulting pattern I are:

$$risk(I) = 99999, \quad size(I) = 0.$$

If pattern I includes no clashes, then the measures are:

$$risk(I) = risk(P) + risk(Q), \quad size(I) = size(P) * size(Q).$$

The intersection operation guarantees that signal dependencies are taken into account and that all additional constraints (and potential conflicts) enter the computations as they arise. If patterns must act simultaneously, then the primary input conditions they represent must be met simultaneously; hence, the symbols must be combined. The pattern combination rule for four-valued logic defined in the above table is commutative, associative, and has a unique identity element (a pattern composed entirely of don't cares); therefore, it constitutes an abelian semigroup. Any other composition table for multiple-valued logic can be used if the operation it defines has the same algebraic properties. As expected, don't cares are overridden by any symbol, clashes override any symbol, and new clashes result only out of a 0-1 or 1-0 intersection.

Signals are again assumed independent when calculating the testability measures of the result. The risk measure of the result is the sum of the two initial risks, since both patterns must act simultaneously. For each possible pattern for P, the gamut of patterns for Q is available; hence, the size of I is the product of the sizes of P and Q.

Example: Let P, Q, R, and T be the patterns below.

$$P=0xx\#,\quad r(P)=99999,\quad s(P)=0$$

$$Q=x1x0,\quad r(Q)=32,\quad s(Q)=9$$

$$R=0x10,\quad r(R)=32,\quad s(R)=5$$

$$T=1010,\quad r(T)=18,\quad s(T)=3$$

Then,

$$I=P!Q=01x\#,\quad r(I)=99999,\quad s(I)=0$$

$$J=Q!R=0110,\quad r(J)=r(Q)+r(R)=32+32=64,\quad s(J)=s(Q)*s(R)=9*5=45$$

$$K=R!T=\#010,\quad r(K)=99999,\quad s(K)=0$$

No information is lost during pattern intersection, but an improper choice in a previous selection may cause the occurrence of a clash when a clash-free pattern for the node exists. Unless one of the patterns is the identity element, pattern intersection always generates a new pattern.

4.3. CIRCUIT LEVELIZING

Circuit levelizing, the first step in the basic algorithm of VICTOR, partitions the circuit description into segments corresponding to complete cell modules (cell name and cell input and output nodes) and orders the cell modules. Circuit levelizing is static and is based on topology. If the signal flow determines the relative ranking of cells, the ordering is dynamic and is called event-driven or selective trace. Dynamic ordering is especially efficient if a logic change causes a low level of activity in the network. Since in VICTOR all signal dependencies are taken into account, the testability evaluation at every node causes a high level of activity; hence, circuit levelizing is used in VICTOR.

Levelizing is the process of assigning a nonnegative integer value to every node and cell in the network. By definition, the level of the PI nodes is zero, and the level of a node other than a primary input is the level of the cell which has the node as an output. The level of a cell in the network is calculated as the maximum input node level plus one.

In VICTOR, the levelizing procedure starts out by initializing all node levels to infinity and PI levels to zero. The procedure consists of successive searches for the cells that have not been assigned a logic level but the inputs of which have been already levelized. Once such a cell is found, its level is calculated, and the operation is repeated. The procedure stops when a circuit pass does not produce any new cell levels. When applied to the circuit example in Figure 4.1 (a repetition of Figure 3.2), circuit levelizing produces the levelized cell list INVERTER, AND2, OR2, and the following node levels:

level 0: A, B
level 1: J
level 2: K
level 3: Z

The use of circuit levelizing has several implications. First, each circuit node is considered to be the output of at most one cell, otherwise the node

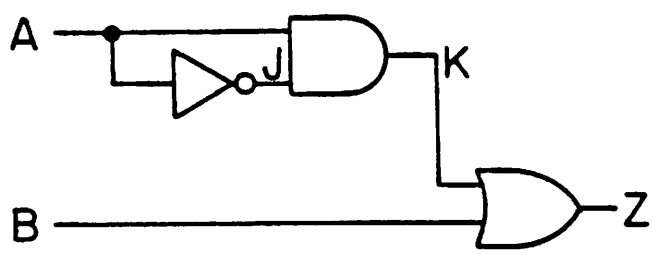


Figure 4.1 Simple circuit example.

level is undefined; hence, tied cell outputs (wired logic) cannot be handled by VICTOR. Second, if all nodes are connected properly, infinite levels point out the presence of feedback loops in the circuit. If a loop exists, at least one cell input level is infinite; hence, all cell output levels are infinite too, and the situation is replicated for the cells in the loop and the cells driven by the ones in the loop. VICTOR identifies the presence of feedback loops (design-rule violations in combinational circuits) and gives the boundary of the circuit area containing the feedback loop as the list of cells with finite levels feeding cells with infinite levels. Third, levelizing introduces a total order relation in the network and generates a *levelized cell list*, a list of cells sorted in increasing order of their level. The list comprises all cells in the network and forms the basis of the ensuing testability calculations.

4.4. CONTROLLABILITY CALCULATION

The controllability calculation in VICTOR consists of the controllability initialization and the evaluation of the node controllability triplets. The computation is completed in a single pass through the circuit and proceeds cell by cell in the order given by the levelized cell list. Fanout nodes are handled as single fault locations, because the controllability triplets for root and branches are identical: controlling the root also controls the branches and vice versa.

4.4.1. Controllability Initialization

Since the calculation of controllability parallels the forward flow of signals from the primary inputs to the primary outputs, only the PI triplets — pattern, risk, and size — need to be initialized. For the rest of the nodes,

triplets are generated as the controllability calculation progresses.

The PI set pattern consists of a 1 for the symbol of the particular primary input and all don't cares (x) for the other PI symbols. Similarly, the PI reset pattern is formed of only don't cares and a 0 for the particular primary input. Thus, the PI patterns of the example circuit are:

$$A_s = 1x, \quad A_r = 0x,$$

$$B_s = x1, \quad B_r = x0.$$

The set and reset risk for a primary input driving a single cell is equal to one. For a fanout node, however, the risk of convergence grows with fanout and distance to the primary outputs. A "risk rectangle area" is calculated as the product of fanout and the level count to the furthest primary output. For the analyzed example:

$$risk(A_s) = risk(A_r) = 2*3 = 6, \quad risk(B_s) = risk(B_r) = 1$$

A PI node can be set or reset in exactly one way, hence both set and reset sizes are equal to one.

$$size(A_s) = size(A_r) = 1, \quad size(B_s) = size(B_r) = 1$$

4.4.2. Cell Controllability Calculation

Except for the primary inputs, all circuit nodes are cell output nodes. Therefore, node control triplets are calculated by processing cell after cell from the levelized cell list. The rules to derive output control triplets from input control triplets for a cell constitute its *control equations*, which must be predefined in a cell controllability library (see Appendix 2).

The procedure to calculate control triplets outlined above is shown on the circuit example. Circuit cells are dealt with in the following order: INVERTER, AND2, OR2. To simplify notation, the control pattern, risk, and size

of a node are represented collectively as triplets. The set and reset triplet of any node V in the circuit are:

$$V_{TS} = \{\ V_s,\ risk(V_s),\ size(V_s)\ \}$$

$$V_{TR} = \{\ V_r,\ risk(V_r),\ size(V_r)\ \}$$

The first processed cell, the INVERTER, is governed by the following control equations:

$$J_s = A_r, \quad J_r = A_s$$

(The relation between patterns implies a similar relation for risk and size.) The control triplets for node J are J_{TS}={0x, 6, 1} and J_{TR}={1x, 6, 1}.

The next cell is the two-input AND gate, AND2, which has the control equations given below:

$$K_s = A_s \mathbin{!} J_s, \quad K_r = A_r \mathbin{?} J_r.$$

(Risk and size evaluation is implicit to the pattern operations.) The control triplets for K are K_{TS}={#x, 99999, 0} and K_{TR}={0x, 6, 2}.

The final cell is the 2-input OR gate, OR2, with the following control equations:

$$Z_s = K_s \mathbin{?} B_s, \quad Z_r = K_r \mathbin{!} B_r.$$

The control triplets for Z, the primary output of the circuit, are Z_{TS}={x1, 1, 1} and Z_{TR}={00, 7, 2}.

After processing the last entry in the levelized cell list, the controllability calculation is complete. If any of the cell outputs is a fanout node, its risk is increased by an amount equal to the product of the fanout and the number of levels to the furthest PO node (the additional risk is analogous to the "risk rectangle area" of the PI fanout nodes).

Cell control equations stem from the cell logic equations and have similar algebraic properties. Based on the associativity of pattern selection and intersection, the generic gates (AND/NAND, OR/NOR) need only a single entry in the controllability library, regardless of the number of cell inputs. A method and a program for translating logic into testability equations, LITE [Van Egmond 82], has been developed.

4.5. OBSERVABILITY CALCULATION

The observability calculation in VICTOR consists of the observability initialization and the evaluation of the node observability triplets. Similar to the controllability calculation, the computation is completed in a single pass through the circuit, but proceeds in the reverse order of the levelized cell list. When analyzing the observability of fanout nodes, branches must be identified and handled as entities separate from the root; therefore, VICTOR assigns them unique names according to the convention introduced in Chapter 3. The triplet notation introduced for node controllability is used for node observability also.

4.5.1. Observability Initialization

Since the observability calculation follows the inverted signal flow, only the PO node triplets need to be initialized. Primary outputs are uniquely observable without imposing any checkpoint constraints; hence, their monitor pattern is the identity element for intersection, the risk is zero, and size is 1. In the circuit example of Figure 4.1, the monitor triplet of primary output Z is initialized this way: Z_{TM}={xx, 0, 1}.

4.5.2. Cell Observability Calculation

Observability and controllability are dual notions and so are the procedures for calculating them. Node monitor triplets are calculated cell by cell, from the primary outputs to the primary inputs, in reverse order of the levelized cell list. The rules to derive input monitor triplets from output monitor triplets for a cell constitute its *monitor equations*, which must be predefined in a cell observability library (see Appendix 2).

The procedure to calculate monitor triplets outlined above is shown for the circuit example in Figure 4.1. Circuit cells are dealt with in the following order: OR2, AND2, INVERTER.

The first cell to be processed, OR2, has the following monitor equations:

$$K_m = Z_m \,!\, B_r, \quad B_m = Z_m \,!\, K_r.$$

Monitor risk and size follow the definitions given earlier in this chapter. The monitor triplets for nodes K and B are K_{TM}={x0, 0, 1} and B_{TM}={0x, 0, 1}.

The next cell to be processed, AND2, has fanout node A as an input. VICTOR identifies it, names the branch A-1-K (root-pin-output, see Chapter 3), and calculates the branch risk as another "risk rectangle area" from the cell to the root of the fanout node. The risk of A-1-K, the product of fanout and the number of levels from the cell to the fanout root, 2*(2-0)=4, enters the monitor calculation of the *other* cell input, J. The input monitor triplets are calculated as

$$A{-}1{-}K_m = K_m \,!\, J_s, \quad J_m = K_m \,!\, A{-}1{-}K_s,$$

resulting in $A{-}1{-}K_{TM}$={00, 0, 1} and J_{TM}={10, 4, 1}. Because one of the cell inputs, A-1-K, is a fanout branch, VICTOR starts evaluating the root monitor triplet by performing the set-monitor and reset-monitor intersections and storing the results.

$$A{-}1{-}K_s \mid A{-}1{-}K_m = \#x,$$

$$A{-}1{-}K_r \mid A{-}1{-}K_m = 00.$$

The last cell in the observability pass, the INVERTER, is connected to the second branch of fanout node A. VICTOR names it A-1-J, calculates its risk, and using the cell monitor equations

$$A{-}1{-}J_m = J_m,$$

computes the monitor triplet $A{-}1{-}J_{TM}$={10, 4, 1}. Cell input A-1-J is a fanout branch, and the set-monitor and reset monitor intersections are calculated and stored again.

$$A{-}1{-}J_s \mid A{-}1{-}J_m = 10,$$

$$A{-}1{-}J_r \mid A{-}1{-}J_m = \#x.$$

For the calculation of fanout node observability, all branch monitor triplets must be evaluated first, and then the root pattern is set equal to the branch monitor pattern for which both set-monitor and reset-monitor intersections are clash-free. If many such patterns exist, the one with the least input constraints (most don't cares) is selected; the next tie-breaker is weight, then order of operation (the root calculation always generates a unique monitor triplet). The root risk and size are calculated following the rules defined for pattern selection. In no such pattern exists, then the root monitor triplet is set to {###...##, 99999, 0}.

In the example, node A-1-J is the second of two branches of node A; therefore, all data necessary for the evaluation of the root monitor triplet has been computed. For both branch patterns, the set-monitor and reset-monitor intersections contain a clash; thus, the root monitor triplet is set to A_m={##, 99999, 0}.

After a single pass through the levelized cell list, all observability values have been calculated. Monitor equations, like control equations, are derived from the logic equations of the cell, and for generic gates less entries in the library are needed: AND/NAND gates share the same observability library entry, as do OR/NOR and XOR/XNOR gates. Similar to cell control equations, the observability encoding of complex functional blocks can proceed automatically, as is done in LITE [Van Egmond 82].

4.6. TEST GENERATION AND REDUNDANCY IDENTIFICATION

4.6.1. Test Generation

A test for a fault is a sufficient collection of PI conditions that simultaneously control and observe the fault, i.e., provoke the fault from the primary inputs and propagate the fault to the primary outputs. The simultaneous requirements translate into simultaneous PI constraints identical to the ones defined previously for pattern intersection; hence, VICTOR generates tests by intersecting control and monitor patterns for every the node in the circuit. Two tests are needed to detect the stuck-at faults on an arbitrary node V:

$$V/0 \ test: \ V_s \ ! \ V_m, \qquad V/1 \ test: \ V_r \ ! \ V_m.$$

The resulting patterns may or may not include clashes.

Patterns containing no clashes, but only 0, 1, and x, represent valid test patterns and can be used as such. The detected faults are, by definition, irredundant. In the circuit shown in Figure 4.1, VICTOR finds eight irredundant faults by generating the following test patterns (stuck-at-0 tests on the left, stuck-at-1 tests on the right):

10		A-1-J
	00	A-1-K
01	00	B
	10	J
x1	00	Z

Although such a table grows only linearly with circuit size, the resulting amount of test data causes difficulties in test application [Muehldorf 81]. To limit this explosion of test data, VICTOR searches two times through the list of test vectors and achieves fault collapsing and test compaction.

Fault collapsing reduces the number of tests by matching faults detected by the same test pattern and eliminates multiple occurrences of the same test pattern. For instance, pattern 00 appears four times in the previous fault list, but just once followed by a four in the list of collapsed test patterns shown below:

00	...	4
10	...	2
x1	...	1
01	...	1

Test compaction reduces the number of collapsed tests by merging different test patterns that are equal but for some don't cares. For example, patterns x1 and 01 are compacted to 01, but patterns 00 and 10 cannot be compacted, because the resulting pattern, #0, comprises a clash. During test compaction, a don't care is the "weak" term (the identity element), and a 0-1 compaction leads to a clash; the operation implementing such rules is pattern intersection. When applied to the list of collapsed faults, test compaction generates the tests listed below in decreasing order of the number of faults detected per test.

00	...	4	50.0%	50.0%
01	...	2	25.0%	75.0%
10	...	2	25.0%	100.0%

The percentages give the fault coverage and the cumulative fault coverage per test vector for the irredundant faults only and *not* for the entire circuit.

4.6.2. Redundancy Identification

If the intersection of the control and monitor patterns for a fault results in a pattern that includes at least one clash, the impossible condition aborts the attempt to generate a test for the given fault. Every clash represents a PI conflict due to convergent fanout and indicates a potential redundancy. VICTOR identifies three types of redundancies (corresponding to the classification of redundancy introduced in Chapter 3):

(1) *uncontrollable redundancy (0 or 1):* a clash in the control patterns, no clash in the monitor pattern;

(2) *unobservable redundancy:* a clash in the monitor pattern, no clash in the control patterns;

(3) *untestable-controllable-observable redundancy:* no clash in the control or monitor patterns, but a clash in the test pattern.

In the example of Figure 4.1, the two node patterns including a clash, K_c=#x and A_m=#0, identify the 1-uncontrollable node K and the unobservable node A, which means that faults K/0, A/0, and A/1 are redundant. VICTOR finds six potentially redundant faults in the circuit. The six faults and their aborted test patterns are listed below (impossible test patterns for stuck-at-0 faults on the left, for stuck-at-1 faults on the right):

#0	#0	A
	#0	A-1-J
#0		A-1-K
#0		J
#0		K

Exhaustive circuit testing (patterns 00, 01, 10, 11) proves that the three additional faults, A-1-J/1, A-1-K/0, and J/0, are redundant and that no other redundancies exist; for the analyzed circuit, the redundancy estimate of VICTOR is correct.

In general, many potential redundancies identified by VICTOR are false redundancies. The linear, heuristic algorithm must err at times when trying to solve the NP-complete problem of fault detection, no matter how good the heuristics. However, the set of potential redundancies includes all true redundancies, since VICTOR detects all fanout convergences. The magnitude of the redundancy identification (RI) error is crucial, because the larger the set of false redundancies, the less information on true redundancies VICTOR gives.

The structure of the potential redundancies in VICTOR is illustrated by the onion model of Figure 4.2: a kernel of correct data, the true redundancies, surrounded by layers and layers of incorrect data, the false redundancies. When the onion grows to encompass a major portion of the fault set, and the kernel of true redundancies stays constant, the RI result loses information content. If the onion includes all circuit faults, the information content of the RI result is zero. VICTOR predicts the triviality that every node in the circuit could be redundant.

The task of identifying the redundant faults in a circuit can be viewed as a two-step process: (1) separate potentially redundant faults from the total fault set, and (2) eliminate false redundancies from the set of potential

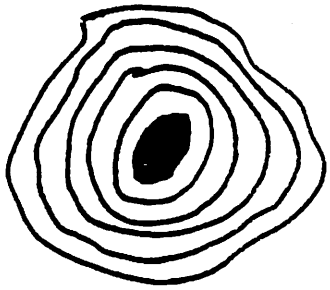


Figure 4.2 Onion model for potentially redundant faults

redundancies. The first step is performed by VICTOR during test generation: find a clash-free test pattern for a fault, thus prove the fault to be irredundant. In the second step, the set of potential redundancies is searched for irredundant faults, the false redundancies. The compacted test patterns generated previously are simulated and many potentially redundant faults are detected. In the onion model, the procedure is analogous to peeling off layer after layer of false redundancies, thus increasing the relative size of the kernel compared to the bulk of the onion. The efficiency of this process grows with circuit complexity, since a test in a large circuit can detect hundreds and sometimes thousands of faults [Bottorff 80]. The process residue is composed of hard-to-test irredundant faults and, if the circuit is redundant, a kernel of redundant faults, both of which require much effort during test generation.

If in a particular circuit VICTOR identifies half the circuit faults as potentially redundant, the RI result provides little information, since half the faults in the circuit must be investigated further. Without any previous information,

all circuit nodes must be investigated. In rare cases, such as the example analyzed in this chapter, the potential redundancies identified by VICTOR contains no false redundancies.

4.7. ALGORITHM COMPLEXITY

VICTOR constitutes an approach to global redundancy identification and test generation that manages to keep algorithm complexity linear through the extensive use of heuristics.

Except for a few search procedures, the operations in the four steps of the algorithm relate linearly to circuit fault and primary input count. Once the circuit has been read in and the node structure established (an *nlogn* process present in any CAD algorithm), the ensuing circuit levelizing is linear. Then, both controllability and observability calculation are complete after a single pass through the levelized cell list, and the amount of cell computation scales linearly with the number of node testability triplets per cell. Test generation and redundancy identification involve one pass through the entire fault list and one pattern operation per fault. Finally, fault collapsing and test compaction for irredundant faults require an *nlogn* search each.

Data storage in VICTOR is a linear function of circuit fault and primary input count. Every node is assigned a set, reset, and monitor triplet, and two test vectors (for the two stuck faults). Therefore, the amount of storage required per node is 2*3=6 integers and 3+2=5 patterns. The patterns for a given circuit depend only on the number of "logical primary inputs", i.e., the chip primary input count and the number of points in the scan path used as test inputs. The former is constrained by packaging constraints, and the latter is constrained by memory limitations during test application [Muehldorf

81]. Hence, the length of a pattern is a weak function of circuit complexity and can be considered constant for VLSI. Finally, intermediate calculations employ a fixed number of variables.

A comparison can be drawn between the Boolean Difference algorithm, the D-algorithm [Breuer 76], and the algorithm in VICTOR. The Boolean Difference method generates *all* tests for *every* fault in the circuit. No heuristics are involved, and the algorithm is complete, but of only theoretical interest because of vast amounts of computations and of data. The Boolean Difference algorithm identifies all redundant faults.

If a test exists, the D-algorithm generates *a* test for *every* fault in the circuit. The D-algorithm is complete and its efficiency relies on heuristics. It uses considerably less computation and storage space than the Boolean Difference, but program implementation for automatic test pattern generation (ATG) is unwieldy. The D-algorithm identifies all redundancies, but after an exhaustive search that requires many iterations and backtracks.

Employing only four linear passes through the circuit and linear storage space, VICTOR generates *a* test for *some* of the faults in the circuit. If a redundancy exists, VICTOR identifies it, but VICTOR also flags many hard-to-test irredundant faults as *potential redundancies*. The extensive use of the risk heuristic drastically reduces algorithm complexity and biases the results such that VICTOR always errs on the conservative side. Irredundant faults are considered redundant, risks predictions are high, and size predictions are low.

CHAPTER 5

VICTOR PROGRAM IMPLEMENTATION

5.1. INTRODUCTION

The global method for redundancy identification and test generation introduced in the previous chapter has been implemented in a program. This chapter presents the details of the VICTOR program implementation: the program structure, the files attached during program execution, and the data base underlying the testability operations. The last section analyzes program portability and language specifics.

5.2. PROGRAM STRUCTURE

The program consists of four modules: the input processor INPROC, the controllability calculation module CONTRL, the observability calculation module OBSERV, and the result processor REPROC. The main routine in each of the four modules contains information on the common blocks and files used in the module. Program VICTOR consists of about 4300 lines of ANSI FORTRAN 77 grouped into 72 routines as shown in Table 5.1 (the entire program source listing is given in Appendix 6).

5.2.1. Module INPROC

The interactive input processor INPROC allows the user to specify input and output file, it checks the input syntax, levelizes the circuit, and copies the analysis data to a set of files. The user specifies interactively the input

Program VICTOR modules	routines	lines
main program & setup routine	2	100
input processor	21	1400
controllability calculation	26	1300
observability calculation	12	700
result processor	11	800
Program VICTOR (total)	72	4300

Table 5.1 Routine and line count in program VICTOR.

file containing the circuit description and the output file to which the global results of program VICTOR are copied. Node name length, library cell name, and cell input/output node count are verified and error messages, if any, are displayed on the terminal. The names of the currently available cells in program VICTOR are defined in subroutine SETUP as shown in Figure 5.1. While checking the input syntax, INPROC sets up the node list and encodes circuit topology into a machine-readable form. Nodes are verified for correct connectivity, and floating nodes and wired logic (tied cell outputs) are detected and reported to the user. Module INPROC produces a levelized cell list and detects the presence of (illegal) feedback loops. The user is warned again, and the boundary of the circuit portion containing feedback loops is determined. Finally, INPROC calls a debugging subroutine NETBUG, to write out all circuit topology data to file VIC.BUG (the data serves only in program debugging and maintenance).

5.2.2. Modules CONTRL and OBSERV

The controllability and observability calculation modules, CONTRL and OBSERV, follow closely the algorithm described in the previous chapter. Since the program has been implemented at an early stage in the development of VICTOR, some terms used in the program differ from the ones

```
c-------------------------------------------------------------
c  library cell names and cell output and input node count
c-------------------------------------------------------------
      data
     + libr(1) /'and2  out=1 in=2'/,    libr(2) /'and3  out=1 in=3'/
     + libr(3) /'and4  out=1 in=4'/,    libr(4) /'and5  out=1 in=5'/
     + libr(5) /'and6  out=1 in=6'/,    libr(6) /'and7  out=1 in=7'/
     + libr(7) /'and8  out=1 in=8'/,    libr(8) /'and9  out=1 in=9'/

     + libr(9) /'aoi21 out=1 in=3'/,    libr(10)/'aoi22  out=1 in=4'/
     + libr(11)/'aoi31 out=1 in=4'/,    libr(12)/'buf   out=1 in=1'/
     + libr(13)/'inv   out=1 in=1'/,    libr(14)/'mux2 out=1 in=3'/

     + libr(15)/'nand2  out=1 in=2'/,   libr(16)/'nand3  out=1 in=3'/
     + libr(17)/'nand4  out=1 in=4'/,   libr(18)/'nand5  out=1 in=5'/
     + libr(19)/'nand6  out=1 in=6'/,   libr(20)/'nand7  out=1 in=7'/
     + libr(21)/'nand8  out=1 in=8'/,   libr(22)/'nand9  out=1 in=9'/

      data
     + libr(23)/'nor2  out=1 in=2'/,    libr(24)/'nor3  out=1 in=3'/
     + libr(25)/'nor4  out=1 in=4'/,    libr(26)/'nor5  out=1 in=5'/
     + libr(27)/'nor6  out=1 in=6'/,    libr(28)/'nor7  out=1 in=7'/
     + libr(29)/'nor8  out=1 in=8'/,    libr(30)/'nor9  out=1 in=9'/

     + libr(31)/'oai21  out=1 in=3'/,   libr(32)/'oai22  out=1 in=4'/
     + libr(33)/'oai31  out=1 in=4'/,   libr(34)/'oai33  out=1 in=6'/

     + libr(35)/'or2  out=1 in=2'/,     libr(36)/'or3  out=1 in=3'/
     + libr(37)/'or4  out=1 in=4'/,     libr(38)/'or5  out=1 in=5'/
     + libr(39)/'or6  out=1 in=6'/,     libr(40)/'or7  out=1 in=7'/
     + libr(41)/'or8  out=1 in=8'/,     libr(42)/'or9  out=1 in=9'/

     + libr(43)/'trag  out=1 in=2'/,    libr(44)/'xnor2  out=1 in=2'/
     + libr(45)/'xor2 out=1 in=2'/
```

Figure 5.1 Library cell name and cell output/input connection definition in subroutine SETUP.

introduced in Chapter 4. The following equivalences hold (terms used in the FORTRAN code on the left, terms defined and used in Chapter 4 on the right):

label <=> pattern
weight <=> risk
merge <=> intersect

Both CONTRL and OBSERV rely on the implementation of the testability equations of each library cell; two dedicated subroutines encode the controllability and the observability cell equations such that general operations are performed at compile time and just circuit specific calculations are executed at

run time.

For example, the controllability routine in module CONTRL for an OR-AND-INVERT cell is presented in Figure 5.2. After term initialization, the routine employs only two operations, SELECT and INTERSECT, to calculate the cell output controllability. Both operations are associative; hence, a two operand composition rule is enough to evaluate an arbitrary number of operands. Since the basic logic functions (AND, NAND, OR, and NOR) map directly into a single SELECT or INTERSECT operation, one generic routine for each handles any number of gate inputs. Cells of the same type but different number of inputs, e.g., NOR4, NOR5, NOR8, have been given different names merely to aid the syntax checker in verifying the correctness of the cell input/output connections.

Of special interest is the encoding of INTERSECT, the sole operation in VICTOR that creates new patterns. Consistent with the general algorithmic approach that the symbolic data manipulation described in Chapter 4 allows, the routine implementing the two-operand INTERSECT (see Figure 5.3) can be changed easily to implement a different operation. The composition table, a 4x4 array of 16 entries, is encoded in four CHARACTER DATA statements. If the algebraic structure in VICTOR is expanded from 4 variables to 16, the only required change in the program is to encode the corresponding 256 entries in the new 16x16 table.

In an effort closely related to VICTOR, the transformation from standard logic equations into control and monitor equations has been automated [Van Egmond 82]. With the aid of program LITE, the entire cell library in program VICTOR can be customized in a few hours.

```
*****************************************************************
            subroutine coai33
*****************************************************************
c  or-and-invert gate:  out=z,  in=a1,a2,,a3,b1,b2,b3 ;
c                              z=((a1+a2+a3).(b1+b2+b3))'

      parameter (lbnam=100, ndmax=10000, maxpi=120, kio=200)
      common /nodind/ list(ndmax),ndfout(ndmax),ndlev(ndmax)
      common /nodco1/ con0(ndmax),con1(ndmax),obs(ndmax),lab(kio)
      common /nodco2/ lut0(ndmax),lsiz0(ndmax),lut1(ndmax),lsiz1(ndmax),
     + luto(ndmax),lsizo(ndmax),kouts(kio),kins(kio),nout,nin,
     +  lut(kio),lsiz(kio),lev(kio)
      character*maxpi, con0, con1, obs, lab

      nodout=kouts(1)

c  for c0  (compute two intermediate labels)
      call interc('c1',kins(1),1,'oai22')
      call interc('c1',kins(2),2,'oai22')
      call interc('c1',kins(3),3,'oai22')

            call select(1,3,lab(10),lut(10),lsiz(10))

      call interc('c1',kins(4),4,'oai22')
      call interc('c1',kins(5),5,'oai22')
      call interc('c1',kins(6),6,'oai22')

            call select(4,6,lab(11),lut(11),lsiz(11))

c  compute c0
          call merge(10,11,con0(nodout),lut0(nodout),lsiz0(nodout))

c  for c1:  compute two intermediate labels
      call interc('c0',kins(1),1,'oai22')
      call interc('c0',kins(2),2,'oai22')
      call interc('c0',kins(3),3,'oai22')

            call merge(1,3,lab(10),lut(10),lsiz(10))

      call interc('c0',kins(4),4,'oai22')
      call interc('c0',kins(5),5,'oai22')
      call interc('c0',kins(6),6,'oai22')

            call merge(4,6,lab(11),lut(11),lsiz(11))

c  compute c1 (assume intermediate labels have level=1)
      lev(10)=1
      lev(11)=1
          call select(10,11,con1(nodout),lut1(nodout),lsiz1(nodout))

      return
      end
```

Figure 5.2 Controllability routine for cell OAI33.

```
*****************************************************************************
      subroutine merge2(lb1,w1,s1,lb2,w2,s2,lbmerg,wmerg,smerg)
*****************************************************************************
c  merge the two incoming labels lb1, lb2 (weight w1, w2, size s1, s2)
c     symbol by symbol and return merged label, weight, and size.

c                    symbol composition table:
c                 |  0     1     x     c
c                 ==========================
c                 0 |  0     c     0     c
c                 1 |  c     1     1     c
c                 x |  0     1     x     c
c                 c |  c     c     c     c


      parameter (maxpi=120)
      common /actsiz/ npiel,npoel,ndel,ndlog,ncell,ninput,nlevel

       character*maxpi, lb1, lb2, lbmerg
       integer w1, w2, wmerg, s1, s2, smerg
       logical clash

*****************************************************************************
*  4-valued logic is implemented in the following data statements
*     modify ONLY symbop and symlis for new logic system

       character*4 symbop(4), symlis
       data   symbop(1) /'0c0c'/,
     +        symbop(2) /'c11c'/,
     +        symbop(3) /'01xc'/,
     +        symbop(4) /'cccc'/
       data   symlis    /'01xc'/
*****************************************************************************


c  merge the two labels character by character & flag occurring clashes.
       clash = .false.
       do 10  l=1,npiel
              irow = index (symlis,lb1(l:l))
              jcol = index (symlis,lb2(l:l))
              lbmerg(l:l) = symbop(irow)(jcol:jcol)
              if (lbmerg(l:l) .eq. 'c')  clash = .true.
  10   continue

c  compute the weight and size of new label.
       if (clash)  then
              wmerg = 99999
              smerg = 0
       else
              wmerg = w1+w2
              smerg = s1*s2
       end if

       return
       end
```

Figure 5.3 Two operand intersection subroutine MERGE2.

5.2.3. Module REPROC

The result processor, REPROC, has no special implementation features. It uses the same INTERSECT operation for test pattern generation and test compaction, performs several shell sorts for alphabetical and numerical ordering of the fault data, and writes the output to several files.

5.3. FILE STRUCTURE

The user specified input and output files containing the circuit description and the final results, and another twenty scratch files are attached to program VICTOR during execution. Both names and logical unit (device) numbers assigned to the files are defined in subroutine SETUP, as shown in Figure 5.4. Files are used during input processing to log syntax errors and

```
c-------------------------------------------------------------
c  file structure: file names and logical unit assignment
c-------------------------------------------------------------

      data tmpfil /
     + 'vic.io',  'vic.syn',  'vic.flo',  'vic.fb',  'vic.net',
     + 'vic.red',  'vic.vec',  'vic8',  'vic9',  'vic10',
     + 'vic11',  'vic12',  'vic13',  'vic14',  'vic15',
     + 'vic16',  'vic17',  'vic18',  'vic19',  'vic20'/

c  tmpfil(1) (vic.io):   user-specified input and output file names
c  tmpfil(2) (vic.syn):  circuit description in standard form
c  tmpfil(3) (vic.flo):  floating nodes and wired logic
c  tmpfil(4) (vic.fb):   boundary of feedback loop area
c  tmpfil(5) (vic.net):  machine-readable circuit net list
c  tmpfil(6) (vic.red):  potentially redundant faults and aborted
c                           test vectors
c  tmpfil(7) (ic.vec):  test vectors for guaranteed irredundant faults
c  tmpfil(8)-tmpfil(20) (vic8-vic20):  spare files for future extensions

c  device number assignments for vax/unix (machine dependent):
c     1-4  = input files;  5 = standard input (from terminal)
c     7-10 = output files; 6 = standard output (to terminal)

      data lu /1,2,3,4,5,6,7,8,9,10/
```

Figure 5.4 File structure definition in subroutine SETUP.

during result processing to store the output; examples are given in the next chapter.

5.3.1. File Name and Circuit Description Files

VIC.IO The file contains the user-specified input and output file names, one per line.

VIC.SYN The file contains the circuit description in standard form obtained from the initial circuit description by deleting comment lines, blank lines, leading blanks, and the characters right of the line continuation sign (plus), by substituting a single blank for all separators (tab, blank, comma, colon, semicolon, parentheses) and by preceding each line with a current line number.

VIC.NET The file contains the circuit node and topology information in the machine readable form used by modules CONTRL and OBSERV and serves only for debugging.

5.3.2. Connection Error Files

VIC.FLO The file contains floating nodes, incorrectly connected primary input and output nodes, tied cell output nodes (wired logic), and warning messages of possible redundancy for wired logic at the output of identical cells.

VIC.FB The file contains the boundary of the feedback loop region, i.e., the highest level cells that feed cells with infinite level.

5.3.3. Fault Information Files

VIC.RED The file contains an alphabetic list of all potentially redundant faults with the corresponding aborted test patterns (patterns contain at least one clash).

VIC.VEC The file contains information on the faults guaranteed to be irredundant: an alphabetic list of faults with their corresponding test patterns and the global fault coverage, test patterns for the collapsed faults and compacted test patterns listed in decreasing order of the number of detected faults per vector and the per cent test data reduction, and a histogram of the fault coverage and the cumulative fault coverage per vector.

output This user-specified output file contains the global results: circuit data and fault data synopses, alphabetic list of all faults (for fanout nodes, the root precedes the branches) and their corresponding triplets of pattern, risk, and size for each of the set, reset, and monitor operations.

5.4. DATA STRUCTURE

The data base in program VICTOR is composed of eleven common blocks grouped in several functional categories. The common blocks used in each of the four program modules are listed in the main calling routine of the module. To allow for future expansion, the dimensions of the arrays and the lengths of some key character variables are adjustable, as shown in Figure 5.5. Overflow protection is enforced throughout the program: an error message specifying the violated dimension or length of a variable is displayed, and execution is terminated.

```
c-----------------------------------------------------------------
c  define the variable array sizes. all parameter statements in
c     the entire program must be changed if these sizes are changed.
c-----------------------------------------------------------------
c  libnum:  number of predefined library cells
c  ndmax:   maximum number of nodes in the circuit
c  maxpi:   maximum number of primary input nodes in the circuit
c  kio:     maximum number of input/output nodes per cell

      parameter (libnum=100, ndmax=10000, maxpi=120, kio=200)

c !!! do not change lengths of the following character variables !!!
c   unless you are willing to update all occurrences in the program.

      character*maxpi, con0, con1, obs, lab, veclis
      character*40 tmpfil, inpf, outf, libr*30, lognod*72
```

Figure 5.5 Adjustable variable sizes in subroutine SETUP.

A description of the various common blocks extracted directly from subroutine SETUP follows.

5.4.1. File Name Data

```
      common /lulist/ lu(10)
c  lu(10):    device number assignment

      common /ukfile/ tmpfil(20)
c  tmpfil(20): temporary work files

      common /iofile/ inpf, outf
c  inpf, outf:       user-specified input and output file names
```

5.4.2. Circuit Node Data

```
      common /nodnam/ libr(libnum), lognod(ndmax)
c  libr(libnum):            library cell names and output/input node count
c  lognod(ndmax):    names of circuit nodes and composite fanout
c                          branch names

      common /nodind/ list(ndmax),ndfout(ndmax),ndlev(ndmax)
c  list(ndmax):       scratch array for general storage:
c                          node fanin in module INPROC
c                          fanout node root index in module OBSERV
c                          sorted array pointers in module REPROC
c  ndfout(ndmax):     node fanout
c  ndlev(ndmax):      node level
```

5.4.3. Circuit Topology Data

```
      common /ckttop/ inpckt(2*ndmax), levord(ndmax/2)
c  inpckt(2*ndmax): machine-readable circuit description as a
c                   collection of standard cell entries of the form:
c                         0
c                         cell index (libr array index of cell)
c                         negative cell output node index
c                                  .............
c                         negative cell output node index
c                         positive cell input node index
c                                  .............
c                         positive cell input node index
c                   where cell index = libr array index of cell name
c                   and node index = lognod array index of node name

c  levord(ndmax/2): levelized cell list:  list of inpckt indexes
c                   corresponding to cells in ascending order
c                   of their level

      common /actsiz/ npiel,npoel,ndel,ndlog,ncell,ninput,nlevel
c  npiel,npoel:           number of primary input and primary output nodes
c  ndel,ndlog:            number of (electrical) nodes and fault locations
c                         (logical nodes) in the circuit
c  ncell:           number of circuit cells
c  ninput:          element count in array inpckt
c  nlevel:          number of circuit levels
```

5.4.4. Node Controllability/Observability Data

```
      common /nodco1/ con0(ndmax),con1(ndmax),obs(ndmax),lab(kio)
c  con0(ndmax):           node 0-controllability label (reset pattern)
c  con1(ndmax):           node 1-controllability label (set pattern)
c  obs(ndmax):            node observability label (monitor pattern)
c  lab(kio):         scratch pad array for label calculations

      common /nodco2/ lut0(ndmax),lsiz0(ndmax),lut1(ndmax),lsiz1(ndmax),
     + luto(ndmax),lsizo(ndmax),kouts(kio),kins(kio),nout,nin,
     +  lut(kio),lsiz(kio),lev(kio)
c  lut0(ndmax),lsiz0(ndmax):      reset (0) weight and size
c  lut1(ndmax),lsiz1(ndmax):      set (1) weight and size
c  luto(ndmax),lsizo(ndmax):      monitor weight and size
c  kouts(kio),kins(kio):   cell output and input node indexes
c  nout,nin:               cell output and input node counts
c  lut(kio),lsiz(kio),lev(kio):   scratch weight, size, and level arrays
c                              for cell testability calculations
```

5.4.5. Test Data

```
      common /tesvec/ veclis(ndmax)
c  veclis(ndmax):    sorted test vectors in decreasing order of
c                         the detected faults per vector
```

```
      common /actest/ nirred,nred,nfcol
c  nirred,nred,nfcol:      number of guaranteed irredundant and potentially
c                          redundant faults
c  nfcol:            irredundant fault count after fault collapsing
```

5.5. PROGRAM PORTABILITY

Program portability is the property of a program to compile and execute on a variety of computers. Ideally, no change shoud be necessary in the original program code, but if only minor changes are required, the program is still considered to be portable. However, the necessary changes should be small, easy to identify, and localized in a few subroutines. The choice of the programming language and the way it is used are the two determining factors in the portability of a program.

The goal of the program implementation for VICTOR has been source code portability, and to attain it, speed and memory performance of the program have been sacrificed. The current VICTOR program implementation is given as a proof of method and should be used as a prototype together with the benchmark examples given in Chapter 6. The input/output processing routines must be extended and augmented, and some core subroutines implementng the testability calculations must be rewritten using an efficient programming language and dynamic memory management to achieve the performance expected from a software product.

5.5.1. Choice of FORTRAN 77

The selection of a programming language for a given application follows a set of prioritized criteria. Newton [Newton 81] compares several programming languages while searching for a "blue collar language for CAD" and lists the employed decision factors: ease of algorithms translation, portability, execu-

tion efficiency, and maintenance. His candidates are PASCAL, C, and FORTRAN preprocessors such as RATFOR.

For VICTOR, the major criterion has been code portability: standardization and availability of the the programming language. Four languages, LISP, PASCAL, C, and FORTRAN 77, have been investigated from this angle. LISP is nonstandard and nonportable, PASCAL has many different dialects, and C lacks a standard document defining the language. FORTRAN 77 has been chosen over C (in spite of the latter's superior control structures and overall qualities), because FORTRAN 77 is available on practically any computer, and an ANSI document exists with the specific goal of promoting "portability of FORTRAN programs for use on a variety of data processing systems" [ANSI 78]; FORTRAN 77 is actually ANSI X3.9-1978 FORTRAN.

5.5.2. Program VICTOR Language

Language use in program VICTOR ows much to the experience gained by implementing the SCOAP algorithm [Goldstein 79] in FORTRAN 77 for use on a variety of systems within Bell Laboratories (see Appendix 1). The lessons learned from SCOAP make up the implementation philosophy employed for VICTOR: abide by the rules, push for legal performance, and document everything.

The code follows the FORTRAN ANSI reference (any possible inconsistency is unintentional), even if the writing of such code has required considerable effort. For example, only two structured control statements, block IF and computed GO TO, are used in the entire program. Machine-dependent code (file names, device number allocation, variable array dimensions, character variable length) is localized in a BLOCK DATA subroutine SETUP (Appendix 5),

which also describes the entire program data base. As a rule, the code does not exploit any machine idiosyncrasy. For example, the routine that converts integers represented as characters to their numeric value, subroutine ATOI (see Appendix 6), consists solely of FORTRAN 77 statements.

The forte of FORTRAN 77, character variables, constitutes the core of many operations in the VICTOR program implementation. During input processing, line parsing uses the character INDEX function to check for input syntax errors and to set up the circuit node list. INDEX is used again as the basic operation in the evaluation of pattern intersection, as shown in Figure 5.3. All patterns — set, reset, monitor, and test — are character variables; therefore, bit packing occurs at compile time and does not depend on machine word length. The inherent 8-bit ASCII encoding allows for ample (up to 256 logic values) future expansion of the symbolic algebra used in VICTOR.

The VICTOR program implementation includes the necessary documentation on its operation in the source code. About one out of every five lines in the program is a comment line explaining the next few lines of code, and every routine starts with a description of the operations performed by the routine. Detailed information about the data structure, file structure, and library cell names is given in subroutine SETUP. The information in the code should suffice for program maintenance and extension.

CHAPTER 6

VICTOR PERFORMANCE EVALUATION

6.1. INTRODUCTION

The performance of VICTOR, the approach to global redundancy identification and test generation, is evaluated in this chapter. First, the correctness of the approach is shown by analyzing several small pathological circuits. Then, program performance is measured for the 74181 4-bit ALU and for an industrial example. Finally, program VICTOR is compared to testability analysis program SCOAP.

6.2. METHOD CORRECTNESS

The benchmarks chosen to verify method correctness in VICTOR are the the circuits examples illustrating the three different redundancy types.

6.2.1. Uncontrollable and Unobservable Redundancy

The first circuit (Figure 6.1) has been analyzed in Chapter 4. VICTOR generates a list of potential redundancies that include not only the 1-uncontrollable node K, but also faults A-1-J/1, A-1-K/0, J/0, A/0, and A/1. As shown before, the redundancy estimate is exact: all potential redundancies are true redundancies. It is important that VICTOR identify the fanout root faults, A/0 and A/1, as redundant, since the circuit falls under the conditions of Armstrong's analysis (simple fanout convergence with unequal inversion parity, see Chapter 3) and fault propagation from the root of the fanout node

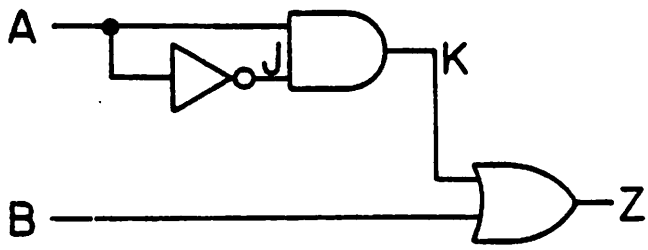


Figure 6.1 Redundant circuit containing 1-uncontrollable fault K/0.

stops at node K, the convergence point. In the second circuit (Figure 6.2), VICTOR identifies faults A/0, A/1, B-2-J/0, and J/0 as potentially redundant, and the faults are actually redundant; hence, the result is 100% correct.

The analysis of the simple examples described above shows that all signal dependencies in the circuit must be taken into acount. To that end, the SCOAP testability analysis algorithm [Goldstein 79 & 80] is applied to the circuit in Figure 6.1. SCOAP calculates a node 0-controllability, CC0, and a node 1-

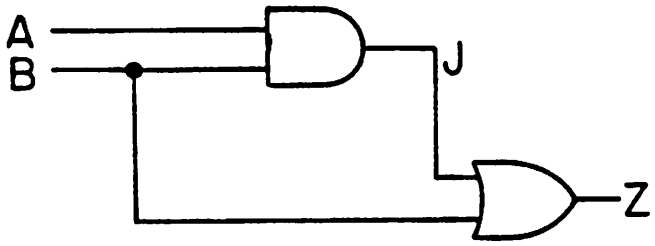


Figure 6.2 Redundant circuit containing unobservable faults A/0 and A/1.

controllability, CC1, as the number of constrained nodes on the path from the primary inputs to the node, such that the node is reset or set. For every cell traversed by the path, controllability is incremented by one, and the path is chosen such that the lowest controllability value is propagated. A simple calculation yields the following controllabilities:

CC0	CC1	Node
1	1	A
1	1	B
2	2	J
1	3	K
2	2	Z

The values for node J are obvious, and for node K the controllabilities are calculated below:

$$CC\ 0(K) = min\ (CC\ 0(A),\ CC\ 0(J)) = min\ (1,2) = 1$$

$$CC\ 1(K) = CC\ 1(A) + CC\ 1(J) = 1+2 = 3$$

SCOAP predicts three node constraints for K=1, but fails to recognize that condition A=1 and J=1 is impossible, since node J depends on node A. This error is not just a slight inaccuracy, but a gross mistake, since a redundant fault is predicted to be irredundant. Many such mistakes are present during the SCAOP evaluation, because cell inputs are assumed independent of each other. A similar SCOAP analysis of the second example incorrectly predicts all four redundant faults to be testable. In general, any testability analysis program that does not take into account signal dependencies does not generate reliable fault data.

6.2.2. Schneider's Example

Schneider's example (Figure 6.3) contains two redundancies, the untestable-controllable-observable faults B-1-K/0 and C-2-K/0. The circuit

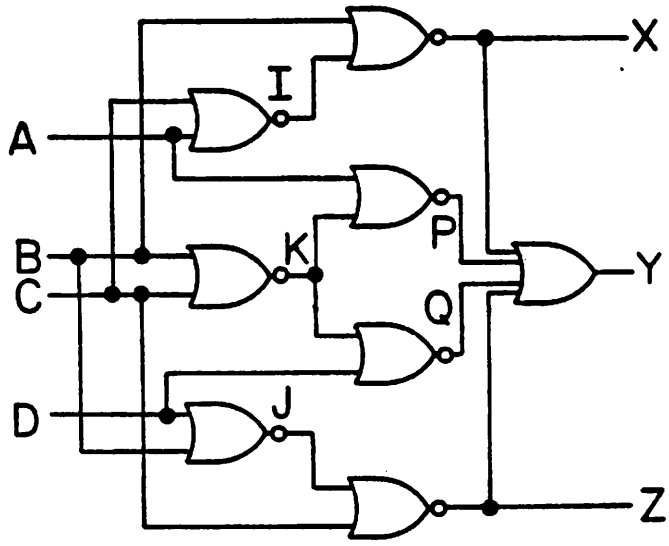


Figure 6.3 Redundant circuit containing untestable-controllable-observable faults B-1-K/0 and C-2-K/1.

description serving as input for program VICTOR is:

```
input  a b c d
output x y z
*
nor2 i, c a
nor2 k, b c
nor2 j, d b
nor2 x, b i
nor2 p, a k
nor2 q, k d
nor2 z, j c
nor4 y, x p q z
```

The program puts out a synopsis of circuit and fault data,

FAULT DATA		*CIRCUIT DATA*	
total single-stuck faults	: 52	*primary inputs*	: 4
faults on fanout branches	: 28	*primary outputs*	: 3
(54% of total)		*circuit nodes*	: 12
potentially redundant faults:	7	*circuit cell count*	: 8
(13% of total)		*circuit level count:*	3

and then prints out an alphabetically ordered list of fault locations with their set, reset, and monitor triplets.

LEGEND:

```
> SET, RESET, MONITOR:  node testability triplets
> 123...., R, S:  pattern, risk, and size of a triplet
> 0, 1, x (don't care), # (clash):  PI values
> monitor pattern containing only clashes indicate
    one of the following:
        * floating nodes and their predecessors
            (check floating node file vic.flo)
        * unobservable fanout roots and predecessors
        * clashes on all primary inputs
> order of primary inputs in a pattern:
            1.  a
            2.  b
            3.  c
            4.  d
```

SET			RESET			MONITOR			
1234	R	S	1234	R	S	1234	R	S	NODE
1xxx	6	1	0xxx	6	1	x111	6	3	a
1xxx	6	1	0xxx	6	1	x111	6	1	a-1-p
1xxx	6	1	0xxx	6	1	x00x	9	2	a-2-i
x1xx	9	1	x0xx	9	1	1xxx	0	4	b
x1xx	9	1	x0xx	9	1	11#0	99999	0	b-1-k
x1xx	9	1	x0xx	9	1	1xxx	0	2	b-1-x
x1xx	9	1	x0xx	9	1	xx00	8	2	b-2-j
xx1x	9	1	xx0x	9	1	xxx1	0	4	c
xx1x	9	1	xx0x	9	1	00xx	8	2	c-1-i
xx1x	9	1	xx0x	9	1	1#10	99999	0	c-2-k
xx1x	9	1	xx0x	9	1	xxx1	0	2	c-2-z
xxx1	6	1	xxx0	6	1	111x	6	3	d
xxx1	6	1	xxx0	6	1	x00x	9	2	d-1-j
xxx1	6	1	xxx0	6	1	111x	6	1	d-2-q
0x0x	15	1	1xxx	6	2	x0xx	6	2	i
x0x0	15	1	xxx1	6	2	xx0x	6	2	j
x00x	22	1	x1xx	13	2	1110	8	2	k
x00x	22	1	x1xx	13	2	1110	8	1	k-1-q
x00x	22	1	x1xx	13	2	0111	8	1	k-2-p
01xx	19	2	1xxx	6	2	x111	4	1	p
x1x0	19	2	xxx1	6	2	111x	4	1	q
10xx	17	2	x1xx	11	2	xxxx	0	2	x
10xx	17	2	x1xx	11	2	1x11	2	1	x-1-y
1111	34	16	10xx	17	8	xxxx	0	1	y
xx01	17	2	xx1x	11	2	xxxx	0	2	z
xx01	17	2	xx1x	11	2	11x1	2	1	z-4-y

Before the analysis of potential redundancies, the risk and size testability measures are investigated. For instance, setting node K imposes more checkpoint constraints than setting node I or J, and the set risk values indicate that. The risk measure does not increase monotonically with level, since the risk rectangle area may shrink when approaching the primary outputs. For example, the set risk of Q exceeds that of node X, a primary output node. A simple inspection of the circuit reveals that primary output node Y can be set in more ways than primary output node X or Y; the node set sizes reflect this discrepancy.

All node patterns are clash-free, except for B $-1-K_m$ and C $-2-K_m$. The full potential redundancy estimate of VICTOR is:

```
                POTENTIALLY REDUNDANT FAULTS

LEGEND:
> 0, 1, x (don't care), # (clash) - PI values
> clash for a PI indicates conflicting 0 and 1 requirements
        due to convergence, hence potential redundancy
> vectors containing only clashes indicate one of the following:
        * floating nodes and their predecessors
            (check floating node file vic.flo)
        * unobservable fanout roots and their predecessors
        * clashes on all primary inputs
> order of primary inputs in a test vector:
        1.  a
        2.  b
        3.  c
        4.  d
> left column:   test vectors for stuck-at-0 faults
> middle column: test vectors for stuck-at-1 faults
> right column:  name of stuck fault locations


1234          1234          FAULT LOCATION
====          ====          ==============
11#0          1##0          b-1-k
1#10          1##0          c-2-k
1##0                        k
1##0                        k-1-q
0##1                        k-2-p


SUMMARY:    52 possible faults
             7 faults are potentially redundant   ( 13%)
```

The VICTOR result contains five false redundancies, which are dealt with later.

Program VICTOR also generates test vectors for some of the redundant faults.

TEST VECTORS FOR IRREDUNDANT FAULTS

```
LEGEND:
> 0, 1, x (don't care) - primary input values
> order of primary inputs in a test vector:
        1.  a
        2.  b
        3.  c
        4.  d
> left column:   test vectors for stuck-at-0 faults
> middle column: test vectors for stuck-at-1 faults
> right column:  name of stuck fault locations
```

1234	1234	FAULT LOCATION
1111	0111	a
1111	0111	a-1-p
100x	000x	a-2-i
11xx	10xx	b
11xx	10xx	b-1-x
x100	x000	b-2-j
xx11	xx01	c
001x	000x	c-1-i
xx11	xx01	c-2-z
1111	1110	d
x001	x000	d-1-j
1111	1110	d-2-q
000x	10xx	i
x000	xx01	j
	1110	k
	1110	k-1-q
	0111	k-2-p
0111	1111	p
1110	1111	q
10xx	x1xx	x
1011	1111	x-1-y
1111	10xx	y
xx01	xx1x	z
1101	1111	z-4-y

```
SUMMARY:   52 possible faults
           45 faults are certainly irredundant  ( 87%)
```

These test patterns are collapsed, and a first test data reduction is achieved.

TEST VECTORS FOR COLLAPSED IRREDUNDANT FAULTS

```
SUMMARY:   17 test vectors for 45 irredundant faults
                (62% reduction)
> left column:     test vectors after fault collapsing
> right column:    number of detected faults per vector

           1111  ...    9
           10xx  ...    5
           1110  ...    5
```

```
0111  ...   4
xx01  ...   4
000x  ...   3
x000  ...   3
xx11  ...   2
11xx  ...   2
x100  ...   1
x1xx  ...   1
100x  ...   1
xx1x  ...   1
1011  ...   1
1101  ...   1
001x  ...   1
x001  ...   1
```

After test compaction, only nine test vectors remain.

COMPACTED TEST VECTORS FOR IRREDUNDANT FAULTS

```
SUMMARY:    9 test vectors for 45 irredundant faults
                 (80% reduction)
> leftmost column:    test vectors after test compaction
> center left column: number of detected faults per vector
> center right column: fault coverage per test vector
> rightmost column:    cumulative fault coverage per test vector

 1.  1111  ...   15    33.3%    33.3%
 2.  1001  ...   11    24.4%    57.8%
 3.  0000  ...    6    13.3%    71.1%
 4.  1110  ...    5    11.1%    82.2%
 5.  0111  ...    4     8.9%    91.1%
 6.  1011  ...    1     2.2%    93.3%
 7.  x100  ...    1     2.2%    95.6%
 8.  001x  ...    1     2.2%    97.8%
 9.  1101  ...    1     2.2%   100.0%
```

The above list gives the number of faults that the test vector is guaranteed to detect, although it may detect some other faults as well, and represents the final test generation result in the VICTOR program.

After having processed the irredundant faults, the investigation of the five potential redundancies identified by VICTOR is carried further. The compacted test vectors, listed previously in decreasing order of detected faults per vector, are simulated for faults. Test pattern 0000 detects all five false redundancies, and after exhausting all test patterns, only faults B-1-K/0 and C-2-K/0 are left as potential redundancies. Again, the VICTOR estimate is exact. In general, however, the set of potential redundancies includes many

false redundancies, the hard-to-test irredundant faults, as is shown in the next section.

6.3. PROGRAM PERFORMANCE

For the program performance evaluation, a medium size circuit example, the 74181 4-bit ALU, has been chosen. The circuit is large enough to allow for data explosion, but small enough (see the circuit diagram in Figure 6.4) to allow for hand analysis and fault evaluation. Also, detailed fault analysis data exists [Akers 82].

The circuit description of the 74181 ALU used by the VICTOR program implementation and the resulting fault testability information, i.e., the potentially redundant faults with the aborted test patterns and the irredundant faults with the corresponding valid test patterns, is included in Appendix 3. A synopsis of the fault data produced by program VICTOR is given below:

FAULT DATA		*CIRCUIT DATA*	
total single-stuck faults	*: 374*	*primary inputs*	*: 14*
faults on fanout branches	*: 220*	*primary outputs*	*: 8*
(59% of total)		*circuit nodes*	*: 77*
potentially redundant faults	*: 210*	*circuit cell count*	*: 63*
(56% of total)		*circuit level count*	*: 7*

It is known that the ALU is irredundant, thus all 210 potential redundancies are false ones. Following the same procedure as for Schneider's example, test generation in VICTOR yields 21 compacted test vectors, and the test data processing achieves a reduction of 87%.

COMPACTED TEST VECTORS FOR IRREDUNDANT FAULTS

SUMMARY: 21 test vectors for 164 irredundant faults
(87% reduction)
> leftmost column: test vectors after test compaction
> center left column: number of detected faults per vector

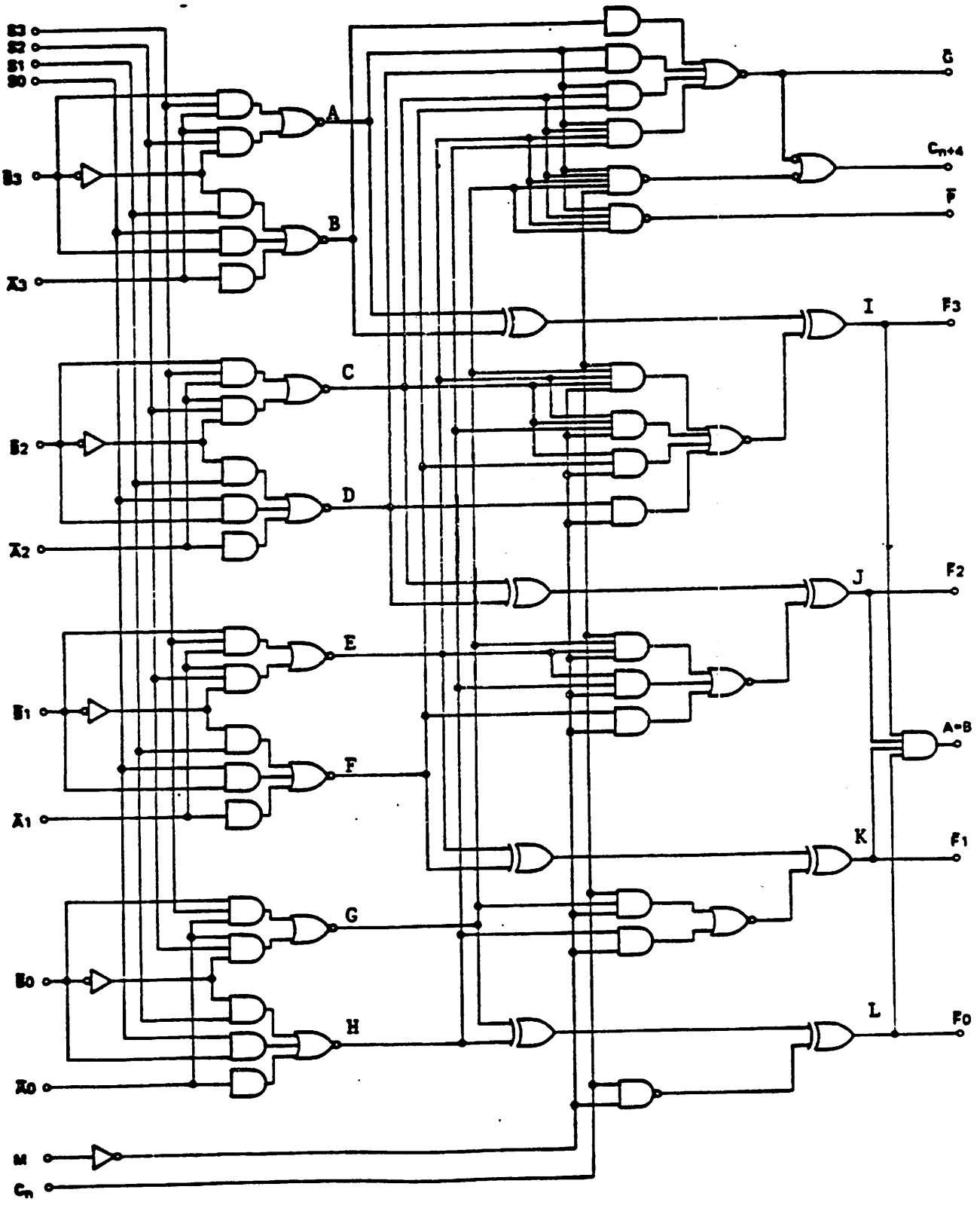


Figure 6.4 74181 ALU logic diagram

```
> center right column: fault coverage per test vector
> rightmost column:    cumulative fault coverage per test vector

 1.  1xxx1111111100  ...  33   20.1%   20.1%
 2.  1x0100x1111101  ...  26   15.9%   36.0%
 3.  110x1101000101  ...  17   10.4%   46.3%
 4.  1x0x1111001100  ...  15    9.1%   55.5%
 5.  110x0100110000  ...  15    9.1%   64.6%
 6.  110x110001000x  ...  14    8.5%   73.2%
 7.  1x1x00x1x11111  ...   7    4.3%   77.4%
 8.  x00110x1x10101  ...   6    3.7%   81.1%
 9.  1x0x1100x1x100  ...   6    3.7%   84.8%
10.  xx0x00000000x0  ...   6    3.7%   88.4%
11.  xx0x01x1x1x1xx  ...   3    1.8%   90.2%
12.  x00x01000100x   ...   3    1.8%   92.1%
13.  1x0x1100x1x110  ...   3    1.8%   93.9%
14.  xx0010x1x1x1xx  ...   2    1.2%   95.1%
15.  1x0x0000110010  ...   2    1.2%   96.3%
16.  1xxx11000000xx  ...   1    0.6%   97.0%
17.  1x0xxx1100x110  ...   1    0.6%   97.6%
18.  1xxx00110000xx  ...   1    0.6%   98.2%
19.  x00xxx0100xx0x  ...   1    0.6%   98.8%
20.  1xxx00000011xx  ...   1    0.6%   99.4%
21.  1x0xxx11110000  ...   1    0.6%  100.0%
```

Again, the compacted tests are simulated for faults, and a histogram indicating the success in eliminating potential redundancies per vector is given in Figure 6.5 (the values for the <o> and <#> symbols correspond to fault simulation ignoring don't cares and taking them into account, respectively).

The residue of this fault simulation is a kernel of 33 potential redundancies listed in Appendix 3. These faults are all false redundancies, but they include the critical faults [Akers 82], known to be hard to test. VICTOR effectively partitions the problem, and not the circuit; instead of having to generate tests for the 374 circuit faults, only 33 faults must be considered, which amounts to a reduction of the problem size by an order of magnitude.

Finally, the industrial example given in Appendix 4 has been analyzed by VICTOR. Lacking a fault simulator, potentially redundant faults cannot be eliminated by simulating the compacted test vectors by hand, therefore no final results could be obtained.

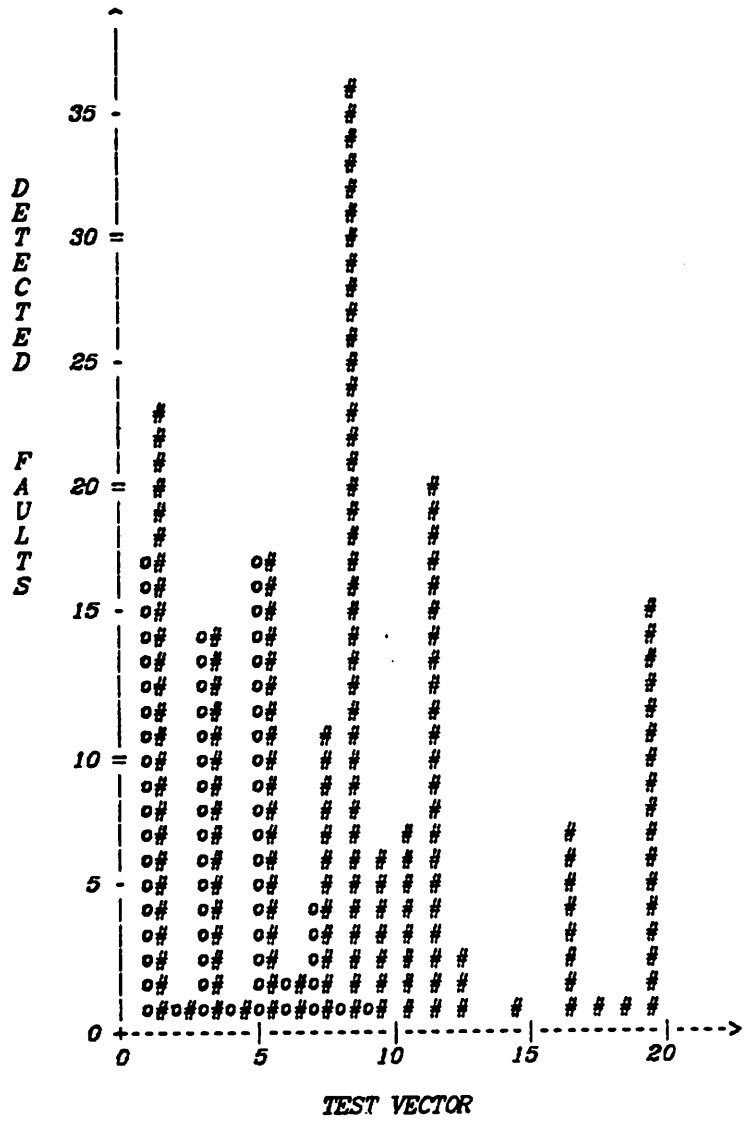


Figure 6.5 Histogram of eliminated false redundancies per test vector

To illustrate the performance of the VICTOR program implementation, the execution times in CPU seconds on a VAX/UNIX 11/780 computer of program VICTOR and program SCOAP are given for three circuits. The first circuit is Shneider's example (Figure 6.3), the second circuit is the 74181 ALU (Figure 6.4), and the third circuit is the industrial example listed in Appendix

4. The performance comparison between the two programs is given in the table below.

Faults	VICTOR	SCOAP
52	7	27
374	26	103
1728	172	1003

VICTOR is faster than SCOAP by about a factor of five, and as the circuit grows in size, this factor increases rapidly since the algorithm in SCOAP is quadratic. The linear operations employed by VICTOR keep computational costs down and require near-linear execution times. The deviation from linearity is caused by the three search procedures during input processing, fault collapsing, and test compaction (see Chapter 4).

CHAPTER 7

CONCLUSIONS

The complexity of current VLSI circuits and the presence of redundant and hard-to-test irredundant faults render computer-aided test procedures prohibitively expensive, even if the circuits contain a scan path and are, therefore, scan testable. The established alternatives, redundancy identification and testability analysis methods, offer little relief: redundancy identification techniques do not handle general circuits, and testability analysis approaches do not identify redundancy because signal are assumed to be independent.

VICTOR offers a solution to the problem in the form of a linear complexity method for global redundancy identification and test generation for scan-testable VLSI circuits. Rather than partitioning the circuit, VICTOR partitions the test problem into separate control and monitor problems that are merged later. VICTOR identifies all redundant and hard-to-test irredundant faults in a general combinational circuit and generates test vectors for most irredundant faults, then collapses and compacts the tests by about an order of magnitude. The algorithm requires only four passes through the fault list, and the algorithm data and computational complexity grows linearly with circuit size and primary input count.

The VICTOR approach relies on a four-valued algebraic structure that can be easily expanded up to 256 logic values. The primitive testability operations, pattern selection and intersection, allow for a direct translation from cell logic equations to cell testability (control and monitor) equations, which

in turn are mapped directly into program code, one routine per cell. Thus, the cell library can be customized in a few hours.

A prototype program for VICTOR has been written in ANSI FORTRAN 77. To achieve portability, program performance (speed and memory) have been sacrificed. The intent has been to provide a prototype program that executes on a variety of computers. If portability can be sacrificed, however, the performance of the current implementation can be substantially improved by rewriting some key routines in an efficient programming language and by using dynamic memory management for the data structures. Furthermore, the input/output processing requires major upgrading if program VICTOR is to be used as a production software tool. The code contains much documentation as comment lines (about one out of every five lines in the program) and it is hoped that it is sufficient for maintenance and further development.

VICTOR can be used as an independent tool but is limited by the uncertainty in the potential redundancy estimate. If a fault simulator is available, the uncertainty can be reduced significantly by simulating the already generated and compacted test patterns, and a high fault coverage can be obtained.

VICTOR, the global approach to redundancy identification and test generation, can be viewed as a component of an integrated computer-aided test system and can be extended in several directions. A simple, serial fault simulation capability can be built into VICTOR and one more global operation performed after test generation. Once the uncertainty of the redundancy estimate is reduced this way, the testability calculation in VICTOR can be repeated. This time, though, advance information on potential redundancies exists and is used to guide the pattern selection. Since the procedures in

VICTOR are global, even this iterated approach is still linear with circuit size. Finally, the required number of such iterations and the achieved fault coverage needs further investigation on a statistically relevant number of industrial VLSI circuits.

APPENDIX 1

The Berkeley FORTRAN 77 Version of SCOAP

Appendix 1 presents the history of the development of the Berkeley FORTRAN 77 version of Dr. Lawrence Goldstein's testability analysis program SCOAP.

SCOAP is a testability analysis program developed by Dr. Lawrence Goldstein at SANDIA National Laboratories [Goldstein 79 & 80]. Program SCOAP is written in FORTRAN 66, executes on a DEC 20 computer, and includes a large portion of nonportable code.

The author has received the source code from Dr. Goldstein early in 1980. After a few unsuccessful attempts to adapt the program to the VAX/UNIX computer environment, the author has decided to rewrite the entire program in ANSI FORTRAN 77. While a summer visitor at Bell Laboratories, the author has developed the FORTRAN 77 version of SCOAP. The author gratefully acknowledges Dr. Goldstein's expert advice and the excellent research environment he enjoyed at Bell Laboratories, environment made possible by Dr. Bernard Murphy, Dr. Wesley Grant, Dr. Hermann Gummel, and Dr. Ajoy Bose.

The FORTRAN 77 version of SCOAP implements exactly the original algorithm and contains about 3000 lines of portable code. The program runs on a 32-bit machine, the VAX 11/780, under either the UNIX or the VMS operating system, and on a 24-bit machine, Harris 570, under the VULCAN operating system.

After returning to the University of California, Berkeley, the author has continued to develop the FORTRAN 77 version of SCOAP. The comments of Mr. Bob Hess of UTMC and of Mr. Steve Menzel of Carleton University have been most helpful and are gratefully acknowledged. In April 1981, the author has started to release the Berkeley version of SCOAP. Currently, more than 35 copies of the Berkeley FORTRAN 77 version of SCOAP have been sent to various industrial and academic locations.

APPENDIX 2

VICTOR Cell Library

Appendix 2 presents the list of names and logic equations of the cells predefined in the VICTOR controllability and observability cell library.

general <and> gate: out=z, in=a,b,c,...
z=a.b.c...

<and-or-invert> gate: out=z, in=a1,a2,b
z=(a1.a2 + b)'

<and-or-invert> gate: out=z, in=a1,a2,b1,b2
z=(a1.a2 + b1.b2)'

<and-or-invert> gate: out=z, in=a1,a2,a3,b
z=(a1.a2.a3 + b)'

<signal buffer>: out=z, in=a
z=a

<signal inverter>: out=z, in=a
z=a'

2-input <mux>: out=z, in=d1,d2,c
z=d1.c' + d2.c

general <nand> gate: out=z, in=a,b,c,...
z=(a.b.c...)'

general <nor> gate: out=z, in=a,b,c,...
z=(a+b+c+...)'

<or-and-invert> gate: out=z, in=a1,a2,b
z=((a1+a2).b)'

<or-and-invert> gate: out=z, in=a1,a2,b1,b2
z=((a1+a2).(b1+b2))'

<or-and-invert> gate: out=z, in=a1,a2,a3,b
z=((a1+a2+a3).b)'

<or-and-invert> gate: out=z, in=a1,a2,a3,b1,b2,b3
z=((a1+a2+a3).(b1+b2+b3))'

general <or> gate: out=z, in=a,b,c,...
z=a+b+c+...

2 input <xnor> gate: out=z, in=a,b
z=a.b + a'.b'

2 input <xor> gate: out=z, in=a,b
z=a.b' + a'.b

APPENDIX 3

Analysis of the 74181 4-bit ALU

Appendix 3 presents the circuit description and the VICTOR results for the 74181 4-bit ALU example analyzed in Chapter 6.

```
*  74181 ALU   (4 bit arithmetic logic unit)
*     circuit description
inputs  S3 S2 S1 S0  B3N A3N B2N A2N B1N A1N B0N A0N M CN
outputs GN CN4 PN F3N F2N A=B F1N F0N
*
inv B3 B3N
inv B2 B2N
inv B1 B1N
inv B0 B0N
inv MN M
*
and3  1,    B3N   S3    A3N
and3  2,    A3N   S2    B3
and2  3,    B3    S1
and2  4,    S0    B3N
buf   5,    A3N
nor2  A,    1     2
nor3  B,    3     4     5
*
and3  6,    B2N   S3    A2N
and3  7,    A2N   S2    B2
and2  8,    B2    S1
and2  9,    S0    B2N
buf   10,   A2N
nor2  C,    6     7
nor3  D,    8     9     10
*
and3  11,   B1N   S3    A1N
and3  12,   A1N   S2    B1
and2  13,   B1    S1
and2  14,   S0    B1N
buf   15,   A1N
nor2  E,    11    12
nor3  F,    13    14    15
*
and3  16,   B0N   S3    A0N
and3  17,   A0N   S2    B0
and2  18,   B0    S1
and2  19,   S0    B0N
buf   20,   A0N
nor2  G,    16    17
nor3  H,    18    19    20
*
xor2  36,   A     B
xor2  42,   C     D
xor2  47,   E     F
xor2  51,   G     H
*
buf   31,   B
and2  32,   A     D
and3  33,   A     C     F
and4  34,   A     C     E     H
nand5 35,   A     C     E     G     CN
nand4 PN,   A     C     E     G
nor4  GN,   31    32    33    34
*
and5  38,   CN    G     E     C     MN
and4  39,   E     C     H     MN
```

```
and3  40,   C     F     MN
and2  41,   D     MN
nor4  37,   38    39    40    41
*
and4  44,   CN    G     E     MN
and3  45,   E     H     MN
and2  46,   F     MN
nor3  43,   44    45    46
*
and3  49,   CN    G     MN
and2  50,   H     MN
nor2  48,   49    50
*
nand2 52,   CN    MN
*
nand2       CN4,  GN    35
xor2  F3N,  36    37
xor2  F2N,  42    43
xor2  F1N,  47    48
xor2  F0N,  51    52
and4  A=B,  F3N   F2N   F1N   F0N
```

POTENTIALLY REDUNDANT FAULTS

LEGEND:

- > 0, 1, x (don't care), # (clash) - primary input values
- > clash for a primary input indicates conflicting 0 and 1 requirements due to convergence, hence potential redundancy
- > vectors containing only clashes indicate one of the following:
  - • floating nodes and their predecessors (check floating node file vic.flo)
  - • unobservable fanout roots and their predecessors
  - • clashes on all primary inputs
- > order of primary inputs in a test vector:
  1. S3
  2. S2
  3. S1
  4. S0
  5. B3N
  6. A3N
  7. B2N
  8. A2N
  9. B1N
  10. A1N
  11. B0N
  12. A0N
  13. M
  14. CN
- > left column: test vectors for stuck-at-0 faults
- > middle column: test vectors for stuck-at-1 faults
- > right column: name of stuck fault locations

12345678901234	12345678901234	FAULT LOCATION
1x0x1#00xxxx0x	xx0x0#00xxxx0x	1
1x0xxx#100xx0x	1x0xxx##00xx0x	10
1x0xxxxx1#000x	xx0xxxxx0#000x	11
	xx0xxxxx0#000x	12
1x#xxxxx##000x	1x0xxxxx##000x	13
1x01xxxx1#000x	1x0xxxxx##000x	14
1x0xxxxx#1000x	1x0xxxxx##000x	15
1xxxxxxxxx1#01	xxxxxxxxxx0#01	16
	xxxxxxxxxx0#01	17
1x1xxxxxxx##01	1x0xxxxxxx##01	18
1x01xxxxxx1#01	1x0xxxxxxx##01	19
	xx0x0#00xxxx0x	2
1x0xxxxxxx#101	1x0xxxxxxx##01	20
xx0x0#00x1x1xx		32
xx0x0#0#00x1xx		33
xx0x0#0#0#00xx		34
	xxxx0#0#0#0#x1	35
1x0x##00xxxx0x		36
1xxx110#0#0#01		38
1x0x110#0#0000		39
1x0x110#00x100		40
1x0xxx##00xx0x		42
1xxxxx110#0#01		44
1x0xxx110#0000		45
1x0xxxxx##000x		47

1xxxxxxx110#01	49
1x0xxxxxxx##01	51
1x0xxx1#00xx0x	6
xx0xxx0#00xx0x	7
1x#xxx##00xx0x	8
1x01xx1#00xx0x	9
xx0x0#00xxxx0x	A
xx0x0#00x1x1xx	A-1-32
xx0x0#0#00x1xx	A-1-33
xx0x0#0#0#00xx	A-1-34
xxxx0#0#0#0#x1	A-1-35
xx0x0#00xxxx0x	A-1-36
x1xxxxxxxx0#01	A0N
x1xxxxxxxx0#01	A0N-1-17
1x0xxxxxxx#101	A0N-1-20
1xxxxxxxxx1#01	A0N-3-16
x10xxxxx0#000x	A1N
x10xxxxx0#000x	A1N-1-12
1x0xxxxx#1000x	A1N-1-15
1x0xxxxx1#000x	A1N-3-11
x10xxx0#00xx0x	A2N
1x0xxx#100xx0x	A2N-1-10
x10xxx0#00xx0x	A2N-1-7
1x0xxx1#00xx0x	A2N-3-6
x10x0#00xxxx0x	A3N-1-2
1x0x1#00xxxx0x	A3N-3-1
1x0x##00xxxx0x	B-2-36
x1xxxxxxxx#101	B0
1x1xxxxxxx##01	B0-1-18
x1xxxxxxxx#101	B0-3-17
x1xxxxxxxx#101	B0N
1xxxxxxxxx1#01	1xxxxxxxxx0#01 B0N-1-16
x1xxxxxxxx#101	B0N-1-B0
1x01xxxxxx1#01	1x01xxxxxx##01 B0N-2-19
	x10xxxxx#1000x B1
1x#xxxxx##000x	1x#xxxxx##000x B1-1-13
	x10xxxxx#1000x B1-3-12
x10xxxxx#1000x	B1N
1x0xxxxx1#000x	1x0xxxxx0#000x B1N-1-11
x10xxxxx#1000x	B1N-1-B1
1x01xxxx1#000x	1x01xxxx##000x B1N-2-14
	x10xxx#100xx0x B2
1x#xxx##00xx0x	1x#xxx##00xx0x B2-1-8
	x10xxx#100xx0x B2-3-7
x10xxx#100xx0x	B2N
1x0xxx1#00xx0x	1x0xxx0#00xx0x B2N-1-6
x10xxx#100xx0x	B2N-1-B2
1x01xx1#00xx0x	1x01xx##00xx0x B2N-2-9
	xx1x#0x1x1x1xx B3
	xx1x#0x1x1x1xx B3-1-3
	x10x#100xxxx0x B3-3-2
1x0x1#00xxxx0x	1x0x0#00xxxx0x B3N-1-1
xx1x#0x1x1x1xx	B3N-1-B3
xx0xxx0#00xx0x	C
1x0x110#00x100	C-1-40
xx0xxx0#00xx0x	C-1-42
xx0x0#0#00x1xx	1x0x0#1100x1xx C-2-33
xx0x0#0#0#00xx	1x0x0#110#00xx C-2-34

xxxx0#0#0#0#x1	1xxx0#110#0#x1	C-2-35
1x0x110#0#0000	1x0x11110#0000	C-2-39
1xxx110#0#0#01	1xxx11110#0#01	C-4-38
1xxx110#0#0#01	1xxx110#0#0#00	CN-1-38
1xxxxx110#0#01	1xxxxx110#0#00	CN-1-44
1xxxxxxx110#01	1xxxxxxx110#00	CN-1-49
xxxx0#0#0#0#x1	xxxx0#0#0#0#x0	CN-5-35
1x0xxx##00xx0x		D
xx0x0#00x1x1xx	xxxx0#x1x1x1xx	D-2-32
1x0xxx##00xx0x		D-2-42
xx0xxxxx0#000x		E
1x0x110#0#0000	1x0x110#110000	E-1-39
1x0xxx110#0000		E-1-45
xx0xxxxx0#000x		E-1-47
xx0x0#0#0#00xx	1x0x0#0#1100xx	E-3-34
xxxx0#0#0#0#x1	1xxx0#0#110#x1	E-3-35
1xxx110#0#0#01	1xxx110#110#01	E-3-38
1xxxxx110#0#01	1xxxxx11110#01	E-3-44
1x0xxxxx##000x		F
1x0x110#00x100	1xxx110#x1x100	F-2-40
1x0xxxxx##000x		F-2-47
xx0x0#0#00x1xx	xxxx0#0#x1x1xx	F-3-33
	1xxx111111110#	F0N-4-A=B
	1x0x111111##00	F1N-3-A=B
	1x0x1111##1100	F2N-2-A=B
	1x0x11##111100	F3N-1-A=B
xxxxxxxxxx0#01		G
xxxxxxxxxx0#01		G-1-51
1xxx110#0#0#01	1xxx110#0#1101	G-2-38
1xxxxx110#0#01	1xxxxx110#1101	G-2-44
1xxxxxxx110#01		G-2-49
xxxx0#0#0#0#x1	1xxx0#0#0#11x1	G-4-35
1x0xxxxxxx##01		H
1x0xxx110#0000	1xxxxx110#x100	H-2-45
1x0xxxxxxx##01		H-2-51
1x0x110#0#0000	1xxx110#0#x100	H-3-39
xx0x0#0#0#00xx	xxxx0#0#0#x1xx	H-4-34
1x0x110#00x100	1x0x110#00x110	MN-3-40
1x0xxx110#0000	1x0xxx110#0010	MN-3-45
1xxxxxxx110#01	1xxxxxxx110#11	MN-3-49
1x0x110#0#0000	1x0x110#0#0010	MN-4-39
1xxxxx110#0#01	1xxxxx110#0#11	MN-4-44
1xxx110#0#0#01	1xxx110#0#0#11	MN-5-38
1x01xxxx1#000x	1x00xxxx1#000x	S0-1-14
1x01xxxxxx1#01	1x00xxxxxx1#01	S0-1-19
1x01xx1#00xx0x	1x00xx1#00xx0x	S0-1-9
1x#xxxxx##000x	1x0xxxxx##000x	S1-2-13
1x1xxxxxxx##01	1x0xxxxxxx##01	S1-2-18
1x#xxx##00xx0x	1x0xxx##00xx0x	S1-2-8
##############	##############	S3
1x0x1#00xxxx0x	0x0x1#00xxxx0x	S3-2-1
1x0xxxxx1#000x	0x0xxxxx1#000x	S3-2-11
1xxxxxxxxx1#01	0xxxxxxxxx1#01	S3-2-16
1x0xxx1#00xx0x	0x0xxx1#00xx0x	S3-2-6

SUMMARY: 374 possible faults
210 faults are potentially redundant (56%)

TEST VECTORS FOR IRREDUNDANT FAULTS

LEGEND:

> 0, 1, x (don't care) - primary input values
> order of primary inputs in a test vector:

1. S3
2. S2
3. S1
4. S0
5. B3N
6. A3N
7. B2N
8. A2N
9. B1N
10. A1N
11. B0N
12. A0N
13. M
14. CN

> left column: test vectors for stuck-at-0 faults
> middle column: test vectors for stuck-at-1 faults
> right column: name of stuck fault locations

12345678901234	12345678901234	FAULT LOCATION
x10xxxxx01000x		12
x1xxxxxxxx0101		17
x10x0100xxxx0x		2
xx1x00x1x1x1xx	xx0x00x1x1x1xx	3
xx0x00x1x1x1xx	xxxxx1x1x1x1xx	31
	xxxxx1x1x1x1xx	32
	xxxxx1x1x1x1xx	33
	xxxxx1x1x1x1xx	34
xxxxx1x1x1x1x0		35
	1x0x1100xxxx0x	36
1xxx11x1x1x1x0	1x0x1100xxxx0x	37
	1xxx11x1x1x1x0	38
	1xxx11x1x1x1x0	39
xx0110x1x1x1xx	xx0x00x1x1x1xx	4
	1xxx11x1x1x1x0	40
1x0x1100x1x100	1xxx11x1x1x1x0	41
	1x0xxx1100xx0x	42
1xxxxx11x1x1x0	1x0xxx1100xx0x	43
	1xxxxx11x1x1x0	44
	1xxxxx11x1x1x0	45
1x0xxx1100x100	1xxxxx11x1x1x0	46
	1x0xxxxx11000x	47
1xxxxxxx11x1x0	1x0xxxxx11000x	48
	1xxxxxxx11x1x0	49
xx0x01x1x1x1xx	xx0x00x1x1x1xx	5
1x0xxxxx110000	1xxxxxxx11x1x0	50
	1xxxxxxxxx1101	51
1xxxxxxxxx11x0	1xxxxxxxxx1101	52
x10xxx0100xx0x		7
	1x0x1100xxxx0x	A
	1x0x1100x1x1xx	A-1-32
	1x0x1100xxxx0x	A-1-36

xxxx00000000xx	1xxx11000000xx	A-1-PN
x1xxxxxxxx0101		A0N
x1xxxxxxxx0101		A0N-1-17
x10xxxxx01000x		A1N
x10xxxxx01000x		A1N-1-12
x10xxx0100xx0x		A2N
x10xxx0100xx0x		A2N-1-7
xx0x01x1x1x1xx	xx0x00x1x1x1xx	A3N
x10x0100xxxx0x		A3N-1-2
xx0x01x1x1x1xx	xx0x00x1x1x1xx	A3N-1-5
1xxx11111111x0	1xxxxxxxxx1101	A=B
xx0x00x1x1x1xx	xxxxx1x1x1x1xx	B
xx0x00x1x1x1xx	xxxxx1x1x1x1xx	B-1-31
	1x0x1100xxxx0x	B-2-36
x1xxxxxxxx0101		B0
x1xxxxxxxx0101		B0-3-17
	x1xxxxxxxx0101	B0N
	x1xxxxxxxx0101	B0N-1-B0
x10xxxxx01000x		B1
x10xxxxx01000x		B1-3-12
	x10xxxxx01000x	B1N
	x10xxxxx01000x	B1N-1-B1
x10xxx0100xx0x		B2
x10xxx0100xx0x		B2-3-7
	x10xxx0100xx0x	B2N
	x10xxx0100xx0x	B2N-1-B2
xx1x00x1x1x1xx		B3
xx1x00x1x1x1xx		B3-1-3
x10x0100xxxx0x		B3-3-2
xx0110x1x1x1xx	xx0100x1x1x1xx	B3N
	xx1x00x1x1x1xx	B3N-1-B3
xx0110x1x1x1xx	xx0100x1x1x1xx	B3N-2-4
	1x0xxx1100xx0x	C
	1x0x111100x100	C-1-40
	1x0xxx1100xx0x	C-1-42
xxxx00000000xx	1xxx00110000xx	C-2-PN
1xxxxxxxxx1101	1xxxxxxxxx1100	CN
1xxxxxxxxx1101	1xxxxxxxxx1100	CN-1-52
xx0x00xxxxxxxx	xxxxx1x1x1x1x0	CN4
	1x0xxx1100xx0x	D
1x0x1100x1x100	1xxx11x1x1x100	D-1-41
	1x0xxx1100xx0x	D-2-42
	1x0xxxxx11000x	E
	1x0xxx11110000	E-1-45
	1x0xxxxx11000x	E-1-47
xxxx00000000xx	1xxx00001100xx	E-3-PN
	1x0xxxxx11000x	F
1x0xxx1100x100	1xxxxx11x1x100	F-1-46
	1x0xxxxx11000x	F-2-47
1xxxxxxxxx11x0	1xxxxxxxxx1101	F0N
1xxx11111111x0		F0N-4-A=B
1xxxxxxx11x1x0	1x0xxxxx11000x	F1N
1xxx11111111x0		F1N-3-A=B
1xxxxx11x1x1x0	1x0xxx1100xx0x	F2N
1xxx11111111x0		F2N-2-A=B
1xxx11x1x1x1x0	1x0x1100xxxx0x	F3N
1xxx11111111x0		F3N-1-A=B
	1xxxxxxxxx1101	G

	1xxxxxxxxx1101	G-1-51
	1xxxxxxx111101	G-2-49
xxxx00000000xx	1xxx0000001 1xx	G-4-PN
xxxxx1x1x1x1xx	xx0x00xxxxxxxx	GN
xxxxx1x1x1x1x0	xx0x00xxxxxxx0	GN-1-CN4
	1xxxxxxxxx1101	H
1x0xxxxx110000	1xxxxxxx11x100	H-1-50
	1xxxxxxxxx1101	H-2-51
1x0x1100x1x110	1x0x1100x1x100	M
1x0x1100x1x100	1x0x1100x1x110	MN
1x0x1100x1x100	1x0x1100x1x110	MN-2-41
1x0xxx1100x100	1x0xxx1100x110	MN-2-46
1x0xxxxx110000	1x0xxxxx110010	MN-2-50
1xxxxxxxxx1101	1xxxxxxxxx1111	MN-2-52
1xxx11xxxxxxxx	xxxx00000000xx	PN
xx0110x1x1x1xx	xx0010x1x1x1xx	S0
xx0110x1x1x1xx	xx0010x1x1x1xx	S0-1-4
xx1x00x1x1x1xx	xx0x00x1x1x1xx	S1
xx1x00x1x1x1xx	xx0x00x1x1x1xx	S1-2-3
x10x0100xxxx0x	x00x0100xxxx0x	S2
x10xxxxx01000x	x00xxxxx01000x	S2-2-12
x1xxxxxxxx0101	x0xxxxxxxx0101	S2-2-17
x10x0100xxxx0x	x00x0100xxxx0x	S2-2-2
x10xxx0100xx0x	x00xxx0100xx0x	S2-2-7

SUMMARY: 374 possible faults
164 faults are certainly irredundant (44%)

TEST VECTORS FOR COLLAPSED IRREDUNDANT FAULTS

SUMMARY: 48 test vectors for 164 irredundant faults
(71% reduction)
> left column: test vectors after fault collapsing
> right column: number of detected faults per vector

1xxxxxxxxx1101	...	11
xx0x00x1x1x1xx	...	10
x1xxxxxxxx0101	...	8
x10xxxxx01000x	...	8
x10xxx0100xx0x	...	8
1x0xxx1100xx0x	...	7
1x0xxxxx11000x	...	7
xxxxx1x1x1x1xx	...	7
1xxx11x1x1x1x0	...	6
1x0x1100xxxx0x	...	6
xx1x00x1x1x1xx	...	6
x10x0100xxxx0x	...	5
xx0110x1x1x1xx	...	5
1x0x1100x1x100	...	5
1xxxxx11x1x1x0	...	5
xxxx00000000xx	...	5
1xxx11111111x0	...	5
1xxxxxxx11x1x0	...	4
1x0xxxxx110000	...	3
xxxxx1x1x1x1x0	...	3
1x0x1100x1x110	...	3
1x0xxx1100x100	...	3
xx0x01x1x1x1xx	...	3
xx0010x1x1x1xx	...	2
xx0x00xxxxxxxx	...	2
xx0100x1x1x1xx	...	2
x00x0100xxxx0x	...	2
1xxxxxxxxx1100	...	2
1xxxxxxxxx11x0	...	2
x00xxx0100xx0x	...	1
1xxxxxxx111101	...	1
x00xxxxx01000x	...	1
1xxxxxxxxx1111	...	1
1x0xxx11110000	...	1
1xxx00110000xx	...	1
1xxx00001100xx	...	1
1x0xxxxx110010	...	1
1xxx11x1x1x100	...	1
1xxx0000001lxx	...	1
1xxxxxxx11x100	...	1
1xxxxx11x1x100	...	1
1x0x111100x100	...	1
x0xxxxxxxx0101	...	1
xx0x00xxxxxxx0	...	1
1xxx11000000xx	...	1
1x0xxx1100x110	...	1
1xxx11xxxxxxxx	...	1
1x0x1100x1x1xx	...	1

COMPACTED TEST VECTORS FOR IRREDUNDANT FAULTS

SUMMARY: 21 test vectors for 164 irredundant faults
(87% reduction)

> leftmost column: test vectors after test compaction
> center left column: number of detected faults per vector
> center right column: fault coverage per test vector
> rightmost column: cumulative fault coverage per test vector

1.	1xxx1111111100	...	33	20.1%	20.1%
2.	1x0100x1111101	...	26	15.9%	36.0%
3.	110x1101000101	...	17	10.4%	46.3%
4.	1x0x1111001100	...	15	9.1%	55.5%
5.	110x0100110000	...	15	9.1%	64.6%
6.	110x110001000x	...	14	8.5%	73.2%
7.	1x1x00x1x11111	...	7	4.3%	77.4%
8.	x00110x1x10101	...	6	3.7%	81.1%
9.	1x0x1100x1x100	...	6	3.7%	84.8%
10.	xx0x00000000x0	...	6	3.7%	88.4%
11.	xx0x01x1x1x1xx	...	3	1.8%	90.2%
12.	x00x010001000x	...	3	1.8%	92.1%
13.	1x0x1100x1x110	...	3	1.8%	93.9%
14.	xx0010x1x1x1xx	...	2	1.2%	95.1%
15.	1x0x0000110010	...	2	1.2%	96.3%
16.	1xxx11000000xx	...	1	0.6%	97.0%
17.	1x0xxx1100x110	...	1	0.6%	97.6%
18.	1xxx00110000xx	...	1	0.6%	98.2%
19.	x00xxx0100xx0x	...	1	0.6%	98.8%
20.	1xxx00000011xx	...	1	0.6%	99.4%
21.	1x0xxx11110000	...	1	0.6%	100.0%

APPENDIX 4

Circuit Description of Industrial Example Circuit

Appendix 4 presents the gate-level description of the industrial example referenced in Chapter 6. The circuit has been obtained through the courtesy of the Siemens Corporation.

```
*****************************************************************
*  Industrial Circuit Example  (Siemens Circuit)
*****************************************************************

input 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 28 29 30  +
      31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50  +
      51 52 60 61 62 63 72 73 79 82 83 95 103   fake

output  23 25 27 53 55 57 59 75 78 81 97 99 104 169 183

*********************************************************
inv  faken, fake

* LEVEL 0 signal  l0
and2  l0, fake faken

* LEVEL 1 signal  l1
or2  l1, fake faken
*********************************************************

* 1QU       66    L0    65    L0    64
* 1qu  66 , l0 65 l0 64
nor2  1qu1 , l0 65
inv  1qu2 , 1qu1
nor2  1qu3 , l0 1qu2
nor2  1qu4 , 1qu1 64
nor2  66 , 1qu3 1qu4

* 1PT 67    XX    XX    48    L0    L0    L0    L0
* 1pt  67 xx xx , 48 l0 l0 l0 l0
inv  xx , 48
inv  67 , xx
or4  xx , l0 l0 l0 l0

* 2PT 68    XX    XX    50    L0    L0    L0    L0
* 2pt  68 xx xx , 50 l0 l0 l0 l0
inv  xx , 50
inv  68 , xx
or4  xx , l0 l0 l0 l0

* 3PT 69    XX    XX    49    L0    L0    L0    L0
* 3pt  69 xx xx , 49 l0 l0 l0 l0
inv  xx , 49
inv  69 , xx
or4  xx , l0 l0 l0 l0

* 4PT 70    XX    XX    51    L0    L0    L0    L0
* 4pt  70 xx xx , 51 l0 l0 l0 l0
inv  xx , 51
inv  70 , xx
or4  xx , l0 l0 l0 l0

* 5PT 71    XX    XX    52    L0    L0    L0    L0
* 5pt  71 xx xx , 52 l0 l0 l0 l0
inv  xx , 52
inv  71 , xx
or4  xx , l0 l0 l0 l0
```

```
* 1QX 53    XX    LO    LO    72    LO    LO    LO
* 1qx  53 xx ,  lo lo 72 lo lo lo
or4  53 , lo lo 72 lo
or2  xx , lo lo

* 2QX 55    XX    LO    LO    73    LO    LO    LO
* 2qx  55 xx ,  lo lo 73 lo lo lo
or4  55 , lo lo 73 lo
or2  xx , lo lo

* 2QU 75    71    LO    74    72
* 2qu  75 , 71 lo 74 72
nor2  2qu1 , 71 lo
inv  2qu2 , 2qu1
nor2  2qu3 , 74 2qu2
nor2  2qu4 , 2qu1 72
nor2  75 , 2qu3 2qu4

* 3QU 78    71    LO    77    76
* 3qu  78 , 71 lo 77 76
nor2  3qu1 , 71 lo
inv  3qu2 , 3qu1
nor2  3qu3 , 77 3qu2
nor2  3qu4 , 3qu1 76
nor2  78 , 3qu3 3qu4

* 4QU 81    71    LO    80    79
* 4qu  81 , 71 lo 80 79
nor2  4qu1 , 71 lo
inv  4qu2 , 4qu1
nor2  4qu3 , 80 4qu2
nor2  4qu4 , 4qu1 79
nor2  81 , 4qu3 4qu4

* 3QX 57    XX    LO    LO    82    LO    LO    LO
* 3qx  57 xx ,  lo lo 82 lo lo lo
or4  57 , lo lo 82 lo
or2  xx , lo lo

* 4QX 59    XX    LO    LO    83    LO    LO    LO
* 4qx  59 xx ,  lo lo 83 lo lo lo
or4  59 , lo lo 83 lo
or2  xx , lo lo

* 6PT 76    XX    XX    61    LO    LO    LO    LO
* 6pt  76 xx xx , 61 lo lo lo lo
inv  xx , 61
inv  76 , xx
or4  xx , lo lo lo lo

* 7PT 84    XX    XX    60    LO    LO    LO    LO
* 7pt  84 xx xx , 60 lo lo lo lo
inv  xx , 60
inv  84 , xx
or4  xx , lo lo lo lo

* 8PT 85    XX    XX    63    LO    LO    LO    LO
* 8pt  85 xx xx , 63 lo lo lo lo
```

```
inv  xx , 63
inv  85 , xx
or4  xx , l0 l0 l0 l0

* 9PT 86    XX    XX    62    L0    L0    L0    L0
* 9pt  86 xx xx , 62 l0 l0 l0 l0
inv  xx , 62
inv  86 , xx
or4  xx , l0 l0 l0 l0

* 5QU 89    L0    88    L0    87
* 5qu  89 , l0 88 l0 87
nor2  5qu1 , l0 88
inv  5qu2 , 5qu1
nor2  5qu3 , l0 5qu2
nor2  5qu4 , 5qu1 87
nor2  89 , 5qu3 5qu4

* 10PT      90    XX    XX    32    L0    L0    L0    L0
* 10pt  90 xx xx , 32 l0 l0 l0 l0
inv  xx , 32
inv  90 , xx
or4  xx , l0 l0 l0 l0

* 11PT      91    XX    XX    30    L0    L0    L0    L0
* 11pt  91 xx xx , 30 l0 l0 l0 l0
inv  xx , 30
inv  91 , xx
or4  xx , l0 l0 l0 l0

* 12PT      92    XX    XX    31    L0    L0    L0    L0
* 12pt  92 xx xx , 31 l0 l0 l0 l0
inv  xx , 31
inv  92 , xx
or4  xx , l0 l0 l0 l0

* 13PT      93    XX    XX    28    L0    L0    L0    L0
* 13pt  93 xx xx , 28 l0 l0 l0 l0
inv  xx , 28
inv  93 , xx
or4  xx , l0 l0 l0 l0

* 14PT      94    XX    XX    29    L0    L0    L0    L0
* 14pt  94 xx xx , 29 l0 l0 l0 l0
inv  xx , 29
inv  94 , xx
or4  xx , l0 l0 l0 l0

* 5QX 27    XX    L0    L0    79    L0    L0    L0
* 5qx  27 xx ,  l0 l0 79 l0 l0 l0
or4  27 , l0 l0 79 l0
or2  xx , l0 l0

* 6QX 25    XX    L0    L0    95    L0    L0    L0
* 6qx  25 xx ,  l0 l0 95 l0 l0 l0
or4  25 , l0 l0 95 l0
or2  xx , l0 l0
```

```
* 6QU        97     71     LO     96     73
* 6qu  97 , 71 lo 96 73
nor2  6qu1 , 71 lo
inv   6qu2 , 6qu1
nor2  6qu3 , 96 6qu2
nor2  6qu4 , 6qu1 73
nor2  97 , 6qu3 6qu4

* 7QU        99     71     LO     98     95
* 7qu  99 , 71 lo 98 95
nor2  7qu1 , 71 lo
inv   7qu2 , 7qu1
nor2  7qu3 , 98 7qu2
nor2  7qu4 , 7qu1 95
nor2  99 , 7qu3 7qu4

* 8QU        102    LO     101    LO     100
* 8qu  102 , lo 101 lo 100
nor2  8qu1 , lo 101
inv   8qu2 , 8qu1
nor2  8qu3 , lo 8qu2
nor2  8qu4 , 8qu1 100
nor2  102 , 8qu3 8qu4

* 7QX        23     XX     LO     LO     103    LO     LO     LO
* 7qx  23 xx ,  lo lo 103 lo lo lo
or4   23 , lo lo 103 lo
or2   xx , lo lo

* 15PT       XX     104    XX     20     LO     LO     LO     LO
* 15pt  xx 104 xx , 20 lo lo lo lo
inv   104 , 20
inv   xx , 104
or4   xx , lo lo lo lo

* 16PT       105    XX     XX     19     LO     LO     LO     LO
* 16pt  105 xx xx , 19 lo lo lo lo
inv   xx , 19
inv   105 , xx
or4   xx , lo lo lo lo

* 17PT       107    106    XX     17     LO     LO     LO     LO
* 17pt  107 106 xx , 17 lo lo lo lo
inv   106 , 17
inv   107 , 106
or4   xx , lo lo lo lo

* 18PT       109    108    XX     18     LO     LO     LO     LO
* 18pt  109 108 xx , 18 lo lo lo lo
inv   108 , 18
inv   109 , 108
or4   xx , lo lo lo lo

* 19PT       111    110    XX     16     LO     LO     LO     LO
* 19pt  111 110 xx , 16 lo lo lo lo
inv   110 , 16
inv   111 , 110
or4   xx , lo lo lo lo
```

```
* 9QU 114   L0    113   L0    112
* 9qu  114 , l0 113 l0 112
nor2  9qu1 , l0 113
inv   9qu2 , 9qu1
nor2  9qu3 , l0 9qu2
nor2  9qu4 , 9qu1 112
nor2  114 , 9qu3 9qu4

* 1PU 115   XX    XX    46    L0    L0
* 1pu 115 xx xx , 46 l0 l0
inv   xx , 46
inv   115 , xx
or2   xx , l0 l0

* 2PU 116   XX    XX    47    L0    L0
* 2pu 116 xx xx , 47 l0 l0
inv   xx , 47
inv   116 , xx
or2   xx , l0 l0

* 3PU 117   XX    XX    44    L0    L0
* 3pu 117 xx xx , 44 l0 l0
inv   xx , 44
inv   117 , xx
or2   xx , l0 l0

* 4PU 118   XX    XX    45    L0    L0
* 4pu 118 xx xx , 45 l0 l0
inv   xx , 45
inv   118 , xx
or2   xx , l0 l0

* 5PU 119   XX    XX    42    L0    L0
* 5pu 119 xx xx , 42 l0 l0
inv   xx , 42
inv   119 , xx
or2   xx , l0 l0

* 6PU 120   XX    XX    43    L0    L0
* 6pu 120 xx xx , 43 l0 l0
inv   xx , 43
inv   120 , xx
or2   xx , l0 l0

* 7PU 121   XX    XX    41    L0    L0
* 7pu 121 xx xx , 41 l0 l0
inv   xx , 41
inv   121 , xx
or2   xx , l0 l0

* 8PU 122   XX    XX    40    L0    L0
* 8pu 122 xx xx , 40 l0 l0
inv   xx , 40
inv   122 , xx
or2   xx , l0 l0

* 9PU 123   XX    XX    39    L0    L0
* 9pu 123 xx xx , 39 l0 l0
```

```
inv  xx , 39
inv  123 , xx
or2  xx , l0 l0

* 10PU      124   XX    XX    38    L0    L0
* 10pu 124 xx xx , 38 l0 l0
inv  xx , 38
inv  124 , xx
or2  xx , l0 l0

* 11PU      125   XX    XX    37    L0    L0
* 11pu 125 xx xx , 37 l0 l0
inv  xx , 37
inv  125 , xx
or2  xx , l0 l0

* 12PU      126   XX    XX    36    L0    L0
* 12pu 126 xx xx , 36 l0 l0
inv  xx , 36
inv  126 , xx
or2  xx , l0 l0

* 13PU      127   XX    XX    35    L0    L0
* 13pu 127 xx xx , 35 l0 l0
inv  xx , 35
inv  127 , xx
or2  xx , l0 l0

* 14PU      128   XX    XX    33    L0    L0
* 14pu 128 xx xx , 33 l0 l0
inv  xx , 33
inv  128 , xx
or2  xx , l0 l0

* 15PU      129   XX    XX    34    L0    L0
* 15pu 129 xx xx , 34 l0 l0
inv  xx , 34
inv  129 , xx
or2  xx , l0 l0

* 16PU      130   64    65    2     131   132
* 16pu 130 64 65 , 2 131 132
inv  64 , 2
inv  130 , 64
or2  65 , 131 132

* 17PU      133   134   135   1     131   130
* 17pu 133 134 135 , 1 131 130
inv  134 , 1
inv  133 , 134
or2  135 , 131 130

* 18PU      136   87    137   3     131   130
* 18pu 136 87 137 , 3 131 130
inv  87 , 3
inv  136 , 87
or2  137 , 131 130
```

```
* 19PU      138   132   139   4     130   138
* 19pu 138 132 139 , 4 130 138
inv  132 , 4
inv  138 , 132
or2  139 , 130 138

* 20PU      140   XX    88    5     134   110
* 20pu 140 xx 88 , 5 134 110
inv  xx , 5
inv  140 , xx
or2  88 , 134 110

* 21PU      141   131   142   6     134   136
* 21pu 141 131 142 , 6 134 136
inv  131 , 6
inv  141 , 131
or2  142 , 134 136

* 22PU      143   XX    144   7     134   136
* 22pu 143 xx 144 , 7 134 136
inv  xx , 7
inv  143 , xx
or2  144 , 134 136

* 23PU      145   XX    146   8     136   111
* 23pu 145 xx 146 , 8 136 111
inv  xx , 8
inv  145 , xx
or2  146 , 136 111

* 24PU      147   148   XX    9     LO    LO
* 24pu 147 148 xx , 9  lO lO
inv  148 , 9
inv  147 , 148
or2  xx ,  lO lO

* 25PU      149   XX    151   11    148   150
* 25pu 149 xx 151 , 11 148 150
inv  xx , 11
inv  149 , xx
or2  151 , 148 150

* 26PU      150   112   152   10    148   150
* 26pu 150 112 152 , 10 148 150
inv  112 , 10
inv  150 , 112
or2  152 , 148 150

* 27PU      153   154   113   13    148   155
* 27pu 153 154 113 , 13 148 155
inv  154 , 13
inv  153 , 154
or2  113 , 148 155

* 28PU      156   155   158   12    108   157
* 28pu 156 155 158 , 12 108  157
inv  155 , 12
inv  156 , 155
```

```
or2  158 , 108  157

* 29PU      159   XX    160   15    108   157
* 29pu 159 xx 160 , 15 108  157
inv  xx , 15
inv  159 , xx
or2  160 , 108  157

* 30PU      157   100   101   14    108   154
* 30pu 157 100 101 , 14 108 154
inv  100 , 14
inv  157 , 100
or2  101 , 108 154

* 1QD 162 161 164 163 107 67 118 69 70 107 117 68 116 115
* 1qd  162 161 164 163 , 107 67 118 69 70 107 117 68 116 115
inv  1qd1 , 107
inv  1qd2 , 107
nor2  1qd3 , 107 67
nor2  1qd4 , 1qd1 118
nor2  1qd5 , 107 69
nor2  1qd6 , 1qd1 70
nor2  1qd7 , 107 117
nor2  1qd8 , 1qd2 68
nor2  1qd9 , 107 116
nor2  1qd10 , 1qd2 115
nor2  162 , 1qd3 1qd4
nor2  161 , 1qd5 1qd6
nor2  164 , 1qd7 1qd8
nor2  163 , 1qd9 1qd10

* 1QH 98 80 171 170 L0 L1 161 172 173 107 164 174
* 1qh  98 80 ,  171 170 l0 l1 161 172 173 107 164 174
inv  1qh1 ,  171
inv  1qh2 ,  170
nor3  1qh3 ,  171 170 l0
nor3  1qh4 ,  1qh1 170 l1
nor3  1qh5 ,  171 1qh2 161
nor3  1qh6 ,  1qh1 1qh2 172
nor3  1qh7 ,  171 170 173
nor3  1qh8 ,  1qh1 170 107
nor3  1qh9 ,  171 1qh2 164
nor3  1qh10 ,  1qh1 1qh2 174
nor4  98 , 1qh3 1qh4 1qh5 1qh6
nor4  80 , 1qh7 1qh8 1qh9 1qh10

* 2QD 166 165 168 167 107 124 125 123 121 107 119 120 122 126
* 2qd  166 165 168 167 , 107 124 125 123 121 107 119 120 122 126
inv  2qd1 , 107
inv  2qd2 , 107
nor2  2qd3 , 107 124
nor2  2qd4 , 2qd1 125
nor2  2qd5 , 107 123
nor2  2qd6 , 2qd1 121
nor2  2qd7 , 107 119
nor2  2qd8 , 2qd2 120
nor2  2qd9 , 107 122
nor2  2qd10 , 2qd2 126
```

```
nor2  166 , 2qd3 2qd4
nor2  165 , 2qd5 2qd6
nor2  168 , 2qd7 2qd8
nor2  167 , 2qd9 2qd10

* 2QH 77 96 171 170 L1 L1 L0 175 L0 L0 162 176
* 2qh  77 96 ,  171 170 l1 l1 l0 175 l0 l0 162 176
inv  2qh1 ,  171
inv  2qh2 ,  170
nor3  2qh3 ,  171 170 l1
nor3  2qh4 ,  2qh1 170 l1
nor3  2qh5 ,  171 2qh2 l0
nor3  2qh6 ,  2qh1 2qh2 175
nor3  2qh7 ,  171 170 l0
nor3  2qh8 ,  2qh1 170 l0
nor3  2qh9 ,  171 2qh2 162
nor3  2qh10 ,  2qh1 2qh2 176
nor4  77 , 2qh3 2qh4 2qh5 2qh6
nor4  96 , 2qh7 2qh8 2qh9 2qh10

* 1PF       177 XX 178 XX XX 84 137 138 135 85 144 111 142 L0 L0
*  1pf  177 xx 178 xx xx , 94 137 138 135 85 144 111 142 l0 l0
or2  1pf1 ,  84 137
or2  1pf2 , 138 135
nor2  1pf3 , 1pf1 1pf2
nor2  1pf4 , 1pf1 1pf3
nor2  1pf5 , 1pf3 1pf2
nor2  xx ,  1pf4 1pf5
inv  177 , xx
or2  1pf6 ,  85 144
or2  1pf7 , 111 142
nor2  1pf8 , 1pf6 1pf7
nor2  1pf9 , 1pf6 1pf8
nor2  1pf10 , 1pf8 1pf7
nor2 xx ,  1pf9 1pf10
inv  178 , xx
or2  xx , l0 l0

* 3QD       180 179 169 183 106 143 149 145 140 71 181 83 182 82
* 3qd  180 179 169 183 , 106 143 149 145 140 71 181 83 182 82
inv  3qd1 , 106
inv  3qd2 , 71
nor2  3qd3 , 106 143
nor2  3qd4 , 3qd1 149
nor2  3qd5 , 106 145
nor2  3qd6 , 3qd1 140
nor2  3qd7 , 71 181
nor2  3qd8 , 3qd2 83
nor2  3qd9 , 71 182
nor2  3qd10 , 3qd2 82
nor2  180 , 3qd3 3qd4
nor2  179 , 3qd5 3qd6
nor2  169 , 3qd7 3qd8
nor2  183 , 3qd9 3qd10

* 4QD       175 176 172 174 106 128 127 129 90 106 91 94 92 93
* 4qd  175 176 172 174 , 106 128 127 129 90 106 91 94 92 93
inv  4qd1 , 106
```

```
inv  4qd2 , 106
nor2  4qd3 , 106 128
nor2  4qd4 , 4qd1 127
nor2  4qd5 , 106 129
nor2  4qd6 , 4qd1 90
nor2  4qd7 , 106 91
nor2  4qd8 , 4qd2 94
nor2  4qd9 , 106 92
nor2  4qd10 , 4qd2 93
nor2  175 , 4qd3 4qd4
nor2  176 , 4qd5 4qd6
nor2  172 , 4qd7 4qd8
nor2  174 , 4qd9 4qd10

* 3QH       170 173 105 107 102 66 114 89 109 141 147 133
* 3qh  170 173 ,  105 107 102 66 114 89 109 141 147 133
inv  3qh1 ,  105
inv  3qh2 ,  107
nor3  3qh3 ,  105 107 102
nor3  3qh4 ,  3qh1 107 66
nor3  3qh5 ,  105 3qh2 114
nor3  3qh6 ,  3qh1 3qh2 89
nor3  3qh7 ,  105 107 109
nor3  3qh8 ,  3qh1 107 141
nor3  3qh9 ,  105 3qh2 147
nor3  3qh10 ,  3qh1 3qh2 133
nor4  170 , 3qh3 3qh4 3qh5 3qh6
nor4  173 , 3qh7 3qh8 3qh9 3qh10

* 4QH       74 XX 171 184 L0 L0 L1 163 L0 L0 L0 L0
* 4qh  74 xx ,  171 184 l0 l0 l1 163 l0 l0 l0 l0
inv  4qh1 ,  171
inv  4qh2 ,  184
nor3  4qh3 ,  171 184 l0
nor3  4qh4 ,  4qh1 184 l0
nor3  4qh5 ,  171 4qh2 l1
nor3  4qh6 ,  4qh1 4qh2 163
nor3  4qh7 ,  171 184 l0
nor3  4qh8 ,  4qh1 184 l0
nor3  4qh9 ,  171 4qh2 l0
nor3  4qh10 ,  4qh1 4qh2 l0
nor4  74 , 4qh3 4qh4 4qh5 4qh6
nor4  xx , 4qh7 4qh8 4qh9 4qh10

* 5QH 181 182 171 184 185 168 166 180 186 167 165 179
* 5qh  181 182 ,  171 184 185 168 166 180 186 167 165 179
inv  5qh1 ,  171
inv  5qh2 ,  184
nor3  5qh3 ,  171 184 185
nor3  5qh4 ,  5qh1 184 168
nor3  5qh5 ,  171 5qh2 166
nor3  5qh6 ,  5qh1 5qh2 180
nor3  5qh7 ,  171 184 186
nor3  5qh8 ,  5qh1 184 167
nor3  5qh9 ,  171 5qh2 165
nor3  5qh10 ,  5qh1 5qh2 179
nor4  181 , 5qh3 5qh4 5qh5 5qh6
nor4  182 , 5qh7 5qh8 5qh9 5qh10
```

```
* 6QH       171 184 105 107 187 177 188 178 190 189 191 192
* 6qh  171 184 ,  106 107 187 177 188 178 190 189 191 192
inv  6qh1 ,  105
inv  6qh2 ,  107
nor3  6qh3 ,  105 107 187
nor3  6qh4 ,  6qh1 107 177
nor3  6qh5 ,  105 6qh2 188
nor3  6qh6 ,  6qh1 6qh2 178
nor3  6qh7 ,  105 107 190
nor3  6qh8 ,  6qh1 107 189
nor3  6qh9 ,  105 6qh2 191
nor3  6qh10 ,  6qh1 6qh2 192
nor4  171 , 6qh3 6qh4 6qh5 6qh6
nor4  184 , 6qh7 6qh8 6qh9 6qh10

* 7QH 185 186 105 106 159 85 86 84 156 111 153 138
* 7qh  185 186 ,  105 106 159 85 86 84 156 111 153 138
inv  7qh1 ,  105
inv  7qh2 ,  106
nor3  7qh3 ,  105 106 159
nor3  7qh4 ,  7qh1 106 85
nor3  7qh5 ,  105 7qh2 86
nor3  7qh6 ,  7qh1 7qh2 84
nor3  7qh7 ,  105 106 156
nor3  7qh8 ,  7qh1 106 111
nor3  7qh9 ,  105 7qh2 153
nor3  7qh10 ,  7qh1 7qh2 138
nor4  185 , 7qh3 7qh4 7qh5 7qh6
nor4  186 , 7qh7 7qh8 7qh9 7qh10

* 2PF 187 XX 188 XX XX 86 158 153 160 159 151 156 152 LO LO
*  2pf  187 xx 188 xx xx , 86 158 153 160 159 151 156 152 lO lO
or2  2pf1 ,  86 158
or2  2pf2 , 153 160
nor2  2pf3 , 2pf1 2pf2
nor2  2pf4 , 2pf1 2pf3
nor2  2pf5 , 2pf3 2pf2
nor2  xx ,  2pf4 2pf5
inv  187 , xx
or2  2pf6 ,  159 151
or2  2pf7 , 156 152
nor2  2pf8 , 2pf6 2pf7
nor2  2pf9 , 2pf6 2pf8
nor2  2pf10 , 2pf8 2pf7
nor2 xx ,  2pf9 2pf10
inv  188 , xx
or2  xx , lO lO

* PK  190 191 XX 189 XX 192 108 157 153 148 150 156 131 139 134 146
* pk 190 191 xx 189 xx 192 , 108 157 153 148 150 156 131 139 134 146
nor3  190 , 108 157 153
nor3  191 , 148 150 156
nor2  189 , 131 139
inv  xx , 189
nor2  192 , 134 146
inv  xx , 192
```

APPENDIX 5

BLOCK DATA Subroutine SETUP

Appendix 5 presents the BLOCK DATA subroutine SETUP which initializes the data and file strucure of program VICTOR.

```
c*****************************************************************
            block data setup
c*****************************************************************

c  setup the data structure, file structure, and the machine
c    dependent parameters in program victor

c---------------------------------------------------------------
c  define the variable array sizes. all parameter statements in
c    the entire program must be changed if these sizes are changed.
c---------------------------------------------------------------
c  lbnum:  number of predefined library cells
c  ndmax:  maximum number of nodes in the circuit
c  maxpi:  maximum number of primary input nodes in the circuit
c  kio:    maximum number of input/output nodes per cell

      parameter (lbnum=100, ndmax=10000, maxpi=120, kio=200)

c---------------------------------------------------------------
c  define the data structure (basic common block arrays)
c---------------------------------------------------------------

      common /lulist/ lu(10)
c  lu(10):  device number assignment

      common /ukfile/ tmpfil(20)
c  tmpfil(20):      temporary work files

      common /iofile/ inpf, outf
c  inpf, outf:    user-specified input and output file names

      common /nodnam/ libr(lbnum), lognod(ndmax)
c  libr(lbnum):          library cell names and output/input node count
c  lognod(ndmax):        names of circuit nodes and composite fanout
c                        branch names

      common /nodind/ list(ndmax),ndfout(ndmax),ndlev(ndmax)
c  list(ndmax):    scratch array for general storge:
c                        node fanin in module INPROC
c                        fanout node root index in module OBSERV
c                        sorted array pointers in module REPROC
c  ndfout(ndmax): node fanout
c  ndlev(ndmax):  node level

      common /ckttop/ inpckt(2*ndmax), levord(ndmax/2)
c  inpckt(2*ndmax):     machine-readable circuit description as a
c                        set of standard cell entries of the form:
c                        0
c                        cell index (libr array index of cell)
c                        negative cell output node index
c                              .............
c                        negative cell output node index
c                        positive cell input node index
c                              .............
c                        positive cell input node index
c                  where cell index = libr array index of cell name
c                  and node index = lognod array index of node name
```

```
c  levord(ndmax/2):     levelized cell list:  list of inpckt indeces
c                         corresponding to cells in ascending order
c                         of their level

      common /actsiz/ npiel,npoel,ndel,ndlog,ncell,ninput,nlevel
c  npiel,npoel:          number of primary inputs and primary outputs
c  ndel,ndlog:           number of circuit nodes and number of
c                          fault locations in the circuit
c  ncell:                number of circuit cells
c  ninput:               element count in array inpckt
c  nlevel:               number of circuit levels

      common /nodco1/ con0(ndmax),con1(ndmax),obs(ndmax),lab(kio)
c  con0(ndmax):          node 0-controllability label (reset pattern)
c  con1(ndmax):          node 1-controllability label (set pattern)
c  obs(ndmax):           node observability label (monitor pattern)
c  lab(kio):             scratch pad array for label calculations

   common /nodco2/ lwt0(ndmax),lsiz0(ndmax),lwt1(ndmax),lsiz1(ndmax),
     +  lwto(ndmax),lsizo(ndmax) kouts(kio),kins(kio),nout,nin,
     +  lwt(kio),lsiz(kio),lev(kio)
c  lwt0(ndmax),lsiz0(ndmax):  reset (0) weight and size
c  lwt1(ndmax),lsiz1(ndmax):  set (1) weight and size
c  lwto(ndmax),lsizo(ndmax):  monitor weight and size
c  kouts(kio),kins(kio):      cell output and input node indeces
c  nout,nin:                  cell output and input node counts
c  lwt(kio),lsiz(kio),lev(kio): scratch weight, size, and level arrays
c                               for cell testability calculations

      common /tesvec/ veclis(ndmax)
c  veclis(ndmax): sorted test vectors in decreasing order of
c                   the detected faults per vector

      common /actest/ nirred,nred,nfcol
c  nirred,nred,nfcol:   number of guaranteed irredundant and
c                         potentially redundant faults
c  nfcol:          irredundant fault count after fault collapsing

c !!! do not change lengths of the following character variables !!!
c   unless you are willing to update all occurencies in the program.

      character*maxpi, con0, con1, obs, lab, veclis
      character*40 tmpfil, inpf, outf, libr*30, lognod*72

c-------------------------------------------------------------------
c  file structure: file names and logical unit assignment
c-------------------------------------------------------------------

      data tmpfil /
     +      'vic.io',  'vic.syn',  'vic.flo',  'vic.fb',  'vic.net',
     +      'vic.red',  'vic.vec',  'vic8',  'vic9',  'vic10',
     +      'vic11', 'vic12', 'vic13', 'vic14', 'vic15',
     +      'vic16', 'vic17', 'vic18', 'vic19', 'vic20'/

c  tmpfil(1) (vic.io):   user-specified input and output file names
c  tmpfil(2) (vic.syn):  circuit description in standard form
c  tmpfil(3) (vic.flo):  floating nodes and wired logic
c  tmpfil(4) (vic.fb):   boundary of feedback loop area
```

```
c  tmpfil(5) (vic.net):  machine-readable circuit net list
c  tmpfil(6) (vic.red):  potentially redundant faults and aborted
c                        test vectors
c  tmpfil(7) (ic.vec):   test vectors for irredundant faults
c  tmpfil(8)-tmpfil(20): spare files for future extensions
c       vic8-vic20

c  device number assignments for vax/unix (machine dependent):
c     1-4  = input files;  5 = standard input (from terminal)
c     7-10 = output files; 6 = standard output (to terminal)

      data lu /1,2,3,4,5,6,7,8,9,10/

c-------------------------------------------------------------
c  library cell names and cell output and input node count
c-------------------------------------------------------------
      data
     + libr(1) /'and2  out=1 in=2'/,    libr(2) /'and3  out=1 in=3'/
     + libr(3) /'and4  out=1 in=4'/,    libr(4) /'and5  out=1 in=5'/
     + libr(5) /'and6  out=1 in=6'/,    libr(6) /'and7  out=1 in=7'/
     + libr(7) /'and8  out=1 in=8'/,    libr(8) /'and9  out=1 in=9'/

     + libr(9) /'aoi21 out=1 in=3'/,    libr(10)/'aoi22  out=1 in=4'/
     + libr(11)/'aoi31 out=1 in=4'/,    libr(12)/'buf   out=1 in=1'/
     + libr(13)/'inv   out=1 in=1'/,    libr(14)/'mux2 out=1 in=3'/

     + libr(15)/'nand2  out=1 in=2'/,   libr(16)/'nand3  out=1 in=3'/
     + libr(17)/'nand4  out=1 in=4'/,   libr(18)/'nand5  out=1 in=5'/
     + libr(19)/'nand6  out=1 in=6'/,   libr(20)/'nand7  out=1 in=7'/
     + libr(21)/'nand8  out=1 in=8'/,   libr(22)/'nand9  out=1 in=9'/

      data
     + libr(23)/'nor2  out=1 in=2'/,    libr(24)/'nor3  out=1 in=3'/
     + libr(25)/'nor4  out=1 in=4'/,    libr(26)/'nor5  out=1 in=5'/
     + libr(27)/'nor6  out=1 in=6'/,    libr(28)/'nor7  out=1 in=7'/
     + libr(29)/'nor8  out=1 in=8'/,    libr(30)/'nor9  out=1 in=9'/

     + libr(31)/'oai21  out=1 in=3'/,   libr(32)/'oai22  out=1 in=4'/
     + libr(33)/'oai31  out=1 in=4'/,   libr(34)/'oai33  out=1 in=6'/

     + libr(35)/'or2  out=1 in=2'/,     libr(36)/'or3  out=1 in=3'/
     + libr(37)/'or4  out=1 in=4'/,     libr(38)/'or5  out=1 in=5'/
     + libr(39)/'or6  out=1 in=6'/,     libr(40)/'or7  out=1 in=7'/
     + libr(41)/'or8  out=1 in=8'/,     libr(42)/'or9  out=1 in=9'/

     + libr(43)/'trag  out=1 in=2'/,    libr(44)/'xnor2  out=1 in=2'/
     + libr(45)/'xor2 out=1 in=2'/

      end
```

APPENDIX 6

Program VICTOR Source Listing

Appendix 6 contains the complete FORTRAN source code of program VICTOR.

To obtain copies of Appendix 6 address inquiries to:

Pamela Bostelman
Industrial Liaison Program
499A Cory Hall
University of California
Berkeley, CA 94720
tel: (415) 642-8312

REFERENCES

[Aho 74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design of and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Agrawal 82] V. D. Agrawal and M. R. Mercer, "Testability Measures —What Do They Tell Us?", *Proceedings International Test Conference*, Philadelphia, Nov. 1982, pp. 391-396.

[Akers 76] S. B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, Vol. C-25, June 1976, pp. 620-630.

[Akers 82] S. B. Akers and B. Krishnamurthy, "A Test Counting Approach to Testability Analysis," private communication, April 1982.

[Ando 80] H. Ando, "Testing VLSI with Random Access Scan," *Digest of Papers Compcon 80*, San Francisco, Feb. 1980, pp. 50-52.

[ANSI 78] *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, New York, 1978.

[Armstrong 66] D. B. Armstrong, "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," *IEEE Transactions on Electronic Computers*, Vol. EC-15, Feb. 1966, pp. 66-73.

[Arzoumanian 81] Y. Arzoumanian and J. Waicukauski, "Fault Diagnosis in an LSSD Environment," *Proceedings 1981 IEEE Test Conference*, Philadelphia, Oct. 1981, pp. 66-68.

[Avizienis 82] A. Avizienis, "The Four-Universe Information System Model for the Study of Fault-Tolerance," *Digest of FTCS-12, 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, June 1982, pp. 6-13.

[Bardell 82] P. H. Bardell, "On a Testability Measure for Random Pattern Testing," private communication, April 1982.

[Beh 82] C. C. Beh, K. H. Arya, C. E. Radke, and K. E. Torku, "Do Stuck Fault Models Reflect Manufacturing Defects," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 35-42.

[Bennets 80] R. G. Bennets, G.D. Robinson, and C. Maunder, "Computer-Aided Measure for Logic Testability: the CAMELOT Program," *Proceedings IEEE International Conference on Circuits and Computers*, Port Chester, Oct. 1980, pp. 1162-1165.

[Berg 82] W. C. Berg and R. D. Hess, "COMET: A Testability Analysis and Design Modification Package," *Proceedings International Test Conference*, Philadelphia, Nov. 1982, pp. 364-378.

[Bernhard 81] R. Bernhard, "#5: Lessons from the Military," Special Report: Reliability, *IEEE Spectrum*, Vol. 18, Oct. 1981, pp. 73-78.

[Bernhard 82] R. Bernhard, "Rethinking the 256-kb RAM," *IEEE Spectrum*, Vol. 19, May 1982, pp. 46-51.

[Blume 83] H. M. Blume, Jr., private communication, Jan. 1983.

[Bose 82] A. K. Bose, P. Kozac, C.-Y. Lo, H. N. Nham, E. Pacas-Skewes, K. Wu, "A Fault Simulator for MOS LSI Circuits," *Proceedings 19th Design*

Automation Conference, Las Vegas, June 1982, pp. 400-409.

[Bottorff 77] P. S. Bottorff, R. E. France, N. H. Garges, and E. J. Orosz, "Test Generation for Large Logic Networks," *Proceedings 14th Design Automation Conference*, New Orleans, June 1977, pp. 479-485.

[Bottorff 79] P. S. Bottorff, R. E. France, and H. C. Godoy, "Automatic Test Generation for LSI Chips and Printed Circuit Boards," *Digest of Technical Papers*, 1979 International Solid-State Circuits Conference, New York, Feb. 1979, pp. 252-253.

[Bottorff 80] P. S. Bottorff, "Computer Aids to Testing - An Overview," *Computer Design Aids for VLSI Circuits*, Sijthoff & Noordhoff International Publishers, The Hague, 1981, pp. 417-464.

[Bottorff 81] P. S. Bottorff, "Partitioning Large LSSD Networks for Test Generation," presentation at the 4th Annual Workshop on Design for Testability, Vail, April 1981.

[Breuer 76] M. A. Breuer and A. D. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press, 1976.

[Breuer 78] M. A. Breuer, "New Concepts in Automated Testing of Digital Circuits," *Proceedings Symposium on Computers-Aided Design of Digital Electronic Circuits and Systems*, Brussels, Belgium, Nov. 1978, pp. 57-80.

[Breuer 79] M. A. Breuer and A. D. Friedman, "TEST/80 - A Proposal for and Advanced Automatic Test Generation System," *Proceedings IEEE AUTOTESTCON*, Oct. 1979, pp. 305-312.

[Carter 82] W. C. Carter, "Signature Testing with Guaranteed Bounds for Fault Coverage," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 75-82.

[Clegg 73] F. Clegg, "Use of SPOOF's in the Analysis of Faulty Logic Networks," *IEEE Transactions on Computers*, Vol. C-22, March 1973, pp. 229-234.

[Cliff 80] R. A. Cliff, "Acceptable Testing of VLSI Components Which Contain Error Correctors," *IEEE Journal of Solid-State Circuits*, Vol. SC-15, Feb. 1980, pp. 61-70.

[Collins 82] C. A. Collins, "IBM 3081 System Overview and Technology," *Proceedings 19th Design Automation Conference*, Las Vegas, June 1982, pp. 75-82.

[Dandapani 74] R. Dandapani and S. Reddy, "On the Design of Logic Networks with Redundancy and Testability Considerations," *IEEE Transactions on Computers*, Vol. C-24, Nov. 1974, pp. 1139-1149.

[Danner 79] F. Danner and W. Consolla, "An Objective PCB Rating System," *Proceedings 1979 IEEE Test Conference*, Cherry Hill, Oct. 1979, pp. 23-28.

[DasGupta 82] S. DasGupta, P. Goel, R. G. Walther, and T. W. Williams, "A Variation of LSSD and Its Implication on Design and Test Pattern Generation in VLSI," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 63-66.

[Dejka 77] W. J. Dejka, "Measure of Testability in Device and System Design," *Proceedings 20th Midwest Symposium on Circuits and Systems*, Aug.

1977, pp. 39-52.

[Dunning 81] B. Dunning and P. Kovijanic, "Demonstration of a Figure of Merit for Inherent Testability," *Proceedings IEEE AUTOTESTCON*, Orlando, Oct. 1981, pp. 515-520.

[Dussault 78] J. A. Dussault, "A Testability Measure," *Proceedings Semiconductor Test Conference*, Cherry Hill, Oct. 1978, pp. 113-116.

[Eichelberger 77] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings 14th Design Automation Conference*, New Orleans, June 1977, pp. 462-468.

[Fong 82a] J. Y. O. Fong, "A Generalized Testability Analysis Algorithm for Digital Logic Circuits," *Proceedings IEEE International Symposium on Circuits and Systems*, Rome, May 1982, pp. 1160-1163.

[Fong 82b] J. Y. O. Fong, "On Functional Controllability and Observability Analysis," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 170-175.

[Frohwerk 77] R. A. Frohwerk, "Signature Analysis: A New Digital Field Service Method," *Hewlett-Packard Journal*, May 1977, pp. 2-8.

[Fujiwara 82] H. Fujiwara and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits," *IEEE Transactions on Computers*, Vol. C-31, June 1982, pp. 555-560.

[Funatsu 75] S. Funatsu, N. Wakatsuki, and T. Arima, "Test Generation Systems in Japan," *Proceedings 12th Design Automation Symposium*, June 1975, pp. 114-122.

[Fung 82] H. S. Fung and J. Y. O. Fong, "An Information Flow Approach to Functional Testability Measures," *Proceedings IEEE International Conference on Circuits and Computers*, New York, Sept. 1982, pp. 460-463.

[Goel 81] P. Goel and B. C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," *Proceedings 18th Design Automation Conference*, Nashville, June 1982, pp. 260-268.

[Goel 82a] D. K. Goel and R. M. McDermott, "An Interactive Testability Analysis Program — ITTAP," *Proceedings 19th Design Automation Conference*, Las Vegas, June 1982, pp. 581-586.

[Goel 82b] P. Goel and M. T. McMahon, "Electronic Chip-in-Place Test," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 83-90.

[Goldstein 79] L. H. Goldstein, "Controllability/Observability Analysis of Digital Circuits," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, Sept. 1979, pp. 685-691.

[Goldstein 80] L. H. Goldstein and E. L. Thigpen, "SCOAP: SANDIA Controllability/Observability Analysis Program," *Proceedings 17th Design Automation Conference*, Minneapolis, June 1980, pp. 190-196.

[Grason 79] J. Grason, "TMEAS, a Testability Measurement Program," *Proceedings 16th Design Automation Conference*, San Diego, June 1979, pp. 156-161.

[Hayes 71] J. P. Hayes, "A NAND Model for Fault Diagnosis in Combinatorial

Logic Networks," *IEEE Transactions on Computers*, Vol. C-20, Dec. 1971, pp. 1496-1506.

[Hayes 78] J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, 1978.

[Hayes 82] J. P. Hayes, "A Fault Simulation Methodology for VLSI," *Proceedings 19th Design Automation Conference*, Las Vegas, June 1982, pp. 393-399.

[Hess 82] R. D. Hess, "Testability Analysis: An Alternative to Structured Design for Testability," *VLSI Design*, March/April 1982, pp. 22-27.

[Ibarra 75] O. H. Ibarra and S. K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, Vol. C-24, March 1975, pp. 242-249.

[IEEE 77] *IEEE Standard Dictionary of Electrical & Electronics Terms*, The Institute of Electrical and Electronics Engineers, Inc., 1977.

[Jensen 82] L. Jensen, *Investigation of Commercially Available Programs and Algorithms for Automatic Test Pattern Generation*, Christian Rovsing A/S Technical Report, Copenhagen, September 1982.

[Keiner 77] W. Keiner and R. West, "Testability Measures," *Proceedings AUTOTESTCON 1977*, pp. 49-55.

[Kirkland 83] T. Kirkland and V. Flores, "Software Checks Testability and Generates Tests of VLSI Design," *Electronics*, Vol. 56, No. 5, March 10, 1983, pp. 120-124.

[Kitano 80] Y. Kitano, S. Kohda, H. Kikuchi, and S. Sakai, "A 4Mb Full Wafer ROM," *Digest of Technical Papers*, 1980 International Solid-State Circuits Conference, San Francisco, Feb. 1980, pp. 150-151.

[Knuth 73] D. E. Knuth, *The Art of Computer Programming*, Volume 1/Fundamental Algorithms, Addison-Wesley, 1973.

[Koenemann 79] B. Koenemann, J. Mucha, G. Zwiehoff, "Built-In Logic Block Observation Techniques," *Proceedings 1979 IEEE Test Conference*, Cherry Hill, Oct. 1979, pp. 37-41.

[Kovijanic 79] P. G. Kovijanic, "Testability Analysis," *Proceedings 1979 IEEE Test Conference*, Cherry Hill, Oct. 1979, pp. 310-316.

[Kovijanic 81] P. G. Kovijanic, "Single Testability Figure of Merit," *Proceedings IEEE Test Conference*, Philadelphia, Oct. 1981, pp. 521-529.

[Kuck 78] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley & Sons, 1978.

[Lee 74] H.-P. S. Lee and E. S. Davidson, "Redundancy Testing in Combinational Networks," *IEEE Transactions on Computers*, Vol. C-23, Sep. 1974, pp. 1029-1047.

[Levi 81] M. W. Levi, "CMOS is Most Testable," *Proceedings 1981 International Test Conference*, Philadelphia, Oct. 1981, pp. 217-220.

[Lineback 82] J. R. Lineback, "CAD Program Evaluates Circuits, Generates Tests Automatically," *Electronics*, Vol. 55, No. 22, Nov. 3, 1982, pp. 45-46.

[Longendorfer 79] B. Longendorfer, "A Testability Measure for Hybrid Circuits," *Proceedings 1979 IEEE Test Conference*, Cherry Hill, Oct. 1979, pp. 298-305.

[Longendorfer 81] B. Longendorfer, "Computer-Aided Testability Analysis of Analog Circuitry," *Proceedings IEEE AUTOTESTCON*, Nov. 1981, pp. 122-127.

[Lowden 79] R. P. Lowden, "Testing a High Density Logic Masterslice," *Digest of Technical Papers*, 1979 International Solid-State Circuits Conference, New York, Feb. 1979, pp. 250-251.

[Mano 80] T. Mano, K. Takeya, Y. Watanabe, K. Kiuchi, T. Ogawa and K. Hirata, "A 256k RAM Fabricated with Molybdenum-Polysilicon Technology," *Digest of Technical Papers*, 1980 International Solid-State Circuits Conference, San Francisco, Feb. 1980, pp. 234-235.

[McKenny 80] V. G. McKenny, "A 5V 64k EPROM Utilizing Redundant Circuitry," *Digest of Technical Papers*, 1980 International Solid-State Circuits Conference, San Francisco, Feb. 1980, pp. 146-147.

[Menzel 82] S. P. Menzel, *Testability Analysis Considerations of Digital Circuits*, M.S. thesis, Department of Electronics, Carleton University, Ottawa, Canada, Aug. 1982.

[Mercer 81] M. R. Mercer, V. D. Agrawal, and C. M. Roman, "Test Generation for Highly Sequential Scan-Testable Circuits Through Logic Transformation," *Proceedings 1981 IEEE Test Conference*, Philadelphia, Oct. 1981, pp. 561-565.

[Muehldorf 81] E. I. Muehldorf and A. D. Savkar, "LSI Logic Testing —An Overview," *IEEE Transactions on Computers*, Vol. C-30, Jan. 1981, pp. 1-17.

[Newton 81] R. A. Newton, "A Blue Collar Language for CAD," *Digest of Papers Compcon 81*, San Francisco, Feb. 1981, pp. 81-82.

[Parker 75a] K. P. Parker and E. J. McCluskey, "Analysis of Logic Circuits with Faults Using Input Signal Probabilities," *IEEE Transactions on Computers*, Vol. C-24, May 1975, pp. 573-578.

[Parker 75b] K. P. Parker and E. J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Transactions on Computers*, Vol. C-24, June 1975, pp. 668-670.

[Ratiu 81] I. M. Ratiu, "Macromodels for Testability Analysis," presentation at the 4th IEEE Workshop on Design for Testability, Vail, April 1981.

[Ratiu 82] I. M. Ratiu, A. Sangiovanni-Vincentelli, and D. O. Pederson, "VICTOR: A Fast Testability Analysis Program," *Proceedings IEEE International Test Conference*, Philadelphia, Nov. 1982, pp. 397-401.

[Roth 66] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, Vol. 10, July 1966, pp. 278-293.

[Rutman 72] R. A. Rutman, *Fault-Detection Test Generation for Sequential Logic by Heuristic Tree Search*, IEEE Computer Repository Paper No. R-72-187.

[Savir 82] J. Savir, *Good Controlability and Observability Do Not Guarantee Good Testability*, IBM Research Report, RC 9432 (#41597), June 1982.

[Schertz 72] D. R. Schertz and G. Metze, "A New Representation for Faults in Combinational Circuits," *IEEE Transactions on Computers*, Vol. C-21, Aug. 1972, pp. 858-866.

[Schneider 67] P. R. Schneider, "On the Necessity to Examine D-chains in Diagnostic Test Generation —An Example," *IBM Journal of Research and Development*, Vol. 11, Jan. 1967, pp. 114.

[Si 78] S.-C. Si, "Dynamic Testing of Redundant Logic Networks," *IEEE Transactions on Computers*, Vol. C-27, Sep. 1978, pp. 828-832.

[Smith 78] J. E. Smith, "On the Existence of Combinational Circuits Exhibiting Multiple Redundancy," *IEEE Transactions on Computers*, Vol. C-27, Dec. 1978, pp. 1221-1225.

[Smith 79] J. E. Smith, "Comments on 'Redundancy Testing in in Combinational Networks,'" *IEEE Transactions on Computers*, Vol. C-28, March 1979, pp. 261-262.

[Stephenson 74] J. E. Stephenson, *A Testability Measure for Register Transfer Level Digital Circuits*, Ph.D. dissertation, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Nov. 1974.

[Stephenson 76] J. E. Stephenson and J. Grason, "A Testability Measure for Register Transfer Level Digital Circuits," *Proceedings 6th International Symposium on Fault Tolerant Computing*, Pittsburgh, June 1976, pp. 101-107.

[Susskind 81] A. Susskind, *Testability and Reliability of LSI*, RADC Report,

RADC-TR-80-384, Jan. 1981, pp. 99-122.

[Takasaki 81] S. Takasaki, M. Kawai, S. Funatsu, and A. Yamada, "A Calculus of Testability Measure at the Functional Level," *Proceedings 1981 International Test Conference*, Philadelphia, Oct. 1981, pp. 95-101.

[Timoc 82] C. C. Timoc, private communication, April 1982.

[To 73] K. To, "Fault Folding for Irredundant and Redundant Combinational Circuits," *IEEE Transactions on Computers*, Vol. C-22, Nov. 1973, pp. 1008-1015.

[Trischler 81] E. Trischler, private communication, Sept. 1981.

[Van Egmond 82] K. Van Egmond, *LITE: Automatic Transformation of Logic Equations into* Testability Equations," M.S. report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Dec. 1982.

[Wadsack 82] R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell Systems Technical Journal*, May-June 1978, pp. 1449-1474.

[Williams 73] M. J. Y. Williams and J. B. Angell, "Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic," *IEEE Transactions on Computers*, Vol. C-22, Jan. 1973, pp. 46-60.

[Williams 81] W. C. Williams, "#6: Lessons from NASA," Special Report: Reliability, *IEEE Spectrum*, Vol. 18, Oct. 1981, pp. 79-84.

[Williams 82] T. W. Williams and K. P. Parker, "Design for Testability — A Sur-

vey," *IEEE Transactions on Computers*, Vol. C-31, Jan. 1982, pp. 2-15.

[Yau 71] S. S. Yau and Y. S. Tang, "On the Identification of Redundancy and Symmetry of Switching Functions," *IEEE Transactions on Computers*, Vol. C-20, Dec. 1971, pp. 1609-1613.