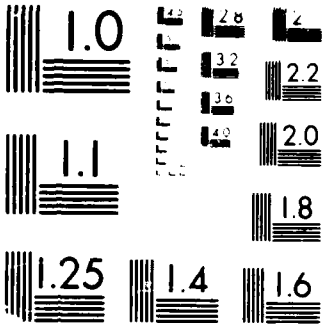
AD-A190 141 DETECTING BRIDGING FAULTS WITH STUCK-AT TEST SETS(U) 1/1
STANFORD UNIV CA CENTER FOR RELIABLE COMPUTING
S D MILLMAN ET AL DEC 87 CRC-TR-87-20 N00014-85-K-0600
UNCLASSIFIED F/G 9/1 NL



MICROCOPY RESOLUTION TEST CHART

AD-A190 141

DTIC FILE COPY

Center for Reliable Computing

Detecting Bridging Faults With Stuck-at Test Sets

Steven D. Millman and Edward J. McCluskey

CRC Technical Report No. 87-20
(CSL TN No. 338)
December 1987

DTIC
ELECTE
FEB 1 0 1988
S D
H

Center for Reliable Computing
ERL 460
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055 USA
(415) 723-1258

Imprimatur: Jon G. Udell, Jr. and Samy Makar.

This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization administered through the Office of Naval Research under Contract No. N00014-85-K-0600.

Copyright © 1987 by the Center for Reliable Computing, Stanford University. All rights reserved, including the right to reproduce this report, or portions thereof, in any form.

Detecting Bridging Faults With Stuck-at Test Sets

Steven D. Millman and Edward J. McCluskey

CRC Technical Report No. 20
(CSL TN No. 338)
December 1987

Center for Reliable Computing
ERL 460
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055 USA
(415) 723-1258

ABSTRACT

Simulations run on sample circuits show that extremely high detection of bridging faults is possible using modifications of psuedo-exhaustive test sets. Real chips often contain bridging faults, and this research shows that stuck-at test sets are not sufficient for detecting such faults. The modified pseudo-exhaustive test sets are easy to generate and require little, or no, fault simulation. Criteria have been found for identifying bridging faults unlikely to be detected by test sets. Techniques for increasing the bridging fault coverage of test sets without consuming excessive computer time are suggested.

Keywords: bridging faults, stuck-at faults, pseudo-exhaustive test, fault modeling.

per Letter

A-1

TABLE OF CONTENTS

Title	Page
Abstract	i
Table of Contents	ii
List of Figures	iii
List of Tables	iii
INTRODUCTION	1
THE BRIDGING FAULT SIMULATION MODEL	3
RESULTS	6
A NEW PSEUDO-EXHAUSTIVE APPROACH FOR BRIDGING FAULTS	13
CONCLUSIONS	16
ACKNOWLEDGEMENTS	16
REFERENCES	17
APPENDIX A: The Test Sets	18
APPENDIX B: Missed and Undetectable Faults	26

LIST OF FIGURES

Figure	Title	Page
1	A CMOS Bridging Fault and Electrical Model	2
2	An AND Nonfeedback Bridging Fault and Logical Model	4
3	An AND Feedback Bridging Fault and Logical Model	4
4	The 74LS181 16-function ALU	7
5	The Parity Tree	9
6	An OR Feedback Bridging Fault for Which the Order of Tests Is Important	10
7	The Four Multiplexer Implementations	12
8	Two Segments With Inputs That Always Have the Same Value	14
9	The Four Multiplexer Implementations	26
10	The Parity Tree	27
11	The 74LS181 16-function ALU	28

LIST OF TABLES

Table	Title	Page
1	Results of 74LS181 Simulations	8
2	Results of Parity Circuit Simulations	8
3	Simulation Results for the Two-Level Multiplexers	10
4	Simulation Results for the Four-Level Multiplexers	11
5	Comparison of Pseudo-Exhaustive Tests for the ALU	15

INTRODUCTION

As the density of devices on VLSI chips has increased, bridging faults have become a research area of great interest [Bhattacharya 85], [Xu 82], [Malaiya 86], [Acken 83]. Recently, work has been done on determining algorithms to generate tests for detecting bridging faults [Karpovsky 80], [Abramovici 83]. However, in a typical VLSI circuit, the number of bridging faults is so large that the task of deriving a test for each one is not economical. This paper describes a method that provides high detection of bridging faults without requiring extensive fault simulation.

A *bridging fault, (x,y)*, occurs when two or more leads are unintentionally shorted together resulting in wired logic. *Feedback bridging faults* occur when the value of y can depend on the value of x in the fault free circuit. To detect a bridging fault, the fault must be activated and the result of the fault must be propagated to an output. In the past, it was thought that the leads involved in the bridging fault had to have different logical values in order to activate the bridging fault, [Karpovsky 80]. This investigation found that this restriction is not necessary for feedback bridging faults since they create "memory" in an otherwise combinational circuit.

In this paper, feedback faults that cause oscillation upon the application of a vector are not considered to be detected by that vector. This limitation is imposed for two reasons based on the *sampling period* of the tester being used. The sampling period is the amount of time after the application of a vector that the tester samples the data on the output pins of the chip. First, if the oscillation period is longer than the sampling period, the fault may not propagate to the output before the next change of inputs. Second, if the oscillation period is shorter than the sampling period, the tester may receive an indeterminate signal that is not guaranteed to indicate a failure.

The wired logic model for bridging faults is still applicable to current technology. A CMOS circuit with a bridging fault and the corresponding electrical model are shown in Fig. 1 [Freeman 86], [Malaiya 86]. Simulations done on CMOS circuits show that bridging faults usually resulted in voltages significantly far from logic thresholds. Intermediate values that could not be correctly interpreted as a zero or a one were rare. This is due to the high noise immunity typically found in CMOS. This provides evidence that bridging faults cause wired logic on the involved

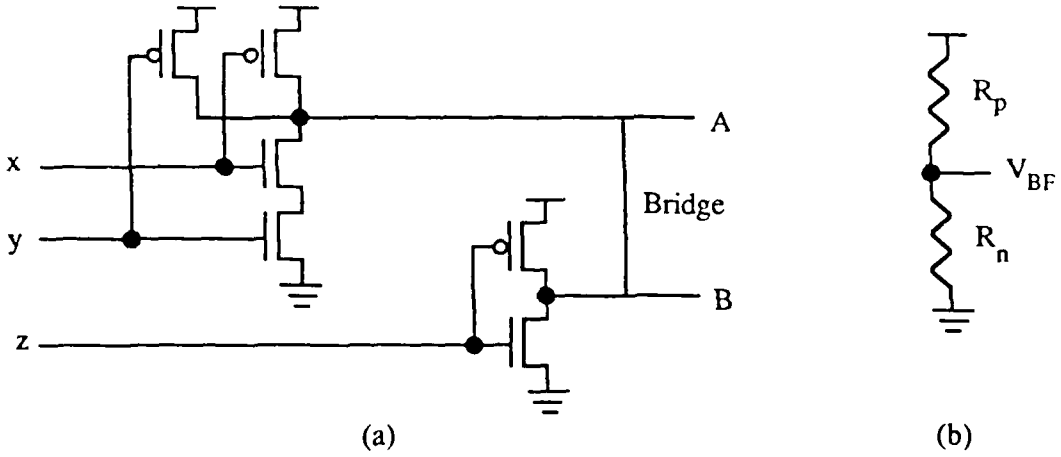


Figure 1. (a) A CMOS bridging fault. (b) The corresponding electrical model.

nodes. It was also found that the wired logic performed depended on the gates driving the bridged leads. As a result, the present work makes no assumptions as to the logic performed: both AND and OR bridging faults were simulated for every pair of leads in each circuit.

It must be noted that not all possible bridging faults need be considered. It has been shown that the detection of one bridging fault can guarantee the detection of another bridging fault, and that the detection of a stuck-at fault can guarantee the detection of one, or more, bridging faults, [Mei 74], [Abramovici 83]. In addition, it was shown that bridging faults on the inputs of an elementary gate are detected by single stuck-at test sets, [Mei 74]. As a result, the methods of fault dominance used to reduce the number of stuck-at faults before pattern generation can be used to reduce the number of bridging faults.

In addition, once a layout of the circuit has been obtained, the list of possible bridging faults can be significantly reduced. This results since only those lines that are adjacent or overlapping are likely to bridge unless all lines in between are also involved. The number of bridging faults left to consider can be reduced from n^2 to 8n, where n is the number of nodes in the circuit [Acken 88]. Since layouts of the circuits studied for this research were not available, bridging faults between all possible pairs of nodes were considered.

Single stuck-at fault test sets, which can be generated by many efficient algorithms, have been proposed for use in detecting bridging faults in two ways. The first method is to take an existing stuck-at test set and determine, through simulation, which bridging faults it detects.

The test vectors are then reordered and/or additional tests are added so that all detectable bridging faults are found [Mei 74]. This method requires extensive computer time since fault simulation must be done for each bridging fault. In addition, each time the test set is reordered, the fault simulation must be repeated to ensure that a feedback bridging fault does not introduce memory that prevents it from being detected. The second method is to alter the stuck-at test pattern generator so that it meets constraints due to the bridging fault problem [Abramovici 83]. These constraints not only slow down the pattern generator, but several vectors may be generated for the same stuck-at fault in order to detect bridging faults associated with that node. In addition, since the modified pattern generator merely follows rules to determine if a bridging fault has been detected by a stuck-at test, it will not be able to efficiently derive tests for bridging faults after all of the stuck-at faults have been detected. Thus, neither of the above methods is efficient for VLSI circuits.

This paper provides a new, economically feasible method for using stuck-at test sets for the detection of bridging faults. Bridging faults that tend to be resistant to the stuck-at test sets can be identified by examining the circuit topology and the stuck-at fault simulation results. Therefore, improved bridging-fault coverage can be attained at a cost only slightly above that of stuck-at test generation alone.

THE BRIDGING FAULT SIMULATION MODEL

The simulations were done at the gate level between all possible pairs of nodes in each circuit, where the nodes consisted of all primary inputs and all gate outputs. Figure 2 illustrates the bridging fault model used for a nonfeedback AND bridging fault. Note that all fanout branches of both leads involved are affected. Since a bridging fault between a stem and a branch is equivalent to the bridging fault between the corresponding stems, only faults between stems were considered.

The model of a feedback AND bridging fault is shown in Fig. 3. (Fanout is treated as above, but is omitted for clarity in the following discussion.) A flip-flop was placed in the feedback loop since circuits with feedback loops unbroken by latches or flip-flops required the use of a slow simulator. To create the impression of a loop without a flip-flop, each test vector was applied three times in succession. If the value in the flip-flop remained stable for the second and third

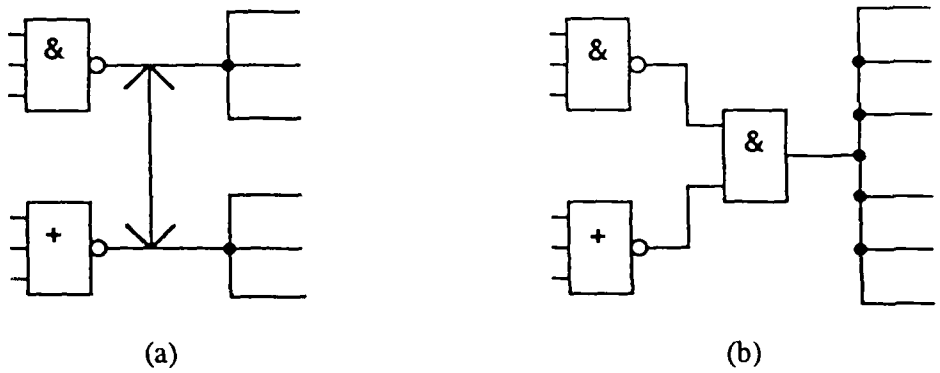


Figure 2. (a) An AND nonfeedback bridging fault. (b) The corresponding logical model.

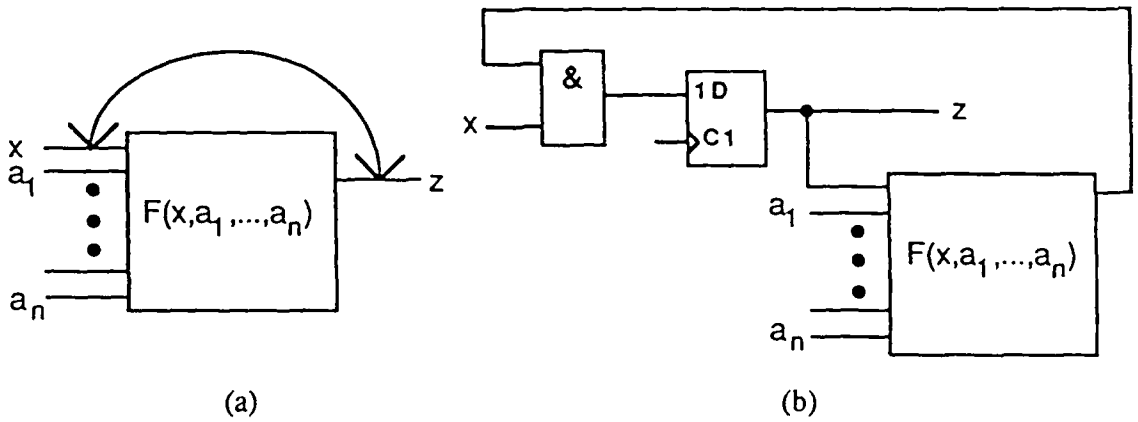


Figure 3. (a) An AND feedback bridging fault. (b) The corresponding logical model.

applications of the vector, then oscillation could not have occurred in the actual circuit. If the value in the flip-flop was not stable, the fault was not considered to be detected.

To show that this model describes the behavior of an actual circuit, consider the AND feedback bridging fault shown in Fig. 3. (The OR feedback bridging fault has a corresponding proof.) It must be noted that regardless of the state of the circuit when the vector is applied, the values on lines $a_1,...,a_n$ will be stable and unaffected by the bridging fault. In addition, if the input x is 0, the output z will become 0 the first time the flip-flop is clocked. It will then remain 0 until the input vector changes. This is true for the actual circuit even if it is oscillating before the application of the vector. Hence, only the cases where x = 1 need be considered. First, the circuit is assumed to be stable (the output is either 0 or 1) before the application of the vector. The cases where the circuit is oscillating before the application of the vector will be considered next.

For convenience, define $A = (a_1,...,a_n)$. There are four cases to examine:

(i) $F(0,A) = 0$, and $F(1,A) = 0$.

The output will go to 0 since both $F(0,A) = 0$ and $F(1,A) = 0$. The first condition is required since the previous vector may have set $z = 0$. The output will remain stable since $F(0,A) = 0$. Only one application of the vector is required for the model to match the actual circuit.

(ii) $F(0,A) = 1$, and $F(1,A) = 1$.

If the previous output was 0, then, since $F(0,A) = 1$, the output will become 1. If the previous output was 1, then the output will remain 1 since $F(1,A) = 1$. In both cases, the output will then remain stable at 1 since $F(1,A) = 1$. Only one application of the vector is required for the model to match the actual circuit.

(iii) $F(0,A) = 0$, and $F(1,A) = 1$.

If the previous output was 0, the output will remain 0 since $F(0,A) = 0$. If the previous output was 1, the output will remain 1 since $F(1,A) = 1$. Again, only one application of the vector is required for the model to match the actual circuit.

(iv) $F(0,A) = 1$, and $F(1,A) = 0$.

If the previous output was 0, then the output will become 1 since $F(0,A) = 1$. However, since $F(1,A) = 0$ the output will now become 0. Hence, the output will oscillate independent of the initial condition of the circuit. Three clockings of the flip-flop are required for the model to oscillate from 0 to 1 and back to 0 (or from 1 to 0 to 1). The first cycle sets up the vector and causes the first change in value. The second and third cause the output to change twice more. Oscillation is detected if the outputs after the second and third cycles do not match.

If the single fault in the circuit is a feedback bridging fault, it is possible that the circuit will oscillate when some, if not all, of the test vectors are applied. When such oscillations occur, the circuit and model will respond to the next vector as follows:

(i) and (ii) Since the output is not a function of the x input, the oscillation will stop and the new output will not be a function of the previous output.

(iii) There is no guarantee that the oscillation will stop in the actual circuit. However, the

oscillation is not self-regenerative since there is no net inversion from x to the output. Hence, given enough time, the output will tend to settle to either 0 or 1. For the purposes of this paper, it is assumed that the amount of time required for the output to stabilize is less than the clock cycle time for the circuit. Without this simplification, the model would have been far too complex to be useful. Since the final value of the output cannot be predicted, when oscillation does occur the value remaining in the flip-flop after the third cycle is used as the initial value for the next vector.

(iv) The oscillations will continue as in case (iv) above.

In cases (i), (ii), and (iii), the bridging fault was detected if the final, stable output of the faulty circuit was different than that of the fault free circuit.

Since dividing all of the possible bridging faults into feedback and nonfeedback groups was a tedious task, even by computer, all faults were simulated using the feedback model. Although this meant that the nonfeedback faults took longer to simulate, time was saved overall since only one model was used and all faults could easily be simulated with one command.

RESULTS

Simulations were run on a 16-function ALU (74LS181) , an 8-bit parity tree of two input XOR gates, and four implementations of a 4-to-1 multiplexer. The test sets used for the simulations, all of which detect 100% of detectable stuck-at faults, are listed in Appendix A. The undetected faults for each test set are listed in Appendix B along with the list of undetectable faults.

Table 1 shows the results for the 16-function ALU shown in Fig. 4. The table shows that the variety of test generation methods for stuck-at faults achieved consistent results in detecting bridging faults. Undetectable bridging faults (such as an OR bridging fault on the input leads of an OR or NOR gate) were not included in the totals. The shorter test sets did not detect as many bridging faults as did the longer test sets. Analysis of the circuit's response to the test sets found that this was due to the shorter test sets failing to propagate the effect of the bridging faults to a primary output. It was also found that bridging faults on the outputs of similar gates, which perform the same logic function, that have common inputs were difficult to detect.

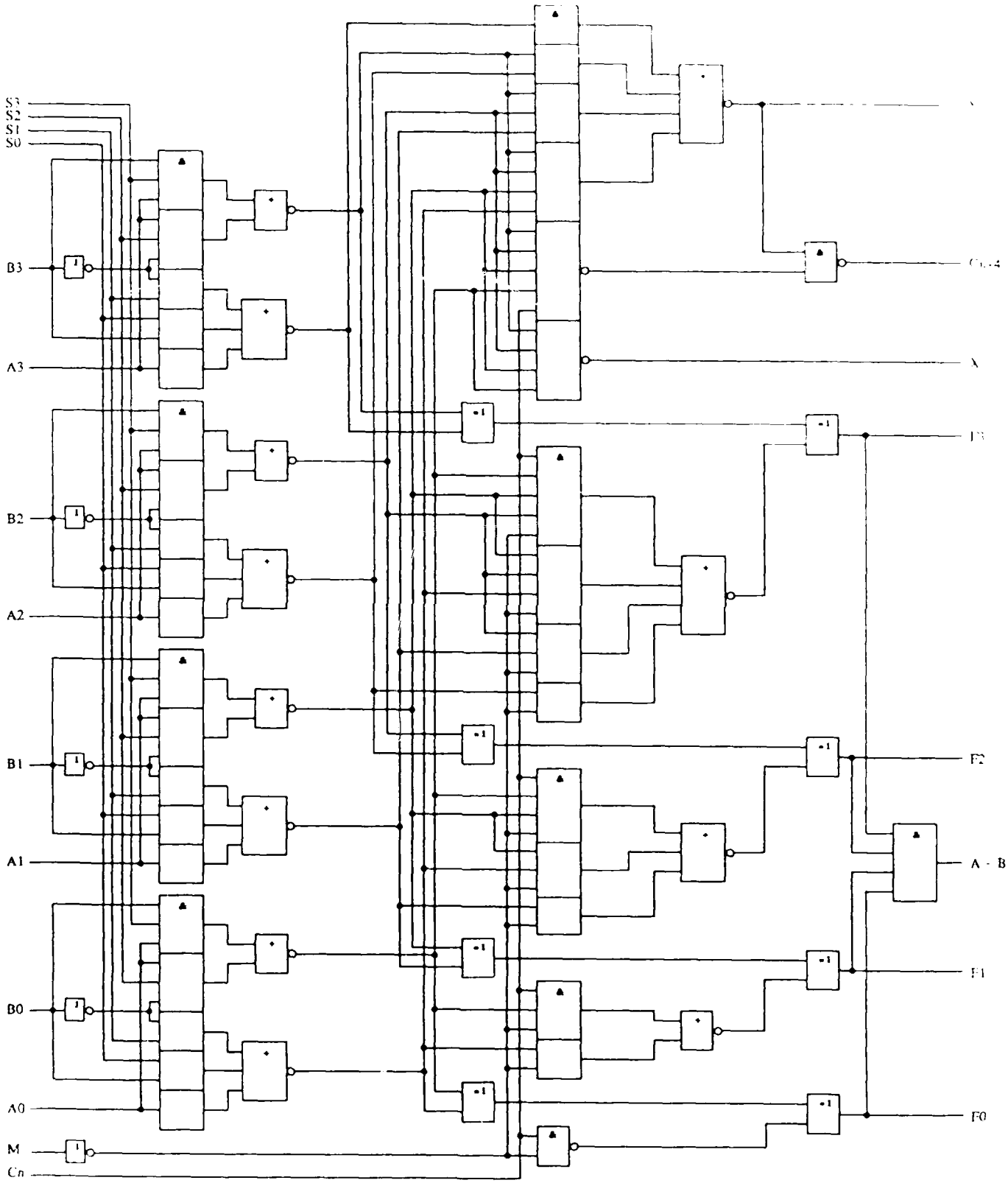


Figure 4. The 74LS181 16-function ALU.

Table 1. Results of 74LS181 simulations.

Test sets	Bryant2	Bryant6	Goel	Hughes	Krish	McC4	Miczo2
Number of test vectors	14	12	35	135	12	124	17
Number of missed AND faults	24	26	4	0	33	0	46
Number of missed OR faults	88	44	5	0	66	1	18
Coverage of AND faults (%)	99.18	99.04	99.86	100.00	98.87	100.00	98.43
Coverage of OR faults (%)	96.79	98.48	99.83	100.00	97.72	99.97	99.38

Total number of each type of fault: 2926

Table 2. Results of parity circuit simulations.

Test sets	Bossen	Bossen2	Millman	Mourad
Number of test vectors	4	4	13	7
Number of missed AND faults	22	22	0	0
Number of missed OR faults	26	22	0	0
Coverage of AND faults (%)	79.05	79.05	100.00	100.00
Coverage of OR faults (%)	75.24	79.05	100.00	100.00

Total number of each type of fault: 105

An example is a bridge between the outputs of a pair of n input AND gates that share n-1 inputs. Even the pseudo-exhaustive test sets, such as McC4, had trouble detecting these faults if the gates were not in the same segment. The only fault that McC4 missed was between nodes that were in "distant" segments. The location of the nodes in distant segments implies that they are not likely to be physically adjacent on an actual chip. Since it is unlikely that a bridging fault between such nodes could occur, the test set detected 100% of all detectable probable bridging faults.

Table 2 shows the results for the parity circuit shown in Fig. 5. The Millman test set is pseudo-exhaustive [McCluskey 86]. The Mourad [Mourad 86] test set was designed to detect all single and double stuck-at faults. The Bossen [Bossen 70] test set consists of four vectors that exhaustively test every XOR gate in the circuit detecting 100% of the single stuck-at faults. This test causes each line to take on one of three possible pattern sets. As a result, for 30 pairs of nodes

in the circuit, the two nodes always have equal values. Since the Bossen test resulted in the output sequence 0011, the Bossen2 sequence was created so that the output sequence became 0101. This ordering was desired since it was expected that feedback bridging fault coverage would be higher if the outputs of the circuit changed more than once during the test set. This expectation, which proved true, is due to the feedback fault causing the circuit to remain in a specific state no matter how the inputs changed. If the outputs of the circuit in the fault free case don't change, then the fault will not be detected.

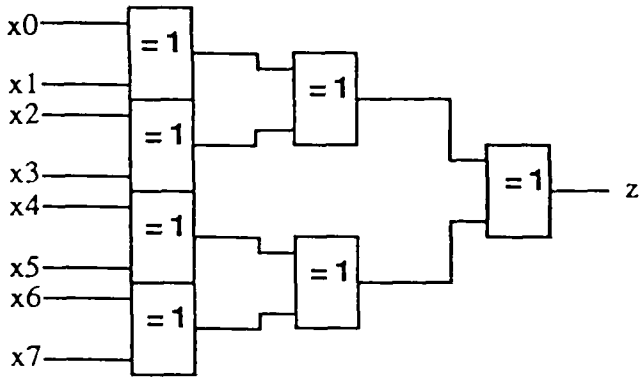
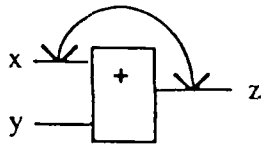


Figure 5. The parity tree.

For example, Fig. 6 shows an OR gate with an OR feedback bridging fault. An exhaustive test of the OR gate can be applied in 24 different orders, two of which are shown. For the first ordering, the fault will not be detected since the output in the presence of the fault is identical to the fault-free output. Here, the output value changes only once. The second ordering causes the fault-free output to change twice, thereby detecting the bridging fault when it latches the output to 1.

While the Bossen test detected all of the bridging faults between leads whose logical values differed at least once, it missed most of the faults between leads whose values never differed. This included all 22 nonfeedback faults of each type between such nodes. The Bossen2 test set detected all of the feedback faults but missed all 22 nonfeedback faults between nodes that never differed. This demonstrates that ultra-short test sets may not do well in detecting nonfeedback bridging faults since activation of the faults is not as likely to occur as in longer test sets. Feedback faults tend to be detected when the nodes of the circuit toggle several times during the application of the test set.



x y	output: fault free	output: in presence of fault
0 0	0	0
0 1	1	1
1 0	1	1
1 1	1	1
0 1	1	1
0 0	0	1
1 0	1	1
1 1	1	1

Figure 6. An OR feedback bridging fault for which the order of tests is important.

Tables 3 and 4 list the results of the simulations for the two- and four-level multiplexer circuits, respectively. The four implementations of the multiplexer are shown in Fig. 7. The mux1 test set is the standard, minimum length, walking zero, walking one multiplexer test set [Makar 87]. Mux2 consists of the same vectors reordered so that the output alternates between zero and one in order to improve feedback coverage. Two pseudo-exhaustive test sets were created following the guidelines discussed below. Mux3 was used for the two level multiplexers and mux4 for the four-level circuit.

Table 3. Simulation results for the two-level multiplexers.

	AND/OR			NAND		
Test sets	mux1	mux2	mux3	mux1	mux2	mux3
Number of test vectors	8	8	12	8	8	12
Number of missed AND faults	0	0	0	0	0	0
Number of missed OR faults	4	0	0	0	0	0
Coverage of AND faults (%)	100.00	100.00	100.00	100.00	100.00	100.00
Coverage of OR faults (%)	94.44	100.00	100.00	100.00	100.00	100.00

Total number of each type of fault: 78

For both two-level implementations and the four-level AND/OR circuit, mux2 caught 100% of the detectable faults. Mux1 failed to catch four OR feedback faults in both AND/OR implementations since the output changed only once. The four-level NAND circuit provided problems for both mux1 and mux2. The number of possible vectors that could detect the

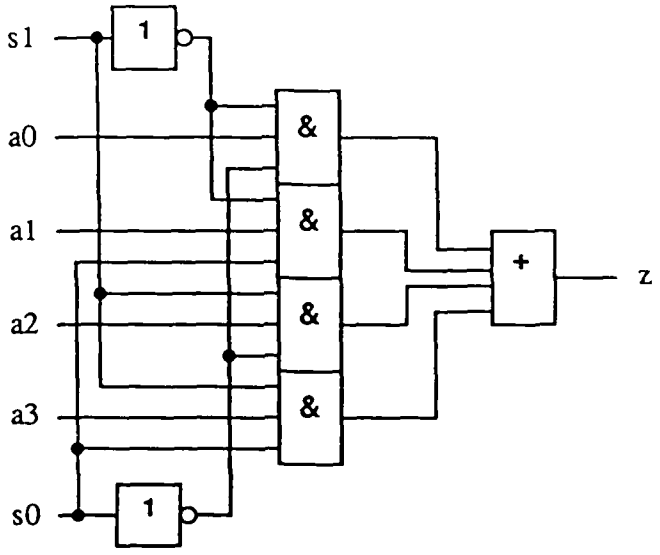
Table 4. Simulation results for the four-level multiplexers.

	AND/OR			NAND		
Test sets	mux1	mux2	mux4	mux1	mux2	mux4
Number of test vectors	8	8	16	8	8	16
Number of missed AND faults	0	0	0	12	12	0
Number of missed OR faults	4	0	0	4	4	0
Coverage of AND faults (%)	100.00	100.00	100.00	90.91	90.91	100.00
Coverage of OR faults (%)	96.92	100.00	100.00	97.04	97.04	100.00

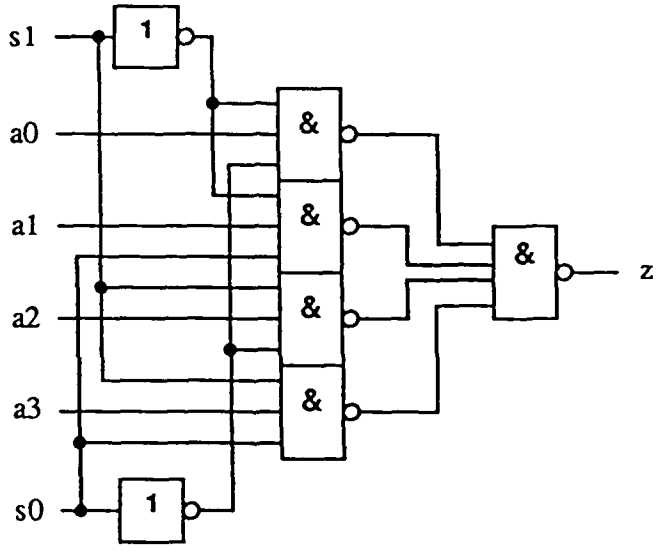
Total number of each type of fault: 136

undetected nonfeedback bridging faults was typically 8 or 12. Since the test sets had only 8 of the 64 possible vectors, the probability was high that none of the detecting vectors were contained in the test sets. This leads to two conclusions. Minimum length test sets achieved excellent coverage of feedback faults, but, as shown above, their coverage of nonfeedback faults is poor. Second, the idea of "functional testing", shown to be inadequate for stuck-at faults in [Sakov 87], is also inadequate for detecting bridging faults. In contrast, the pseudo-exhaustive test sets caught all detectable bridging faults.

Bridging faults unlikely to be detected by a stuck-at test set can be identified as follows. Nonfeedback bridging faults between leads whose logical values rarely, if ever, differ are difficult to detect. Feedback faults are less likely to be detected if leads seldom change value during the application of the test set. These leads can be identified during fault-free simulation of the stuck-at test set. By counting how many times a lead toggles, and whether it always, or nearly always, toggles at the same time as any other leads will allow these bridging faults to be identified without doing fault simulations. Simulators that perform toggle counting are currently used to estimate fault



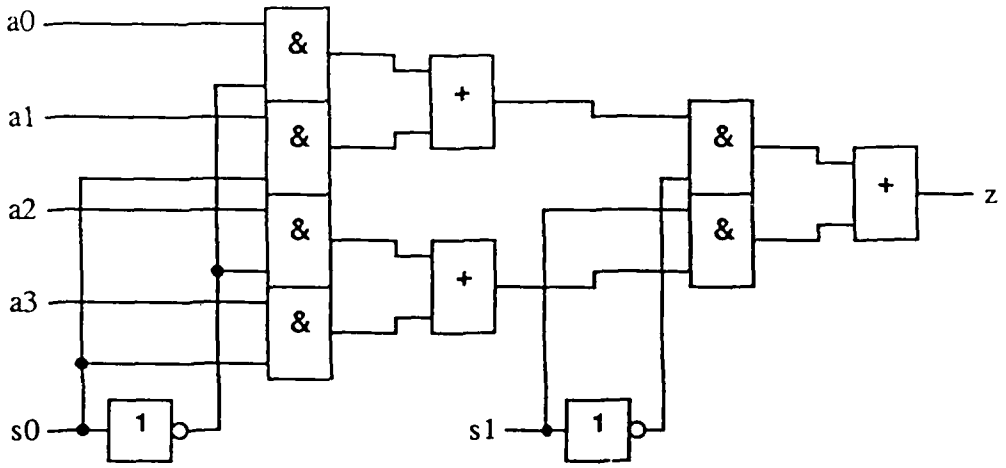
(a) The two-level AND/OR multiplexer.



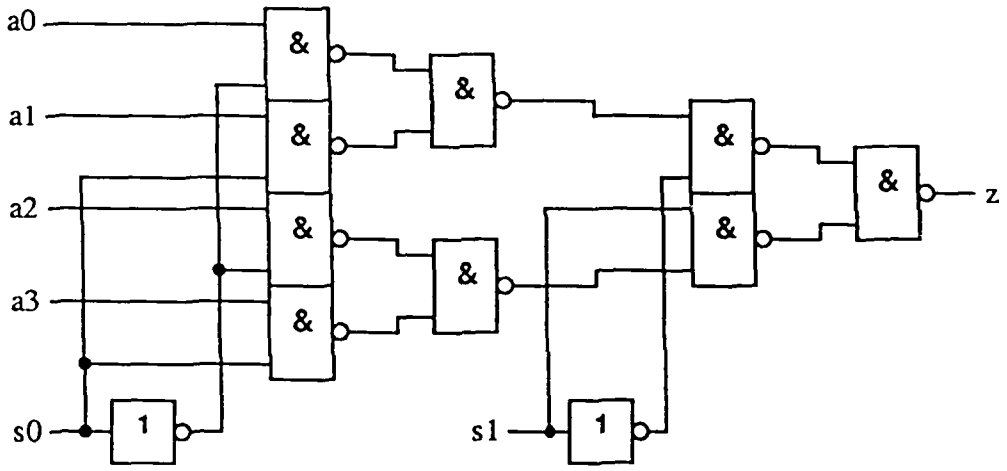
(b) The two-level NAND multiplexer.

Figure 7. The four multiplexer implementations.

coverage and could be adapted to identify bridging faults that are difficult to detect. Bridging faults between outputs of similar gates that share inputs are a subset of the bridging faults between leads whose values rarely differ since such outputs will tend to be the same. However, these faults are easily identified by examining a logic diagram of the circuit.



(c) The four-level AND/OR multiplexer.



(d) The four-level NAND multiplexer

Figure 7. (Continued)

A NEW PSEUDO-EXHAUSTIVE APPROACH FOR BRIDGING FAULTS

The experience gained from simulating the above circuits has shown that pseudo-exhaustive test sets that detect most, if not all, bridging faults can be generated easily. Since pseudo-exhaustive tests will detect any fault within a segment that does not introduce state, all nonfeedback faults within each segment will be detected. If the segments are well chosen, there will be few nonfeedback faults between segments that need attention. Hence, the majority of bridging faults that are of concern are feedback faults. Since fault simulation is unnecessary for the stuck-at faults

when using pseudo-exhaustive testing, fault simulation need only be done for those feedback bridging faults that may be difficult to detect.

After choosing the segments for the circuit under test, the following procedure should be used to create a pseudo-exhaustive test set. First, generate the test patterns for each segment. Second, whether the segments are to be tested serially or in parallel, do not allow inputs to the circuit to always have the same values. The reason for this is shown in Fig. 8 where a bridging fault between w and y will go undetected even though both segments are exhaustively tested. Hence, the circuit and segment inputs should go through all possible combinations, as often as possible, whether they drive the same segment or not. It is shown below that w = y' is not sufficient to guarantee good coverage. Finally, the vectors should be ordered so that all the nodes change value as often as possible, with preference given to the segment outputs. This can be achieved since the test set does not need to test one segment, then a second segment, then a third, etc. Once they have been created, the vectors can be applied in any order. Hence, a random ordering of the vectors, with adjustments following a fault-free simulation of the circuit, can result in a good test set for bridging faults.

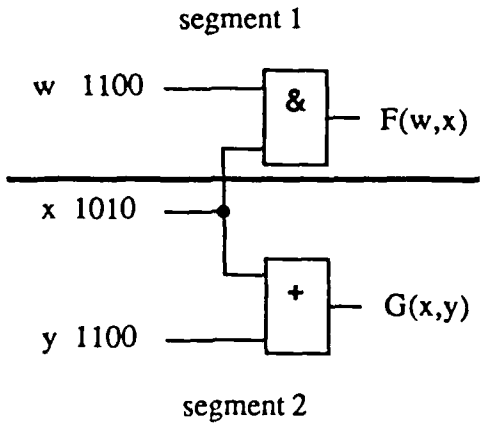


Figure 8. Two segments with inputs that always have the same value (w and y).

For example, in the McC4 test set for the ALU, the first 108 vectors test the right half of the circuit while the final sixteen test the left half. For the final sixteen patterns, it would be possible to set all of the A inputs to the same value and still detect 100% of all stuck-at faults. However, the

coverage of bridging faults would be low since all of the bridges between these inputs would have gone undetected. In addition, this causes the outputs of second level gates in different segments to take on the same values thereby preventing bridging faults between those outputs from being detected. The original pseudo-exhaustive test set, McC1, which had the A inputs as well as the B inputs tied together for the final sixteen vectors, achieved lower results than McC4.

Even when segments are being tested in parallel, the inputs of the separate segments should differ often, and should not be related. When McC3 was created from McC1 by removing the restriction on the A and B inputs, it was found that several easily detected bridging faults were still missed. Two reasons were found that caused this reduction in coverage from the ideal: First, two segments in the right half of the circuit shared many, but not all, inputs. Two of the unshared inputs were always the logical complement of each other during the first 108 vectors. As a result, several faults between the segments between the outputs of similar gates with many shared inputs went undetected. By changing the order of the inputs to one of the segments - a relatively easy task - the problem was solved. The second reason found was that not all of the inputs to separate segments were going through all possible combinations as in the example in Fig. 8. Again, changing the order of the tests for some of the segments solved the problem.

Table 5. Comparison of pseudo-exhaustive test for the ALU.

Test sets	McC1	McC3	McC4
Number of test vectors	124	124	124
Number of missed AND faults	25	3	0
Number of missed OR faults	16	7	1
Coverage of AND faults (%)	99.15	99.90	100.00
Coverage of OR faults (%)	99.45	99.76	99.97

Total number of each type of fault: 2926

McC4, the result of these changes to McC3, performed extremely well. It missed only one bridging fault: an OR feedback bridging fault between distant gates in different segments. A comparison of the results for the three pseudo-exhaustive tests is shown in Table 5. This technique was also used in creating the test sets for the multiplexers. As seen in Tables 3 and 4, these test sets were quite successful.

CONCLUSIONS

This research has shown that stuck-at test sets can provide over 98% coverage of both AND and OR bridging faults in typical circuits. The results may be lower for extremely regular circuits or ultra-short test sets. However, these results are not adequate for today's VLSI circuits. Pseudo-exhaustive test techniques are well suited for detecting bridging faults since they result in extremely high coverage of those faults while guaranteeing 100% detection of stuck-at faults. Bridging fault coverage can be increased by doing fault simulation and test generation for bridging faults that are identified as hard to detect. These bridging faults occur between nodes that rarely, if ever, differ, or that seldom change value. In addition, if the nodes in the fault-free circuit toggle often, feedback faults are easier to detect. This is true even if the nodes involved always have equal values. Methods for identifying such nodes have been presented. These methods use results available from fault-free simulations. A simple solution is to randomly reorder the test vectors to increase toggling and therefore increase bridging fault coverage. As a result, computer time for test generation will be only slightly greater than the time required for stuck-at fault generation alone.

ACKNOWLEDGMENTS

The authors wish to thank John M. Acken for his comments and suggestions, and Laung-Terng Wang and Guntram Wolski for their help with C programming and the UNIX operating system. This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization administered through the Office of Naval Research under Contract No. N00014-85-K-0600. Thanks are due to Aida Corporation for providing the Apollo workstation and the simulator.

REFERENCES

[Abramovici 83] Abramovici, M. and P. Menon, "A Practical Approach to Fault Simulation and Test Generation for Bridging Faults," *Proc. Int. Test Conf.*, pp. 138-142, Philadelphia, PA, Oct. 18-20, 1983.

[Acken 88] Acken, John M., *Deriving Accurate Fault Models*, PhD. Thesis, Stanford University (in preparation).

[Acken 83] Acken, John M., "Testing for Bridging Faults (Shorts) in CMOS Circuits," *20th Des. Autom.Conf.*, pp. 717-778, Miami Beach, FL, June 27-29, 1983.

[Bhattacharya 85] Bhattacharya, B.B., B. Gupta, S. Sarkar, and A.K. Choudhury, "Testable Design of RMC Networks with Universal Tests for Detecting Stuck-at and Bridging Faults," *IEE Proceedings*, Vol. 132, Pt. E, No. 3, pp. 155-161, May 1985.

[Bossen 70] Bossen, D.C., D.L. Ostapko, and A.M. Patel, "Optimum Test Patterns for Parity Networks," *Proc. AFIPS Fall 1970 Joint Comput. Conf.*, Vol. 37, pp. 63-68, Houston, TX, Nov. 1970.

[Freeman 86] Freeman, G., "Development of Logic Level CMOS Bridging Fault Models," Center for Reliable Computing Technical Report 86-10, 1986.

[Hughes 85] Hughes, J.L.A., S. Mourad, and E.J. McCluskey, "An Experimental Study Comparing 74LS181 Test Sets," *COMPCON85*, pp. 384-387, San Francisco, CA, Feb. 26-28, 1985.

[Karpovsky 80] Karpovsky, M. and S.Y.H. Su, "Detection and Location of Input and Feedback Bridging Faults Among Input and Output Lines," *IEEE Trans. Comput.*, C-29, No. 6, pp. 523-527, 1980.

[Makar 87] Makar, S., "On the Testing of Multiplexers," Center for Reliable Computing Technical Report (in preparation).

[Malaiya 86] Malaiya, Y., A.P. Jayasumana and R. Rajsuman, "A Detailed Examination of Bridging Faults," *IEEE Int. Conf. on Comput. Design*, pp. 78-81, Port Chester, NY, Oct. 6-9, 1986.

[McCluskey 86] McCluskey, E.J., *Logic Design Principles*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1986, p. 461.

[Mei 74] Mei, K., "Bridging and Stuck-at Faults," *IEEE Trans. Comput.*, C-23, No. 7, pp. 720-727, 1974.

[Mourad 86] Mourad, S., J.L.A. Hughes, and E.J. McCluskey, "Stuck-At Fault Detection in Parity Trees," Center for Reliable Computing Technical Report 86-7, 1986.

[Sakov 87] Sakov, J. and E.J. McCluskey, "Functional Test Pattern Generation For Random Logic," Center for Reliable Computing Technical Report 87-1, 1987.

[Xu 82] Xu, S. and S. Su, "Testing Feedback Bridging Faults Among Internal, Input and Output Lines by Two Patterns," *IEEE Int.Conf. on Circuits and Computers*, pp. 214-217, New York, NY, Sept. 28 - Oct. 1, 1982.

Appendix A: The Test Sets

```
-------------------------------
design: multiplexer
test set: mux1
author: S. Makar [Makar 87]
method: functional test
-------------------------------
           ssaaaa
           103210
         ---------
         1: 001110
         2: 011101
         3: 101011
         4: 110111
         5: 000001
         6: 010010
         7: 100100
         8: 111000
```

```
-------------------------------
design: multiplexer
test set: mux2
author: S. Makar [Makar 87]
method: functional test
-------------------------------
           ssaaaa
           103210
         ---------
         1: 001110
         2: 000001
         3: 011101
         4: 010010
         5: 101011
         6: 100100
         7: 110111
         8: 111000
```

```
-------------------------------
design: multiplexer
test set: mux3
author: S. Millman and S. Makar
method: pseudo-exhaustive
-------------------------------
           ssaaaa
           103210
         ---------
         1: 000000
         2: 010000
         3: 100000
         4: 110000
         5: 001110
         6: 000001
         7: 011101
         8: 010010
         9: 101011
        10: 100100
        11: 110111
        12: 111000
```

```
-------------------------------
design: multiplexer
test set: mux4
author: S.D. Millman
method: pseudo-exhaustive
-------------------------------
           ssaaaa
           103210
         ---------
         1: 000000
         2: 000001
         3: 001110
         4: 001111
         5: 010000
         6: 011101
         7: 010010
         8: 011111
         9: 100000
        10: 101011
        11: 100100
        12: 101111
        13: 110000
        14: 111000
        15: 110111
        16: 111111
```

```
-------------------------------
design: parity
test set: Bossen
author: D.C. Bossen [Bossen 70]
method: exhaustive test of
        each gate
-------------------------------
           xxxxxxxx
           76543210
         -----------
         1: 00000000
         2: 01110111
         3: 10011101
         4: 11101010
```

```
-------------------------------
design: parity
test set: Bossen2
author: D.C. Bossen [Bossen 70]
method: exhaustive test of
        each gate
-------------------------------
           xxxxxxxx
           76543210
         -----------
         1: 00000000
         2: 01010111
         3: 10111101
         4: 11101010
```

```
-------------------------------
design: parity
test set: Millman
author: S. Millman
method: pseudo-exhaustive
-------------------------------
          xxxxxxxx
          76543210
        -----------
        1: 00000000
        2: 00000001
        3: 00000010
        4: 00000011
        5: 00000100
        6: 00001000
        7: 00001100
        8: 00010000
        9: 00100000
       10: 00110000
       11: 01000000
       12: 10000000
       13: 11000000
```

```
-------------------------------
design: parity
test set: Mourad
author: S. Mourad [Mourad 86]
method: augmentation of Bossen
-------------------------------
          xxxxxxxx
          76543210
        -----------
        1: 11101010
        2: 01111101
        3: 10010111
        4: 00000000
        5: 01001011
        6: 11100101
        7: 11001110
```

```
-------------------------------
design: ALU
test set: Bryant2
author: R. Bryant [Hughes 85]
method: test pattern generation
        program
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        ----------------
        1: 00101000000001
        2: 00101000100001
        3: 10100111110001
        4: 10100110001101
        5: 01100011101000
        6: 10011001011101
        7: 01011101100100
        8: 01011100111001
        9: 11010111000000
```

```
-------------------------------
test set: Bryant2 (cont.)
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        ----------------
       10: 01001111111110
       11: 01001000111111
       12: 11101011100011
       13: 10010110100011
       14: 11011000100110
```

```
-------------------------------
design: ALU
test set: Bryant6
author: R. Bryant [Hughes 85]
method: test pattern generation
        program
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        ----------------
        1: 10100111110001
        2: 11100011001001
        3: 01001110111000
        4: 01010000001100
        5: 10011011010101
        6: 11100101100011
        7: 01111101100100
        8: 10010000110100
        9: 10100100001001
       10: 01111111010001
       11: 01101011010110
       12: 10011000110011
```

```
-------------------------------
design: ALU
test set: Goel
author: P. Goel [Hughes 85]
method: test pattern generation
        program
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        ----------------
        1: 10011000010010
        2: 00100001000001
        3: 00000001001010
        4: 11001111100000
        5: 00011101011001
        6: 01110000100011
        7: 11100001000111
        8: 00010110000001
        9: 10010111001000
       10: 11110100110001
       11: 11111000100001
       12: 10010100010011
       13: 10110000001111
       14: 01101101000110
```

```
-------------------------------
test set: Goel (cont.)
-------------------------------
            ssssbbbbaaaa c
            321032103210mn
        ------------------
     15: 01100000000110
     16: 10101001101101
     17: 11100010000011
     18: 01001011001000
     19: 01000000001000
     20: 01110000010011
     21: 01111100010000
     22: 11010010001000
     23: 10100000101000
     24: 01101011100001
     25: 10100110011001
     26: 10101110110000
     27: 11101010101001
     28: 10101001100000
     29: 10101011101001
     30: 10001101110000
     31: 10101010100000
     32: 01100101001111
     33: 10100110010100
     34: 10100101010000
     35: 10100011001011
```

```
-------------------------------
design: ALU
test set: Hughes
author: J. Hughes [Hughes 85]
method: pseudo-random
-------------------------------
            ssssbbbbaaaa c
            321032103210mn
        -----------------
      1: 11111111111111
      2: 10111111111001
      3: 10011111111010
      4: 01001111111101
      5: 11100111111000
      6: 01110111011100
      7: 00111011011110
      8: 00011011101111
      9: 11001101101001
     10: 10100101110010
     11: 01010110010101
     12: 11101010011100
     13: 01110011101010
     14: 00111101001101
     15: 11011001110000
     16: 01101110100100
     17: 00110100111010
     18: 00011111010001
     19: 11001010111010
     20: 01100111101001
     21: 11110101011010
     22: 01111011010101
     23: 11111010101000
```

```
-------------------------------
test set: Hughes (cont.)
-------------------------------
            ssssbbbbaaaa c
            321032103210mn
        ------------------
     24: 01111101101000
     25: 00111101110100
     26: 00011110110110
     27: 00001110111011
     28: 11000111111111
     29: 10100111011001
     30: 10010011011010
     31: 01001011001101
     32: 11100001101000
     33: 01110101000100
     34: 00111000010110
     35: 00011010100011
     36: 11001100101111
     37: 10100101110101
     38: 10010110010000
     39: 01001010011000
     40: 00100011101000
     41: 00010101001100
     42: 00001001010110
     43: 00000010100111
     44: 11000100001101
     45: 10100001010100
     46: 01010010000110
     47: 00101000001011
     48: 11010001100111
     49: 10101100000001
     50: 10010000110110
     51: 01001110000011
     52: 11100000111111
     53: 10110111000101
     54: 10011000011000
     55: 01001011100000
     56: 00100100101100
     57: 00010101010010
     58: 00001010010101
     59: 11000010101100
     60: 01100101001010
     61: 00110001010101
     62: 11011010000000
     63: 01101000101000
     64: 00110101100000
     65: 00011100010100
     66: 00001010110010
     67: 00000110101001
     68: 11000101011110
     69: 01100011010111
     70: 11110010001001
     71: 10111001001110
     72: 01011001100111
     73: 11101100100001
     74: 10110100110110
     75: 01011110010011
     76: 11101010111111
     77: 10110111101101
     78: 10011101011000
```

test set: Hughes (cont.)

```
            ssssbbbbaaaa c
            321032103210mn
        ------------------
         79: 01001011110100
         80: 00100110101110
         81: 00010101011011
         82: 11001011010011
         83: 10100010101011
         84: 10010101001111
         85: 10001001010001
         86: 10000010100010
         87: 01000100001001
         88: 11100001010110
         89: 01110010000111
         90: 11111000001101
         91: 10111001100100
         92: 01011100100110
         93: 00101100110011
         94: 11010110110111
         95: 10101110011101
         96: 10010011111100
         97: 01001111001110
         98: 00100001111111
         99: 11010111000001
        100: 10101000011010
        101: 01010011100001
        102: 11101100001010
        103: 01110001110001
        104: 11111110000010
        105: 01111000111001
        106: 11111111100110
        107: 01111100111111
        108: 11111111110101
        109: 10111110111000
        110: 01011111111000
        111: 00101111111100
        112: 00010111111110
        113: 00001111011111
        114: 11000011111001
        115: 10100111001010
        116: 01010001011101
        117: 11101011000000
        118: 01110000101100
        119: 00111101000010
        120: 00011000110101
        121: 11001110100100
        122: 01100100111010
        123: 00110111010001
        124: 11011010011010
        125: 01101011101001
        126: 11110101101010
        127: 01111101010101
        128: 11111010110000
        129: 01111110101000
        130: 00111101111000
        131: 00011111110100
        132: 00001110111110
        133: 00000111111011
```

test set: Hughes (cont.)

```
            ssssbbbbaaaa c
            321032103210mn
        ------------------
        134: 11000111011011
        135: 10100011011011
```

design: ALU
test set: Krish
author: B. Krishnamurthy
[Hughes 85]
method: test pattern generation program

```
            ssssbbbbaaaa c
            321032103210mn
        ------------------
          1: 01011111000001
          2: 01011111111100
          3: 01011110000101
          4: 01011100001001
          5: 01010000100011
          6: 01011000010001
          7: 01010000001100
          8: 10100001000000
          9: 10100010110000
         10: 10100101001100
         11: 10101110111110
         12: 10101011001111
```

design: ALU
test set: McCl
author: E.J. McCluskey
[Hughes 85]
method: pseudo-exhaustive

```
            ssssbbbbaaaa c
            321032103210mn
        ------------------
          1: 10101000011100
          2: 10100000011101
          3: 10101000111111
          4: 10101000111110
          5: 10101001011000
          6: 10100001011001
          7: 10101001111011
          8: 10101001111010
          9: 10101001011100
         10: 10100001011101
         11: 10101001111111
         12: 10101001111110
         13: 10101010010100
         14: 10100010010101
         15: 10101010110111
         16: 10101010110110
```

test set: McCl (cont.)

	ssssbbbbaaaa c 321032103210mn
17:	10101011010000
18:	10100011010001
19:	10101011110011
20:	10101011110010
21:	10101011010100
22:	10100011010101
23:	10101011110111
24:	10101011110110
25:	10101010011100
26:	10100010011101
27:	10101010111111
28:	10101010111110
29:	10101011011000
30:	10100011011001
31:	10101011111011
32:	10101011111010
33:	10101011011100
34:	10100011011101
35:	10101011111111
36:	10101011111110
37:	10101100001100
38:	10100100001101
39:	10101100101111
40:	10101100101110
41:	10101101001000
42:	10100101001001
43:	10101101101011
44:	10101101101010
45:	10101101001100
46:	10100101001101
47:	10101101101111
48:	10101101101110
49:	10101110000100
50:	10100110000101
51:	10101110100111
52:	10101110100110
53:	10101111000000
54:	10100111000001
55:	10101111100011
56:	10101111100010
57:	10101111000100
58:	10100111000101
59:	10101111100111
60:	10101111100110
61:	10101110001100
62:	10100110001101
63:	10101110101111
64:	10101110101110
65:	10101111001000
66:	10100111001001
67:	10101111101011
68:	10101111101010
69:	10101111001100
70:	10100111001101
71:	10101111101111

test set: McCl (cont.)

	ssssbbbbaaaa c 321032103210mn
72:	10101111101110
73:	10101100011100
74:	10100100011101
75:	10101100111111
76:	10101100111110
77:	10101101011000
78:	10100101011001
79:	10101101111011
80:	10101101111010
81:	10101101011100
82:	10100101011101
83:	10101101111111
84:	10101101111110
85:	10101110010100
86:	10100110010101
87:	10101110110111
88:	10101110110110
89:	10101111010000
90:	10100111010001
91:	10101111110011
92:	10101111110010
93:	10101111010100
94:	10100111010101
95:	10101111110111
96:	10101111110110
97:	10101110011100
98:	10100110011101
99:	10101110111111
100:	10101110111110
101:	10101111011000
102:	10100111011001
103:	10101111111011
104:	10101111111010
105:	10101111011100
106:	10100111011101
107:	10101111111111
108:	10101111111110
109:	00000000000010
110:	00000000111110
111:	00001111000010
112:	00001111111110
113:	01010000000010
114:	01010000111110
115:	01011111000010
116:	01011111111110
117:	10100000000010
118:	10100000111110
119:	10101111000010
120:	10101111111110
121:	11110000000010
122:	11110000111110
123:	11111111000010
124:	11111111111110

```
-------------------------------
design: ALU
test set: McC3
author: E.J. McCluskey
method. pseudo-exhaustive
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        -----------------
        1: 10101000011100
        2: 10100000011101
        3: 10101000111111
        4: 10101000111110
        5: 10101001011000
        6: 10100001011001
        7: 10101001111011
        8: 10101001111010
        9: 10101001011100
       10: 10100001011101
       11: 10101001111111
       12: 10101001111110
       13: 10101010010100
       14: 10100010010101
       15: 10101010110111
       16: 10101010110110
       17: 10101011010000
       18: 10100011010001
       19: 10101011110011
       20: 10101011110010
       21: 10101011010100
       22: 10100011010101
       23: 10101011110111
       24: 10101011110110
       25: 10101010011100
       26: 10100010011101
       27: 10101010111111
       28: 10101010111110
       29: 10101011011000
       30: 10100011011001
       31: 10101011111011
       32: 10101011111010
       33: 10101011011100
       34: 10100011011101
       35: 10101011111111
       36: 10101011111110
       37: 10101100001100
       38: 10100100001101
       39: 10101100101111
       40: 10101100101110
       41: 10101101001000
       42: 10100101001001
       43: 10101101101011
       44: 10101101101010
       45: 10101101001100
       46: 10100101001101
       47: 10101101101111
       48: 10101101101110
       49: 10101110000100
       50: 10100110000101
       51: 10101110100111
       52: 10101110100110
```

```
-------------------------------
test set: McC3 (cont.)
-------------------------------
          ssssbbbbaaaa c
          321032103210mn
        -----------------
       53: 10101111000000
       54: 10100111000001
       55: 10101111100011
       56: 10101111100010
       57: 10101111000100
       58: 10100111000101
       59: 10101111100111
       60: 10101111100110
       61: 10101110001100
       62: 10100110001101
       63: 10101110101111
       64: 10101110101110
       65: 10101111001000
       66: 10100111001001
       67: 10101111101011
       68: 10101111101010
       69: 10101111001100
       70: 10100111001101
       71: 10101111101111
       72: 10101111101110
       73: 10101100011100
       74: 10100100011101
       75: 10101100111111
       76: 10101100111110
       77: 10101101011000
       78: 10100101011001
       79: 10101101111011
       80: 10101101111010
       81: 10101101011100
       82: 10100101011101
       83: 10101101111111
       84: 10101101111110
       85: 10101110010100
       86: 10100110010101
       87: 10101110110111
       88: 10101110110110
       89: 10101111010000
       90: 10100111010001
       91: 10101111110011
       92: 10101111110010
       93: 10101111010100
       94: 10100111010101
       95: 10101111110111
       96: 10101111110110
       97: 10101110011100
       98: 10100110011101
       99: 10101110111111
      100: 10101110111110
      101: 10101111011000
      102: 10100111011001
      103: 10101111111011
      104: 10101111111010
      105: 10101111011100
      106: 10100111011101
      107: 10101111111111
```

---

test set: McC3 (cont.)

```
            ssssbbbbaaaa c
            321032103210mn
        -------------------
        108: 10101111111110
        109: 10001100011010
        110: 10000110001110
        111: 10000011100110
        112: 10001001110010
        113: 11101100011010
        114: 11100110001110
        115: 11100011100110
        116: 11101001110010
        117: 00111100011010
        118: 00110110001110
        119: 00110011100110
        120: 00111001110010
        121: 01011100011010
        122: 01010110001110
        123: 01010011100110
        124: 01011001110010
```

---

design: ALU
test set: McC4
author: E.J. McCluskey and
S.D. Millman
method: pseudo-exhaustive

```
            ssssbbbbaaaa c
            321032103210mn
        -------------------
         1: 10101000011100
         2: 10100000011111
         3: 10101000111101
         4: 10101000111110
         5: 10101001011000
         6: 10100001011011
         7: 10101001111001
         8: 10101001111010
         9: 10101001011100
        10: 10100001011111
        11: 10101001111101
        12: 10101001111110
        13: 10101010010100
        14: 10100010010111
        15: 10101010110101
        16: 10101010110110
        17: 10101011010000
        18: 10100011010011
        19: 10101011110001
        20: 10101011110010
        21: 10101011010100
        22: 10100011010111
        23: 10101011110101
        24: 10101011110110
        25: 10101010011100
        26: 10100010011111
```

---

test set: McC4 (cont.)

```
            ssssbbbbaaaa c
            321032103210mn
        -------------------
        27: 10101010111101
        28: 10101010111110
        29: 10101011011000
        30: 10100011011011
        31: 10101011111001
        32: 10101011111010
        33: 10101011011100
        34: 10100011011111
        35: 10101011111101
        36: 10101011111110
        37: 10101100001100
        38: 10100100001111
        39: 10101100101101
        40: 10101100101110
        41: 10101101001000
        42: 10100101001011
        43: 10101101101001
        44: 10101101101010
        45: 10101101001100
        46: 10100101001111
        47: 10101101101101
        48: 10101101101110
        49: 10101110000100
        50: 10100110000111
        51: 10101110100101
        52: 10101110100110
        53: 10101111000000
        54: 10100111000011
        55: 10101111100001
        56: 10101111100010
        57: 10101111000100
        58: 10100111000111
        59: 10101111100101
        60: 10101111100110
        61: 10101110001100
        62: 10100110001111
        63: 10101110101101
        64: 10101110101110
        65: 10101111001000
        66: 10100111001011
        67: 10101111101001
        68: 10101111101010
        69: 10101111001100
        70: 10100111001111
        71: 10101111101101
        72: 10101111101110
        73: 10101100011100
        74: 10100100011111
        75: 10101100111101
        76: 10101100111110
        77: 10101101011000
        78: 10100101011011
        79: 10101101111001
        80: 10101101111010
        81: 10101101011100
```

test set: McC4 (cont.)

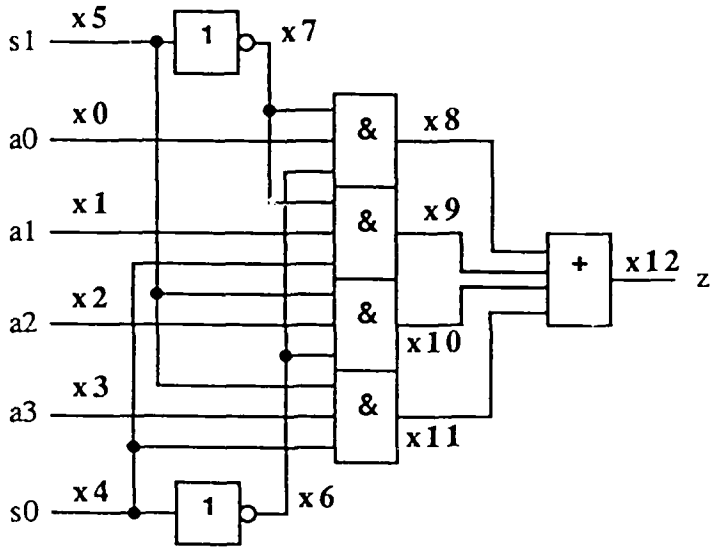
```
            ssssbbbbaaaa c
            321032103210mn
         ------------------
         82: 10100101011111
         83: 10101101111101
         84: 10101101111110
         85: 10101110010100
         86: 10100110010111
         87: 10101110110101
         88: 10101110110110
         89: 10101111010000
         90: 10100111010011
         91: 10101111110001
         92: 10101111110010
         93: 10101111010100
         94: 10100111010111
         95: 10101111110101
         96: 10101111110110
         97: 10101110011100
         98: 10100110011111
         99: 10101110111101
        100: 10101110111110
        101: 10101111011000
        102: 10100111011011
        103: 10101111111001
        104: 10101111111010
        105: 10101111011100
        106: 10100111011111
        107: 10101111111101
        108: 10101111111110
        109: 00000000000010
        110: 11111111111110
        111: 11100010110010
        112: 11101001001010
        113: 11100100000110
        114: 10011001101010
        115: 10011100010110
        116: 10010110101010
        117: 10010011010110
        118: 00111010010110
        119: 00110101101010
        120: 01000111100110
        121: 01001101011010
        122: 01001010111110
        123: 01110000000010
        124: 00101111111110
```

design: ALU
test set: miczo2
author: A. Miczo [Hughes 85]
method: compressed random

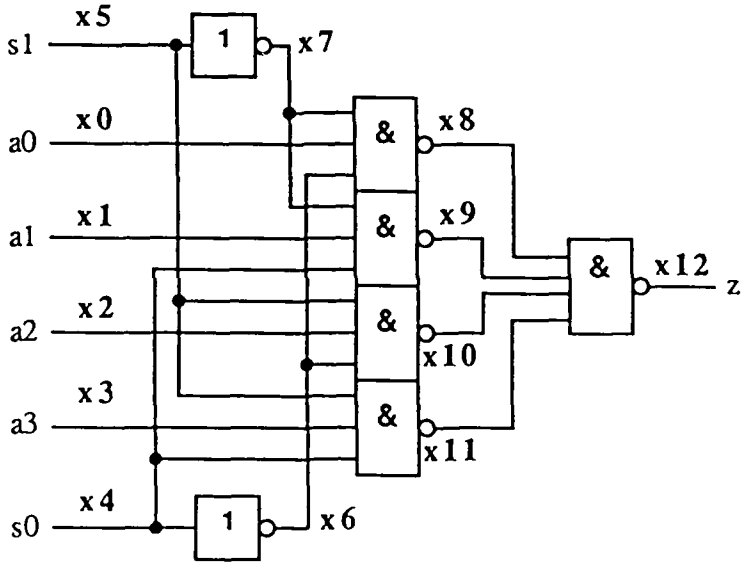
```
            ssssbbbbaaaa c
            321032103210mn
         ------------------
          1: 10010110011001
          2: 10011100110001
          3: 10011000100000
          4: 10010100110001
          5: 10010010111001
          6: 10010001111101
          7: 10010001111000
          8: 10010000111101
          9: 10011110000000
         10: 01101010100100
         11: 01100101011001
         12: 01101111001101
         13: 01001100000011
         14: 01100000110001
         15: 10010001110101
         16: 10010001101100
         17: 10011000111101
```

Appendix B: Missed and Undetectable Faults

The circuits used for the simulations are repeated below with all of the nodes used for bridging faults labeled. The lists of undetected and undetectable faults follow.

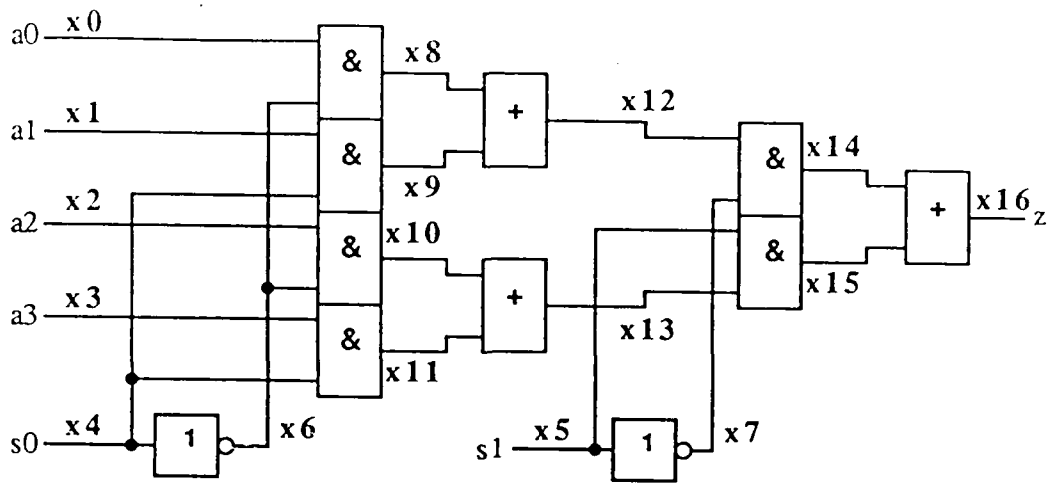


(a) The two-level AND/OR multiplexer.

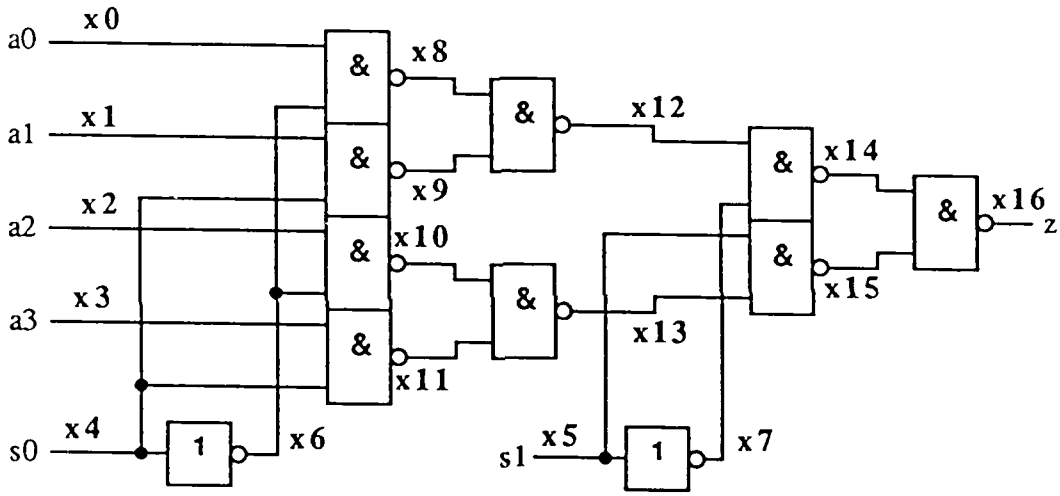


(b) The two-level NAND multiplexer.

Figure 9. The four implementations of the multiplexer.



(c) The four-level AND/OR multiplexer.



(d) The four-level NAND multiplexer.

Figure 9. (Continued)

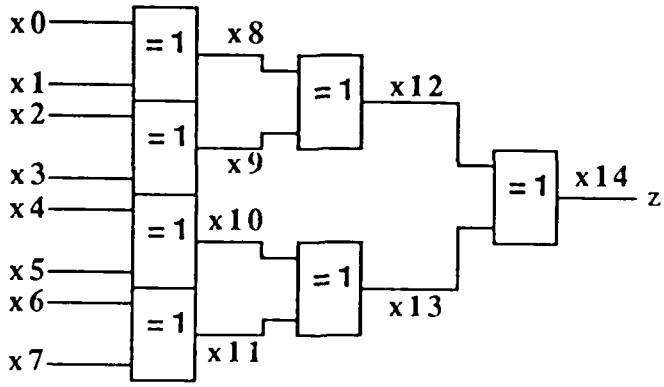


Figure 10. The parity tree.

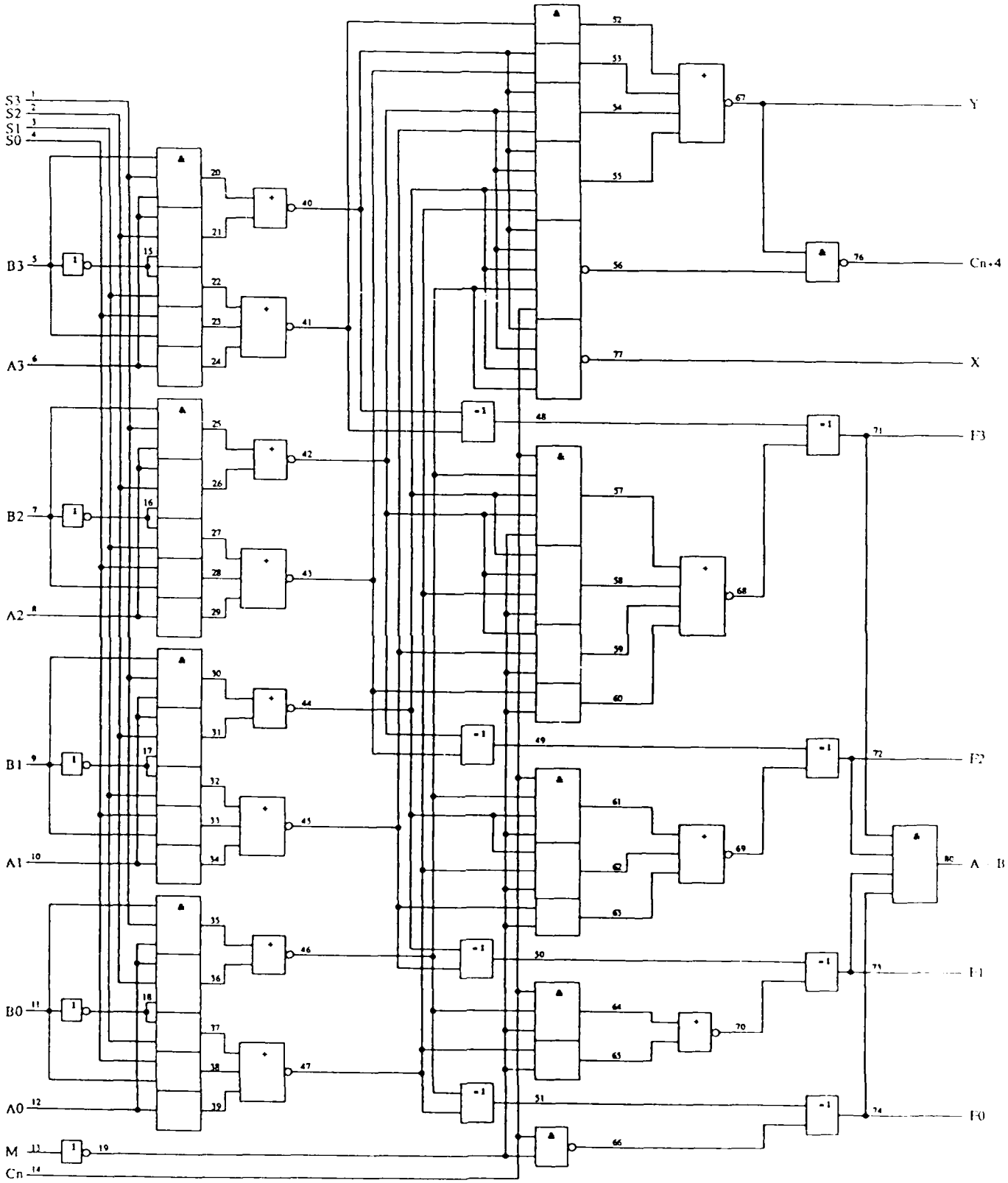


Figure 11. The 74LS181 16-function ALU

Bossen missed ANDs

x0 x2
x0 x4
x0 x11
x1 x5
x1 x6
x1 x9
x1 x13
x2 x4
x2 x11
x3 x7
x3 x8
x3 x10
x4 x11
x4 x12
x5 x6
x5 x9
x6 x9
x7 x8
x7 x10
x8 x10
x9 x13
x11 x12

Bossen missed ORs

x0 x2
x0 x4
x0 x11
x1 x5
x1 x6
x1 x9
x1 x13
x2 x4
x2 x11
x3 x7
x3 x8
x3 x10
x3 x14
x4 x11
x4 x12
x5 x6
x5 x9
x6 x9
x7 x8
x7 x10
x7 x14
x8 x10
x8 x14
x9 x13
x10 x14
x11 x12

Bossen2 missed ORs

x0 x2
x0 x4
x0 x11
x1 x6
x1 x9
x1 x10
x2 x4
x2 x11
x3 x5
x3 x7
x3 x8
x3 x13
x4 x11
x4 x12
x5 x7
x5 x8
x6 x9
x6 x10
x7 x8
x8 x13
x9 x10
x11 x12

Bossen2 missed ANDs

x0 x2
x0 x4
x0 x11
x1 x6
x1 x9
x1 x10
x2 x4
x2 x11
x3 x5
x3 x7
x3 x8
x3 x13
x4 x11
x4 x12
x5 x7
x5 x8
x6 x9
x6 x10
x7 x8
x8 x13
x9 x10
x11 x12

mux1 two-level AND/OR missed ORs

x8 x12
x9 x12
x10 x12
x11 x12

mux1 four-level AND/OR missed ORs

x8 x12
x9 x12
x14 x16
x15 x16

mux1 four-level NAND missed ORs

x0 x10
x1 x11
x2 x8
x3 x9

mux1 four-level NAND missed ANDs

x0 x10
x0 x11
x1 x10
x1 x11
x2 x8
x2 x9
x3 x8
x3 x9
x8 x13
x9 x10
x10 x12
x11 x12

mux2 four-level NAND missed ORs

x0 x10
x1 x11
x2 x8
x3 x9

mux2 four-level NAND missed ANDs

x0 x10
x0 x11
x1 x10
x1 x11
x2 x8
x2 x9
x3 x8
x3 x9
x8 x13
x9 x13
x10 x12
x11 x12

Bryant2 missed ANDs	Bryant6 missed ANDs	Goel missed ANDs	Krish missed ANDs	McC1 missed ANDs	McC3 missed ANDs	Miczo2 missed ANDs
t17 t18	t18 t53	t4 t52	t1 t3	t1 t3	t54 t59	t1 t4
t21 t58	t18 t60	t28 t33	t2 t4	t2 t4	t55 t58	t2 t3
t21 t62	t23 t33	t28 t52	t4 t14	t3 t52	t56 t68	t4 t55
t21 t65	t23 t38	t31 t53	t8 t54	t16 t17		t4 t58
t22 t30	t26 t52		t8 t59	t16 t18		t4 t62
t22 t53	t27 t30		t8 t63	t17 t18		t4 t65
t22 t60	t28 t38		t14 t23	t21 t26		t15 t17
t23 t35	t31 t36		t20 t25	t21 t31		t15 t32
t30 t53	t31 t60		t22 t32	t21 t36		t16 t18
t30 t60	t33 t38		t32 t55	t23 t28		t16 t37
t32 t37	t36 t60		t32 t58	t23 t33		t17 t22
t32 t57	t37 t53		t32 t62	t23 t38		t18 t27
t32 t61	t38 t57		t32 t65	t26 t31		t21 t26
t37 t57	t53 t60		t37 t54	t26 t36		t21 t32
t37 t61	t54 t59		t37 t59	t27 t32		t21 t37
t37 t64	t54 t63		t37 t63	t27 t37		t22 t31
t53 t60	t55 t58		t38 t57	t28 t33		t22 t32
t54 t59	t55 t62		t38 t61	t28 t38		t23 t28
t54 t63	t55 t65		t38 t64	t31 t36		t23 t33
t57 t61	t56 t68		t53 t60	t32 t37		t23 t55
t58 t62	t57 t61		t54 t59	t33 t38		t23 t58
t58 t65	t57 t64		t54 t63	t53 t60		t23 t62
t59 t63	t58 t62		t55 t58	t54 t59		t23 t65
t62 t65	t58 t65		t55 t62	t55 t58		t26 t32
	t59 t63		t55 t65	t56 t68		t26 t37
	t61 t64		t56 t68			t27 t36
	t62 t65		t57 t61			t27 t37
	t46 t51		t57 t64			t28 t33
			t58 t62			t28 t55
			t58 t65			t28 t58
			t59 t63			t28 t62
			t61 t64			t32 t37
			t62 t65			t33 t55
						t33 t58
						t33 t62
						t39 t57
						t53 t60
						t54 t59
						t54 t63
						t55 t58
						t55 t62
						t58 t62
						t58 t65
						t59 t63
						t61 t64
						t62 t65

Bryant2 missed ORs	Bryant6 missed ORs	Goel missed ORs	Krish missed ORs	Krish missed ORs (cont.)	McC1 missed ORs	McC3 missed ORs	Miczo2 missed ORs
t6 t22	t1 t15	t10 t33	t1 t3		t1 t3	t6 t23	t1 t4
t8 t23	t1 t22	t23 t27	t2 t4	t53 t58	t2 t4	t8 t28	t6 t16
t10 t32	t5 t28	t31 t55	t5 t28	t53 t59	t21 t26	t11 t33	t13 t80
t12 t37	t8 t28	t33 t53	t6 t8	t53 t60	t21 t31	t12 t38	t15 t17
t21 t58	t9 t27	t33 t54	t6 t22	t54 t58	t21 t36	t55 t58	t16 t18
t22 t65	t9 t28		t6 t23	t54 t59	t23 t28	t55 t59	t16 t23
t22 t77	t12 t37		t6 t46	t54 t60	t23 t33	t55 t60	t16 t24
t30 t53	t12 t38		t7 t33	t55 t58	t23 t38		t21 t26
t30 t55	t14 t22		t8 t23	t55 t59	t26 t31		t22 t27
t30 t57	t23 t28		t8 t24	t55 t60	t26 t36	-------	t22 t32
t30 t58	t23 t49		t8 t27	t57 t80	t26 t80	McC4	t22 t67
t30 t60	t26 t54		t8 t28	t58 t80	t28 t33	missed	t23 t62
t32 t37	t26 t55		t8 t46		t28 t38	ORs	t23 t63
t32 t57	t28 t33		t9 t37		t31 t36	-------	t27 t37
t32 t58	t28 t38		t9 t38		t31 t80		t36 t57
t32 t59	t31 t36		t10 t32		t33 t38	t31 t55	t36 t61
t32 t60	t31 t55		t10 t33				t36 t80
t32 t63	t31 t57		t11 t32				t37 t62
t32 t76	t31 t58		t12 t33				
t33 t52	t33 t37		t12 t37				
t33 t53	t33 t38		t12 t38				
t33 t54	t36 t57		t13 t71				
t33 t55	t37 t52		t15 t72				
t33 t76	t37 t53		t20 t25				
t36 t74	t37 t54		t21 t80				
t37 t57	t37 t55		t22 t44				
t37 t58	t37 t57		t22 t52				
t37 t59	t37 t58		t22 t54				
t37 t60	t37 t59		t22 t55				
t37 t65	t37 t60		t22 t57				
t37 t72	t37 t62		t22 t58				
t53 t59	t37 t65		t22 t59				
t53 t60	t37 t74		t22 t71				
t54 t59	t37 t76		t23 t28				
t54 t60	t53 t58		t23 t29				
	t53 t59		t24 t46				
	t53 t60		t27 t53				
	t54 t58		t27 t54				
	t54 t59		t27 t55				
	t54 t60		t28 t33				
	t55 t58		t32 t52				
	t55 t59		t32 t53				
	t55 t60		t32 t54				
	t58 t80		t32 t55				
			t32 t57				
			t32 t58				
			t32 t59				
			t32 t60				
			t32 t76				
			t33 t38				
			t33 t39				
			t35 t52				
			t35 t53				
			t37 t51				
			t38 t51				

Undetectable OR faults in the two-level AND/OR multiplexer

x8 x9
x8 x10
x8 x11
x9 x10
x9 x11
x10 x11

Undetectable AND faults in the two-level NAND multiplexer

x8 x9
x8 x10
x8 x11
x9 x10
x9 x11
x10 x11

Undetectable AND faults in the four-level AND/OR multiplexer

x7 x12

Undetectable OR faults in the four-level AND/OR multiplexer

x7 x10
x7 x11
x7 x13
x8 x9
x10 x11
x14 x15

Undetectable AND faults in the four-level NAND multiplexer

x7 x12
x8 x9
x10 x11
x14 x15

Undetectable OR faults in the four-level AND/OR multiplexer

x7 x13

Undetectable OR faults in the ALU

t20 t21
t22 t23
t22 t24
t23 t24
t25 t26
t27 t28
t27 t29
t28 t29
t30 t31
t32 t33
t32 t34
t33 t34
t35 t36
t37 t38
t37 t39
t38 t39
t52 t53
t52 t54
t52 t55
t53 t54
t53 t55
t54 t55
t57 t58
t57 t59
t57 t60
t58 t59
t58 t60
t59 t60
t61 t62
t61 t63
t62 t63
t64 t65

END
DATE
FILMED
5-88
DTIC