An Integrated Test Plan for an Advanced Very Large Scale Integrated Circuit Design Group

William S. Didden
University of Central Florida

AN INTEGRATED TEST PLAN FOR AN ADVANCED VERY LARGE SCALE INTEGRATED CIRCUIT DESIGN GROUP

BY

WILLIAM S. DIDDEN
B.S., Lehigh University, 1973

RESEARCH REPORT

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program
of the College of Engineering
University of Central Florida
Orlando, Florida

Fall Term
1984

ABSTRACT

VLSI testing poses a number of problems which includes the selection of test techniques, the determination of acceptable fault coverage levels, and test vector generation. Available device test techniques are examined and compared. Design rules should be employed to assure the design is testable. Logic simulation systems and available test utilities are compared. The various methods of test vector generation are also examined.

The selection criteria for test techniques are identified. A table of proposed design rules is included. Testability measurement utilities can be used to statistically predict the test generation effort. Field reject rates and fault coverage are statistically related. Acceptable field reject rates can be achieved with less than full test vector fault coverage. The methods and techniques which are examined form the basis of the recommended integrated test plan. The methods of automatic test vector generation are relatively primitive but are improving.

ACKNOWLEDGEMENTS

The information provided in this report is based on the work and theories of a great many technical specialists who have contributed much toward furthering the state of very large scale integrated circuit testing. Their contributions are gratefully acknowledged.

The author is especially indebted to Tom Hoffman and Dan Scheflin who guided and encouraged him, Brian Petrasko for his guidance and suggestions in organizing the information, Willie Mack for her assistance in word processing, and Mary Jo Didden for her proofreading and patience.

PREFACE

Testing is becoming a major portion of the AVLSI design effort. As many smaller design groups begin to design VLSI circuits, they encounter testing problems, often from a lack of information or attempting to follow older methods established for board level designs.

The old methods of testing focused on board level designs which employed largely combinatorial, small and medium scale integrated circuits installed on assemblies with numerous, easily probed test points. The test strategies and their development were frequently relegated to a test engineer. The simplicity of the board function often permitted the test engineer to work independently of the design engineer. Test development often did not begin until after the design was completed and possibly even assembled. However, the resulting test coverage was often fairly high for a small number of test vectors. As a result, this method of testing was considered adequate.

The development of Very Large Scale Integrated (VLSI) circuits created a number of testing problems. The circuits are often sequential, difficult to probe, and have an extremely limited number of input/output pins which can serve as test points. Frequently, the available test points must be shared with other functions during normal operation. The device complexity requires the designer and test engineer to exchange considerable amounts of information. Adequate test coverage typically requires several thousand test vectors.

These testing problems have been solved by a number of new methods. More I/O economical test design strategies are applied during the early design phases. The information transfer between the design and test engineers is avoided by requiring the designer to assume all the test development responsibilities. Test vectors must be developed in the logic simulation phase to verify correct circuit operation. As a result, testing is becoming a major design effort which requires integration at most design levels.

This effort is an attempt to suggest an integrated test development plan. The test development plan includes a number of steps which are not currently performed. Their inclusion will reduce the surprise element previously associated with VLSI circuit test.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Testing Requirements of the AVLSI Design Group

An integrated test plan is required for AVLSI designs. The test plan must provide both design guidance and recommended testing to the VLSI designer during all design, manufacturing and assembly phases. The guidance and recommended testing required during each phase are discussed below.

The integrated test plan must aid the designer's selection of a test technique and guide the designer to assure the IC design is easily testable. In addition, the test plan should recommend methods to assess the testability of the final design. The achieved testability levels should be compared to previously established design goals.

The test plan must include a method to verify the new IC design. The test results will be used to correct the manufacturing process. In addition, a reduced, go/no go test must be provided for device acceptance tests and subsequent go/no go device tests administered prior to final assembly.

The integrated test plan must provide information to aid other testing groups. The information, provided according to the test plan, must aid in the development of board level tests on existing test systems. In addition, the test plan should provide information to aid in developing system tests.

Approach

The test plan requirements can be achieved by:

1. providing a handbook of test methods and techniques,

2. establishing economical testability design goals,

3. providing design guidelines during the design phase to assure the device is testable,

4. requiring the designer to verify by simulation that testability design goals have been achieved, and

5. discussing the currently available methods of test vector generation.

The first two items are addressed in Chapter I. Appropriate design examples are included in Appendices B and C. The design guidelines and verification by simulation are examined in Chapter II. The simulation requirements are intended for and are applicable to TEGAS design simulations. The currently available methods of test vector generation are explored in Chapter III.

CHAPTER I

VLSI TEST METHODS AND TECHNIQUES

Approach

The VLSI test methods and techniques can best be examined by:

1. determining the types of faults common to CMOS LSI,
2. postulating appropriate fault models,
3. identifying effective test methods,
4. examining known test techniques, and
5. developing testability design goals.

CMOS LSI Faults

Frequently, initial LSI circuit faults are the result of an imperfect manufacturing process. Prototype devices may simultaneously contain a number and variety of physical defects. The failure mechanisms which afflict MOS LSI logic circuits are varied. Galiay (1980) identified the principle failure mechanisms, shown in Table 1, of an LSI microprocessor. The failure sites were randomly distributed within the device. The insignificant failures were due to very large imperfections (e.g., a scratch across the entire chip's surface) and could be detected by almost any method.

TABLE 1

PRINCIPLE FAILURE MECHANISMS
(DERIVED FROM GALIAY 1980)

Failure Mechanism	Observed Frequency
Short Between Metallizations	39%
Open Metallizations	14%
Short Between diffusions	14%
Open Diffusion	6%
Short Between Metallization and Substrate	2%
Inobservable	10%
Insignificant	15%

The faults associated with each failure mechanism can be reduced to two primary types. See Reddy (1983), Beh (1982), and Galiay (1980). The first type consists of a circuit open which, if tested statically, produces a constant voltage at the circuit output. Typically, the open run permits leakage currents to drive the associated floating transistor gates to the power rail voltage potentials. The second fault type consists of shorts which can introduce an asynchronous sequential loop, modify the realized function, or introduce analog behavior. Typically, the short occurs between adjacent metallizations. Frequently, the short causes the resulting circuit to behave as a wired-OR.

Fault Models

VLSI fault effects can be divided into two areas, parametric and logical. A logical fault causes the logic function of a circuit to be altered to some other. One prevalent type of logical fault causes the circuit signal to be fixed at a constant value. If the signal is fixed at logical one, the circuit is said to be stuck-at-one. Conversely, if the signal is fixed at logical zero, the circuit is stuck-at-zero. Another common logical fault causes two signals to behave as a wired-OR function. Parametric fault effects typically alter the magnitude of a circuit parameter causing a change in circuit speed, current, or voltage. A frequently encountered parametric fault is excessive gate delay.

Test Methods

Testing methods can be divided into three types. DC testing, also known as static or functional testing, consists of applying test input vectors to the device under test and analyzing the corresponding steady state outputs to determine correct functional behavior. AC testing, also known as dynamic or parametric testing, verifies the time-related behavior of the circuit elements and/or the magnitudes of voltage and current levels. See Barzilai (1983). Clock rate testing is similar to DC testing but occurs at clock frequencies near the device maximum. Clock rate testing is performed to reduce device test time and to prevent data loss in time dependent circuits such as MOS memories.

The DC and AC test methods should both be employed. The combination of the two techniques will detect stuck-at-faults and excessive gate delay conditions. The AC test vectors should be alternated to induce gate input transitions at clock rates. The alternation of vectors increases the probability of detecting excessive gate delays. See Barzilai (1983).

Test Techniques

A taxonomical representation of all test techniques is shown in Figure 1. The VLSI test techniques of interest to the design group require specialized hardware for implementation. These techniques can be divided into built-in, on-line and built-in, off-line approaches. These approaches are discussed in the works of Muehldorf (1981), Williams (1982), and Buehler (1982). The relative advantages and disadvantages of each method are summarized.

The five principal off-line test techniques, applicable to VLSI devices, are :

1. self-oscillation,
2. self-comparison,
3. partition,
4. scan, and
5. built-in logic block observer/highly integrated logic device observer.

Examples of the built-in, off-line techniques are included in Buehler (1982). See Appendix B. The example of a new off-line test technique, HILDO, is included in Appendix C.

```
                         DIGITAL
                     TEST TECHNIQUES
                            :
                            :
                  :---------------------------:
                  :                           :
            BUILT-IN TEST                 EXTERNAL
                  :                         TEST
                  :                           :
             :-----------------:              :
             :                 :              :
          ON-LINE          OFF-LINE           :
       (CONCURRENT)    (NON-CONCURRENT)       :
             :                 :              :
             :                 :              :
     :-------------:           :              :
     :             :           :              :
INFORMATION    HARDWARE        :              :
 REDUNDANT     REDUNDANT       :              :
 (CODING)    (REPLICATION)     :              :
                               :              :
                               :              :
                     :--------------:         :
                     :              :         :
                 RESIDENT       RESIDENT      :
                 SOFTWARE       HARDWARE      :
                                              :
                                              :
                                       :------------:
                                       :            :
                                    SOFTWARE    AUTOMATIC
                                   DIAGNOSTICS    TEST
                                                EQUIPMENT
```

Figure 1. Digital Test Techniques (derived from Bueheler 1982)

The built-in on-line test techniques, applicable to VLSI devices, will be briefly discussed due to their limited applications within custom VLSI devices. The principal techniques are :

1. duplicate hardware with comparators and

2. totally self-checking circuits.

Built-in, Off-Line Test Techniques

Self-Oscillation. The self-oscillation test technique consists of establishing feedback paths through the device under test. Typically, the elements of the feedback path will oscillate at a frequency determined by the gate delays and stray circuit capacitance. If the elements in the path oscillate, then the logic in the feedback path is presumed to be free of stuck-at-level faults.

The self-oscillation method requires a sensitized path to be established in the device under test. The technique of Susskind (1973) may be adapted for this procedure. Oscillation testing, although largely unexplored in theoretical work, offers a relatively simple method of testing high-speed logic such as gallium arsenide ICs (Long 1980).

In the worst case, an N input combinatorial logic block with M outputs will require an M input multiplexer and an N output decoder. In addition, an exclusive-OR function may be required to establish the proper feedback polarity. At least six I/O pins must be provided for external control of the multiplexer, decoder and exclusive-OR functions.

Self-Comparison. The self-comparison technique partitions the device under test into a number of functionally identical modules during test. The modules may be constructed by circuit reconfiguration or the designer may identify functionally similar modules within the undisturbed circuit. Optionally, the modules may be considered functionally similar for a limited set of test vectors.

Test vectors are simultaneously applied to the inputs of the functionally identical modules. The outputs of the modules are compared by a self-checking checker circuit. See Anderson (1973) and Breuer (1976). If the checker results are identical, then the circuit is considered functional.

Each identical output requires a dedicated, totally self-checking checker. The additional checker hardware is approximately six gates and two I/O pins per pair of signal inputs. The ad hoc reconfiguration logic, if any, must also be added to the device.

Partition. The partitioning technique divides the device under test into a number of functionally independent modules during test. The independent modules are constructed by reconfiguring the device under test to eliminate connections between modules. The connections are terminated in new outputs and inputs. This method reduces the number of test vectors which must be applied and also permits the test vectors to be applied simultaneously. For example, if a circuit with seven inputs is to be exhaustively tested, then at most

$$2^{7} = 128 \text{ unique test vectors} \tag{1}$$

must be applied. If the circuit can be partitioned into a three input and a four input set of modules, then the number of unique test vectors is reduced to

$$2^{4} = 16 \text{ test vectors} \tag{2}$$

or less. See McClusky (1980).

The partition technique may require a number of additional I/O pins. The number of I/O pins is equal to twice the number of connections which must be severed. Up to six gates of additional hardware must be included for each severed connection.

Scan. The scan test technique has a number of variations (Williams 1982). The variations include Level Sensitive Scan Design (LSSD), Scan/Set Logic, Scan Path, and Random Access Scan. The LSSD and Scan Path methods convert all the memory storage elements of a device to a single shift register during the test cycle. Test vectors are sequentially applied by shifting the vector into the appropriate memory elements, returning the device to the normal operation configuration, operating the device for a number of cycles, and then returning to the test mode and retrieving the new memory element data. The new memory element data is compared to a previously determined, correct response. See Eichelberger (1978).

The Scan Path and LSSD techniques are identical except for the hardware implementation of the memory storage elements. The Random Access Scan technique employs a direct addressing scheme which permits any latch element to be randomly selected then sampled or set. The Scan/Set technique does not use the existing registers but adds additional test registers. The additional registers are used as supplemental test points.

The additional I/O requirements are four pins for the LSSD, Scan Path, and Scan/Set methods. The I/O requirements can be reduced by combining test and normal functions. The additional test hardware requirements are estimated to be four to twenty percent of the device hardware (Williams 1982). The I/O requirements of the Random Access Scan technique can be reduced to as few as six pins if two onboard scan address decoders are included.

Built-In Block Observer/Highly Integrated Logic Device Observer. The Built-in Logic Block Observer, BILBO, is an on-chip signature analyzer. The BILBO develops a pseudo-random test vector, applies it to the device under test, and compresses the response into a characteristic signature. The random test vectors can be generated by a linear feedback shift register (LFSR). Alternately, precalculated vectors can be stored in a dedicated, test ROM. Another LFSR is used to compress and store the circuit response. The LFSRs are initialized with a fixed seed prior to running the test. The compressed response is then compared to the expected response. See Bhavsar (1981), Konemann (1980), and Frohwerk (1977).

The Highly Integrated Logic Device Observer, HILDO, is similar in operation to the BILBO. However, the HILDO uses a single LFSR to generate the test vectors and store the compressed response. The device's circuitry is used as the feedback path within the shift register. The HILDO method uses less hardware than the BILBO but the test vector patterns are less predictable and require additional simulation effort to confirm the adequacy of testing (Beucler 1984).

The BILBO technique requires four additional I/O pins per device, an input data latch, and an output data latch. Existing latches may be modified by adding approximately six gates per latch element. The HILDO hardware requirements are similar except that only a single data latch is required.

Advantages/Disadvantages of Built-in, Off-line Test Techniques. The relative merits of each off-line test technique were compared in the test implementations of Appendices B and C. Each test technique was implemented on a device containing a half adder and a full adder circuit. The merits of each technique are summarized in Table 2.

Built-in, On-line Test Techniques

The built-in, on-line test techniques can be further divided into information redundant and hardware redundant approaches. Both approaches are capable of detecting and correcting errors due to intermittent failures. In addition, most errors are detected almost immediately

TABLE 2

RELATIVE MERITS OF BUILT-IN, OFF-LINE TEST TECHNIQUES
DERIVED FROM BUEHELER (1982)

Factor	Adder and Registers	Self Oscillation	Self Comparison	Partition	Scan Path	BILBO/ HILDO
Test Generator	External	Local	External	External	External	Local
Test Analyzer	External	External	Local	External	External	Local
Gates*	63	2	12	18	36	49
Transistors	188	3	48	52	149	153
Fault Coverage(%)	100	80***	100	100	100	100**
Total Number of faults	52	54	80	76	48	48
Total Number of Tests	5	4	7	5	5	15

TABLE 2-CONTINUED

Notes

* For the test techniques, the numbers indicate the number of gates and transistors needed to implement the test circuitry.

** Testing is excessive, 15 tests are executed when only five are needed for 100 percent fault coverage of the adder.

*** As explained in Appendix B, only 80 percent fault coverage is possible in the oscillation mode, but 100 percent fault coverage is possible if the add mode is also used.

after occurrence and before substantial data corruption has occurred.

The information redundant test techniques typically employ parity checks and Hamming codes to detect and correct errors. The information redundant techniques employ a fixed number of code bits which must be maintained with the data. Frequently, specialized hardware is employed to generate and check the code bits. See Breuer (1976).

The hardware redundant test techniques consist of duplicate hardware with comparators and totally self-checking circuits. If the hardware is doubled and a comparator verifies that the outputs agree, then the system can only detect a failure. Additional hardware is required to correct a faulted output. A triplicate voting system contains tripled hardware and a comparator which selects the majority output and ignores the dissenting output. Breuer (1976) has shown that triplicate voting systems have a limited operating time in which the reliability is greater than that of a simplex system.

The comparator used in the redundant hardware techniques is vulnerable to failure. Thus, the question of who checks the checker must be addressed. Anderson (1973) proved that a class of totally self-checking circuits exist. The circuits are characterized by multiple outputs. Valid outputs are represented by M high level outputs out of N outputs. The comparator is referred to as an M out of N totally self-checking circuit.

Testability Design Goals

The adequacy of a test program should be compared to a test coverage design goal. Often, the ideal test goal is considered to be 100% test coverage. The ideal goal is difficult to achieve in practice. Work by Agrawal (1982b) suggests that 100% test coverage is usually excessive.

The design goal should be to minimize total cost. The minimum cost occurs when the cost to develop additional test vectors approximates the total repair cost for the defective units which are not detected during the test. The principal test development cost is dominated by the test vector's engineering development time. If the defect is detected before final test, then

$$\mathrm{Cdrtv} \cong \mathrm{Trc} = \mathrm{Npu}\ R(F)\ \mathrm{Cr} \qquad (3)$$

or

$$R(F) \cong \frac{\mathrm{Cdrt}}{\mathrm{Npu\ Cr}} \qquad (4)$$

where

Cdrtv is the cost, in dollars, to develop the remaining test vectors,

Trc is the total repair cost in dollars,

Npu is number of production units,

Cr is the repair cost per unit in dollars, and

R(F) is the field reject rate.

The desired test coverage is dependent on the desired field reject rate and the process yield. The desired test coverage can now be calculated by the method of Agrawal (1982b). This method is briefly outlined below.

Test Model

The device may contain a number of physical defects. The number of physical defects is assumed to be proportional to the area of the device. Further, the average size of the physical defects is large compared to logic structures on the device. Thus, each defect will induce a number of logical faults and the number of faults per device will have a Poisson distribution. Thus, the chip yield can be approximated by

$$Y = (1 + L*Do*A)^{-(1/L)} \qquad (5)$$

where

$Do^2 L$ is the variance of Do,

Do is the physical defect density in defects per square mil,

A is the device area in square mils, and

Y is the device yield in percent.

Probability of Accepting a Faulty Device

A device is assumed to possess n individual stuck-at or other faults out of a maximum possible N faults. The fault probability is

$$P(n) = (1-Y)\,\frac{(No - 1)^{n-1}}{(n-1)!}\,e^{-(No - 1)} \qquad (6)$$

for n = 1, 2, 3, ... and

$$P(0) = Y \qquad (7)$$

where

P(n) is the probability of n logical faults and

No is the average number of faults on a defective device.

The average number of faults, No, can be estimated from a previous process run or can be experimentally determined during the initial process run. The experimental derivation procedure is given below.

The hypergeometric probability of accepting the device during testing is

$$Qo(n) = \frac{\binom{N-n}{m}}{\binom{N}{m}} \tag{8}$$

where

Qo(n) is the probability of accepting a device with n logical faults.

For large values of N, equation (8) can be approximated by the binomial probability

$$Qo(n) = (1 - n/N)^{m} \cong (1 - m/N)^{n} = (1 - F)^{n} \tag{9}$$

where

F=m/N is the ratio of faults tested, m, to the maximum possible number of faults, N.

The probability of testing and accepting a defective device is

$$Ybg(F) = \sum_{n=1}^{N} Qo(n)\, P(n) \tag{10}$$

$$\cong (1 - F)\,(1 - Y)\, e^{-(No-1)F} \tag{11}$$

where

Ybg(F) is the probability of a accepting a faulty device tested to fault coverage F.

Experimental Determination of the Average Number of Faults

The average number of faults can be experimentally determined from a small number of devices and a limited set of test vectors. Typically, 100 to 200 devices and 20 to 30 test vectors are required for VLSI devices. The fault coverage of each test vector need not be great but the fault coverage must be known.

The devices are set up for test and each test vector is sequentially applied. The test is halted at the first test which the device fails or when the vectors are exhausted. The vector at which the test halts is recorded. From this information, a graph of failure probability as a function of fault coverage is constructed.

The average number of faults can be determined from the graph by two methods. The first method is by graphical comparison with theoretically derived curves. The theoretical probability of device failure is

$$P(F) = 1 - Y - Ybg(F) \qquad (12)$$

where

$P(F)$ is the probability of failure of a device when tested to fault coverage F.

Substituting equation (11) yields

$$P(F) = (1 - Y)\,[1 - (1 - F)e^{-(No - 1)F}]. \qquad (13)$$

A failure probability curve can be generated by evaluating equation (13) over the range of F. A family of curves is constructed by evaluating equation (13) for several different values of No. The family of curves is superimposed on the experimentally derived graphs and the best curve fit is selected. The corresponding value of No is selected as the average number of faults on a defective device. See Figure 2.

The second method of extracting the average number of faults is to estimate the derivative of the failure rate. The derivative can be obtained by differentiating equation (13) with respect to F. Thus,

$$P'(F) = \frac{dP(F)}{dF} \qquad (14)$$

$$= (1 - Y)\,[\,1 + (1 - F)\,(No - 1)]\,e^{-(No - 1)F} \qquad (15)$$

and

$$P'(0) = (1 - Y)\,No\ . \qquad (16)$$

P'(0) is the slope of the probability curve near the origin and can easily be approximated. Thus, No can be determined from the derivative. If the yield is unknown, equation (16) can be approximated as

$$No \cong P'(0)\ . \qquad (17)$$

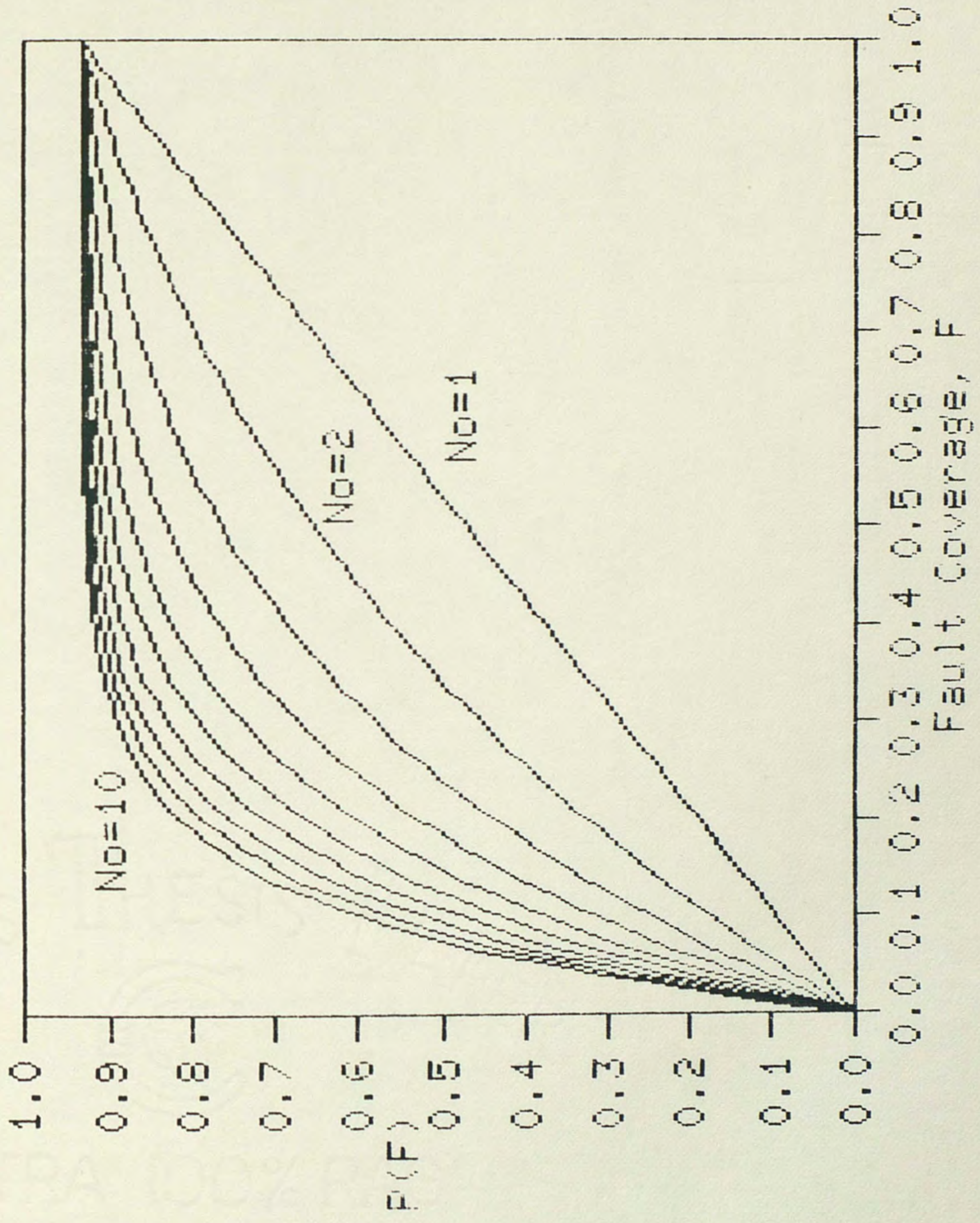


Figure 2. Failure Probability as a Function of Fault Coverage and the Average Number of Failures

Field Reject Rate and Desired Fault Coverage

The field reject rate is the ratio of defective devices which are accepted to the total number of devices which pass the acceptance test. Thus,

$$R(F) = \frac{Ybg(F)}{(Y + Ybg(F))} \tag{18}$$

where

R(F) is the field reject ratio.

Substitution of equation (11) yields

$$R(F) = \frac{(1 - F)(1 - Y)\, e^{-(No -1) F}}{Y + (1 - F)(1 - Y)\, e^{-(No -1) F}} , \tag{19}$$

the field reject rate. The desired fault coverage can be obtained by rewriting equation (19) as

$$Y = \frac{(1 - F)(1 - R)\, e^{-(No -1) F}}{R + (1 - F)(1 - R)\, e^{-(No -1) F}} . \tag{20}$$

Equation (20) is plotted for two different field reject rates and two values of No. The results are shown in Figure 3 and Figure 4.

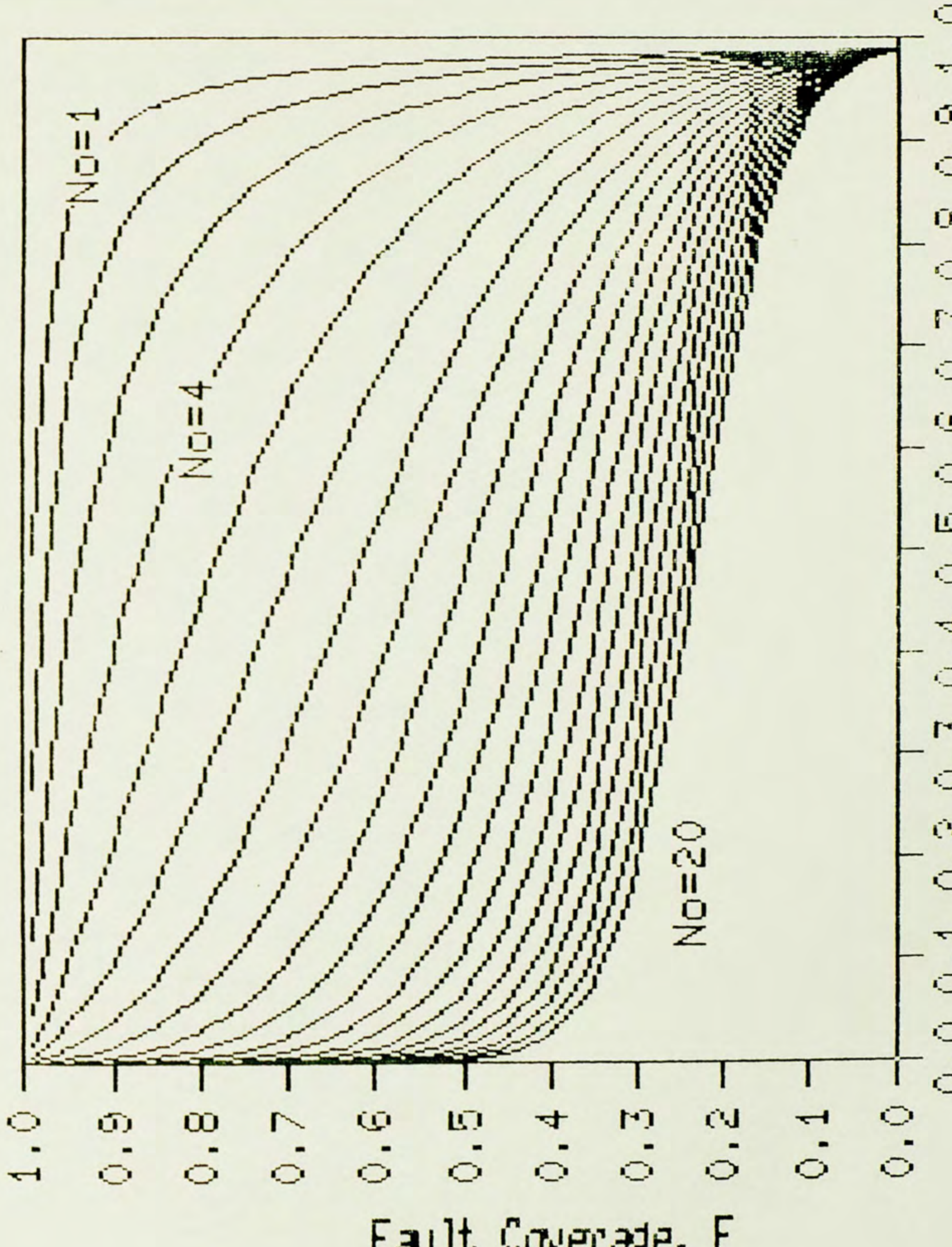


Figure 3. Fault Coverage Required for a Field Reject Rate of R=0.01

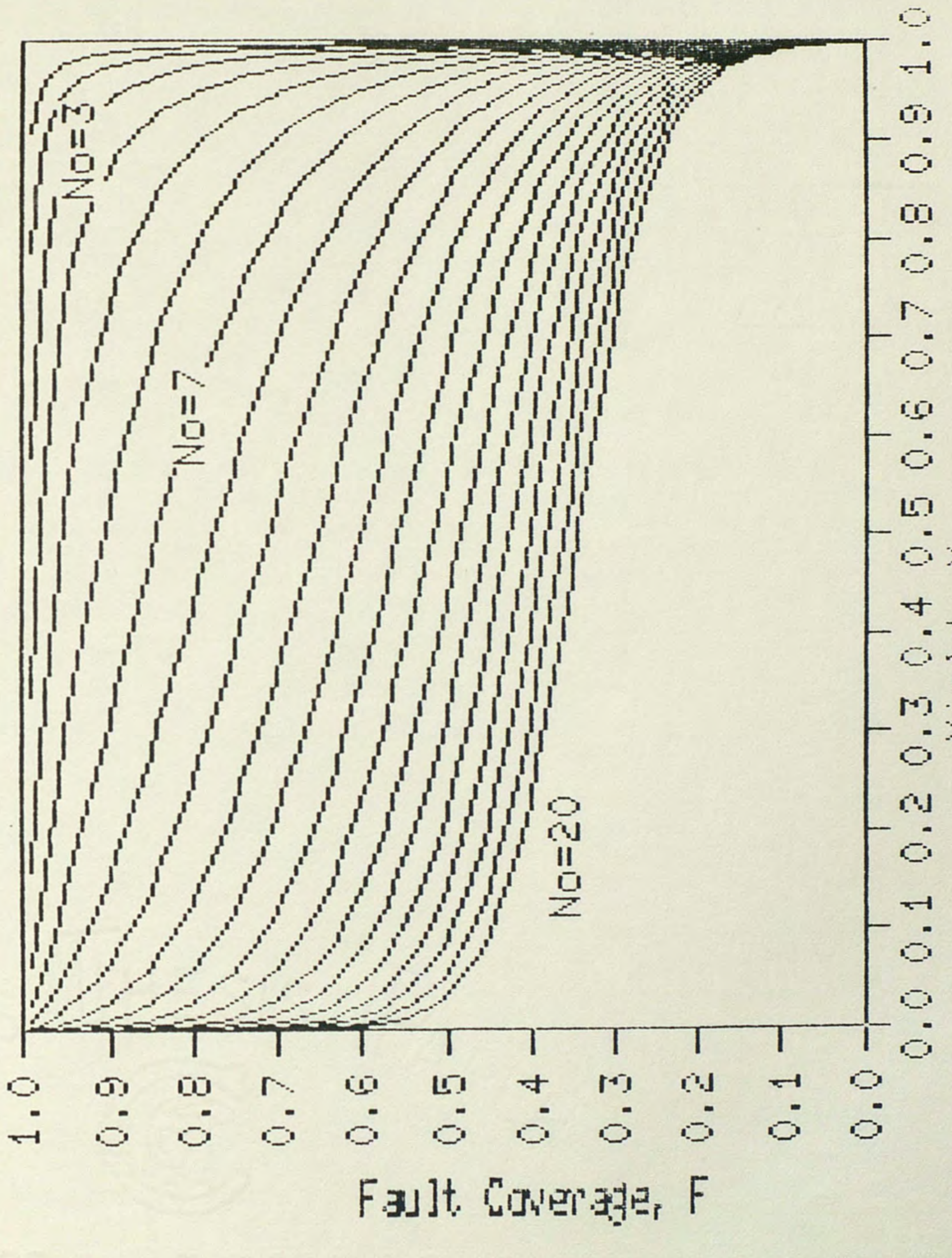


Figure 4. Fault Coverage Required for a Field Reject Rate of R=0.001

Chapter II

DESIGN GUIDELINES

Approach

The design of a testable VLSI device can best be executed by:

1. observing a set of design guidelines throughout the design phase,
2. measuring the relative testability of a design simulation, and
3. estimating the test generation effort which must be expended to meet the fault coverage goals of Chapter I.

If the relative testability of the design is too low or the test generation effort is excessive then the following remedial actions should be taken:

1. locate regions of logic which are difficult to test,

2. apply remedial design rules to improve device testability, and

3. repeat the cycle until the desired relative testability figure or the test generation effort goal is achieved.

The relative testability figure of a device is frequently determined by a testability measurement utility. The measurement program is an optional utility in several VLSI logic simulation software packages.

Design Rules

Quality is designed, not tested into products. Thus, the design of testable VLSI devices can be simplified by observing a number of empirical design rules during the block hardware and detailed design phases. The design rules can be applied at several design levels. The empirically derived design rules and their intended applicability are shown in Table 3. The design rules can be divided into three major groups; placement of scan path test point flip-flops, general design practices, and clock distribution related rules. The placement of test point flip-flops, TPFFs, and related restrictions may be relaxed for pipeline delay registers and in other cases indicated by experience. Where applicable, possible

TABLE 3

DESIGN RULES
(DERIVED FROM GENERAL ELECTRIC 1981)

Rule No.	Description	Level of Application			
		Circuit	Macro	Device	Board
Test Access Rules					
1*	Test Point Flip-Flop (TPFF)	X	X	X	X
R1*	Relaxation of Rule 1		X	X	X
R1A*	Topographical Loops		X	X	X
R1B*	Maximum Sequential Depth		X	X	X
2*	Adding TPFFs		X	X	X
2A*	Difficult to Test Networks		X	X	
2B*	Maximum Logic Block Size		X	X	X
2C*	Primary Inputs and Outputs		X	X	X
3*	Organization of TPFFs			X	
4*	Accessing TPFFs			X	
5*	IC Interface			X	
6*	Circuits Requiring Special Access		X	X	
General Design Rules					
1	Asynchronous Loops	X	X	X	X
2	Critical Hazards and Races	X	X	X	X
3	Latch Design	X			
4	One-Shots		X	X	
5	Clock Generators			X	
6	Error-Correcting Logic		X	X	
7	Unused Gate Inputs	X	X	X	X
8	Unspecified ROM fields			X	
9	Bus Conflict in Test Mode			X	
10	Dynamic Storage	X			
11	Tri-State Bus Receivers	X			

Note
* This rule is applicable to Scan methods only.

TABLE 3--CONTINUED

Rule No.	Description	Level of Application Circuit	Macro	Device	Board
Clock Distribution					
1	Critical Hazards and Races	X	X	X	X
2	Flip-Flop Type	X	X		
3	One Clock Input		X	X	
4	Clock Skew		X	X	X
5	Clock Inputs Free of Hazards		X	X	X
6	Static Inputs to Latches		X	X	X
7	Independent Clock and Data		X	X	X
8	ANDing of Clocks		X	X	X

TABLE 3--CONTINUED

Rule No.	Rule
Test Access Rules	
1*	Test Point Flip-Flop (TPFF) - Each latch and flip-flop that is not part of a memory array shall be part of a test point flip-flop (TPFF) string.
R1A*	Relaxation of Rule 1 - Topographical Loops A few internal flip-flops can be omitted but not under the following conditions: TPFFs must be retained when it is required to observe a topographical loop.
R1B*	Relaxation of Rule 1 - Maximum Sequential Depth A few internal flip-flops can be omitted but not under the following conditions: A maximum sequential depth of three layers may exist in any of the block of logic not containing TPFFs.
2A*	Adding TPFFs - Difficult to Test Networks TPFFs should be added (beyond those required by Rule 1): When a network has been identified as difficult to test by a testability measurement utility.
2B*	Adding TPFFs - Maximum Logic Block Size TPFFs should be added (beyond those required by Rule 1): When the block size surrounded by TPFFs is larger than 1000 gates. Maximum block size shall be 200 gates for control logic and unique random logic between macros.
2C*	Adding TPFFs - Primary Inputs and Outputs All primary device input and outputs shall have a TPFF on the them. Exceptions are clocks and real time control signals.
3*	Organization of TPFFs - TPFFs shall be organized in a single serial string except where chip organization makes parallel test access more attractive.

TABLE 3--CONTINUED

Rule No.	Rule
Test Access Rules - Continued	
4*	Accessing TPFFs - The test access scheme of the network must have the property of simultaneous inclusion of TPFFs in a chain and mutual exclusion of chains in a group.
5*	IC Interface The device shall have a test mode and a test interface with the following capabilities: 1. ability to turn on the test mode, 2. interface should be independent of mission processing when in the test mode, 3. means of injecting signals into the TPFFs, and 4. means of retrieving test response data from the TPFFs.
6*	Circuits Requiring Special Access - Direct test access shall be provided to all RAMs and ROMs. PLAs and the digital I/O of analog functions shall have TPFFs on their inputs and outputs.
General Design Rules	
1	Asynchronous Loops Asynchronous loops shall exist only within latches.
2	Critical Hazards and Races No critical hazard or race shall exist. (Same as Clock Distribution Rule 1)
3	Latch Design All non-array memory elements shall be binary latches that are free of critical races and hazards on clock transition and indeterminate states.

TABLE 3--CONTINUED

Rule No.	Rule
General Design Rules - Continued	
4	One-Shots One shot multivibrators shall not be used unless externally controllable degating logic capable of disabling the one shot and substituting a controllable level is incorporated.
5	Clock Generators Clock generators shall not be used unless externally controllable degating logic capable of disabling the clock and substituting a controllable level is incorporated.
6	Error-Correcting Logic - When error-correcting logic is used, test access shall be provided to the uncorrected data.
7	Unused Gate Inputs All unused gate inputs shall be tied to a constant logic state.
8	Unspecified ROM fields Unspecified ROM fields shall produce a consistent logic state.
9	Bus Conflict in Test Mode - The test mode control shall not cause two (CMOS) outputs tied together to be active and complementary at the same time.
10	Dynamic Storage The dynamic state of a node shall not be used for storage.
11	Tri-State Bus Receivers Tri-state buses shall be designed so that each receiver will have a logical output value when all drivers are disabled (to avoid potential oscillation problems).

TABLE 3--CONTINUED

Rule No.	Rule
Clock Distribution Rules	
1	Critical Hazards and Races - No critical hazards and races shall exist.
2	Flip-Flop Type - Flip-flops shall be two-phase level sensitive flip-flops. The second choice is single phase level sensitive flip-flops. Edge triggered flip-flops shall not be used.
3	One Clock Input - The clock input to a latch must depend on one and only one clock primary input.
4	Clock Skew - The inactive interval of the external primary clock shall include the worst case clock skew.
5	Clock Inputs Free of Hazards - All clock inputs shall be free of pulses due to critical hazards. In conjunction with Rule 3, this can be insured by the following equivalent rule. The primary clock signal feeding a latch shall not reconverge with odd parity.
6	Static Inputs to Latches - All data inputs of a latch shall have reached their final value when the active phase of the clock begins.
7	Independent Clock and Data - The clock and data input of a latch shall not depend on a common global clock.
8	ANDing of Clocks - Two combinational functions may be ANDed with each other and then ANDed with the clock. They shall not be ANDed with the clock and then ANDed with each other.

Note
 * This rule is applicable to Scan methods only.

alternate implementations are suggested to the designer, e.g., the use of degating logic for clocks and one shot circuits. Many of the general design rules were developed from good practice rules of thumb.

Other less obvious design rules have been included to avoid subtle design problems. For example, the error-correcting rule prevents failed logic from reducing overall device reliability. The bus conflict rule was mentioned to avoid test vector induced bus conflicts. A better approach is to decode a bus enable word into individual bus transmit enables, thus assuring that only one bus enable is active. The clock distribution rules were included to insure good design practices. The design rules should also serve as a checklist during design reviews.

Block Hardware Design Phase

A design test technique can be selected after system test techniques have been selected and general functional definition, e.g., placement of ALUs, registers, counters, etc., has occurred. The designer should then:

1. have the general, functional block definition of the device,

2. consider the relevant factors and select a device test technique,

3. implement the test technique within the block definition as suggested by the rules of the technique,

4. observe the design rules during the block design,

5. capture the design in a high-level register transfer language (RTL), and

6. verify the algorithm of the test technique.

General Test Technique Selection

The selection of a system compatible, test technique should be based on a number of relevant factors. Among these are:

1. the device architecture,

2. the desired fault coverage,

3. compatibility with the test strategy of the next higher assembly level,

4. the available test generators,

5. the available test analyzers, and

6. the remaining device capacity.

A close examination of the relevant items will prevent the designer from re-inventing a number of pitfalls. Often, the selection is not obvious due to the large number of factors and available test techniques. The additional hardware required to implement the most desirable test techniques should be identified and the hardware impact estimated to insure the required test hardware does not exceed the available device capacity. The designer should also determine the availability and capabilities of the device test facilities. Insufficient capacity may force the test strategy to be altered, e.g., selection of an oscillation testing technique over a scan technique. Similarly, particularly low field reject rate goals which require unachievable fault coverage figures of some test methods may alter the selection. Frequently, a combination of techniques are employed, e.g., a 32 bit adder is split into two 16 bit adders for a scan test cycle. The information obtained during the examination can be used in conjunction with Table 2 to determine the best possible test technique or combination of techniques for the device.

Concurrent Testing

The algorithm embodied by the hardware should be verified. At the block design stage, a number of high-level programming languages can adequately represent the design. A better approach is to employ a high-level register transfer/computer hardware description language with low-level constructs and fault simulation capabilities. The high-level portions of the device's algorithm can be expanded during the block hardware design process. The expansion process reduces transition effort and assures the device's architecture is retained during the transition.

The device's model must include specific test hardware algorithms to permit test modeling during the block hardware design phase. This is particularly true for the HILDO technique. Early algorithm tests can determine the optimum connections of the HILDO register. The algorithm testing will also substantiate the choice of test strategy.

Register Transfer/Computer Hardware Description Languages. Levendel (1982) notes the division of computer hardware description languages into procedural and non-procedural types. In a non-procedural language, the order of the statements in a function is not critical. All statements within the function are effectively executed in parallel and specified delays are associated with the transfer. In a procedural language, all statements of a function are executed sequentially as in a conventional programming language.

Levendel (1982) noted the fault processing capabilities of several existing languages and proposed several methods of simulating faults within computer hardware description languages, CHDL. Levendel also proposed a method of extending test pattern generation to both procedural and non-procedural languages.

AHPL was an early procedural language which did not incorporate fault handling procedures (Hill 1978). The non-procedural language ISPS provides for data, control, and operational fault simulation (Barbacci 1981). Control faults are simulated in the conditional section of behavior definition statements. In addition, the user can insert operational faults to simulate data failures within arithmetic, logic, and other functional units. Both hard and transient fault models may be inserted at runtime. Presently, ISPS does not incorporate an ATG utility.

NmPc is a set of functional design and simulation tools intended for VLSI design. The modeling utility is based on the non-procedural language ISP (Rose 1984). NmPc can perform low-level emulation using constructs. NmPc can also perform stuck at fault simulation for state registers. Both control and data failures can be simulated. The faults may be inserted and removed at runtime. NmpC can also simulate state interactions, e.g., shorted conductors. Both logical OR and AND state interactions can be simulated. In addition, NmPc can estimate fault coverage by determining the lines of ISP code which are executed during simulated hardware activity. At present, Nmpc does not incorporate an ATG utility.

Detailed Hardware Design Phase

Several functions must be performed in the detailed design phase. Among the functions noted by Breuer (1976) are the test related functions of fault coverage measurement and evaluation of test point alternatives. See Table 4. Test point effectiveness can be viewed as a portion of the general problem of obtaining adequate fault coverage. The major functions of the detailed hardware

TABLE 4

APPLICATIONS OF LOGIC SIMULATION
(DERIVED FROM BREUER 1976)

1. Hardware Design Verification
    a. Verify Logical Correctness
    b. Timing Analysis
        Delay Models
        Race and Hazard Analysis
    c. Initialization Analysis
2. Fault Analysis
    a. Fault Coverage
    b. Timing Analysis Under Fault Conditions
    c. Initialization Under Fault Conditions
    d. Fault Induced Races and Hazards
    e. Evaluation of Test Point Effectiveness
    f. Evaluation of Self Checking Circuitry
    g. Evaluation of Fail-Safe Circuitry
    h. Evaluation of Roll Back* Hardware-Software
3. Software Development
    a. Debugging Software to Run on Hardware Not Yet Implemented
    b. Development of Diagnostic Software Programs and Microcode for Computer Systems

Notes

* The detection of failure causes the system to roll back to the last fault free state and retry the operation.

design phase are:

1. conversion of RTL to the gate level logic design language (LDL),

2. verification that the language crossover was accurately implemented,

3. performing a timing analysis,

4. adherence to the applicable design rules, and

5. determining the fault coverage of the device.

The design must now cross over to a specific logic design language. At present, an RTL to LDL crossover utility does not exist and the designer must perform this step manually. The RTL simulation results can assure design fidelity in the LDL representation. That is, the block design phase algorithm test can be repeated and the results compared. The designer should continue to observe the detailed design rules of Table 3. The timing analysis should be performed when the possibility of hazards and races exists.

Concurrent Testing

The concurrent testing of the detailed hardware design phase must verify the adequacy of the test strategy and calculate the fault coverage of the design. Adequate fault coverage will permit the design to achieve the field reject rate goal. Two methods of testing, which can be used during the detailed design phase, are software logic simulation systems and SSI/MSI/LSI breadboards. Breadboards enjoyed wider acceptance before the development of faster, more versatile logic simulators.

Breadboard. Breadboards employ easily alterable wiring and readily available SSI/MSI/LSI devices to simulate a VLSI design. Time scaling is often employed to permit signals to be distributed over the larger breadboard assembly. Frequently, the breadboard also serves as a prototype in higher assembly levels.

Two breadboards may be preferred. Although the multiple copies must be updated when revisions are made, the effort is not exceptional. Multiple breadboards provide additional user access and permit a side by side hardware verification. Hardware verification can discriminate between design failures and random noise or test berth problems, e.g., bad conductors and ICs (Kidder 1981).

When the design is implemented on a breadboard, a guided probe signature analyzer can verify the controllability of test points. Similarly, manual fault insertion can be used to roughly estimate the fault coverage of the design. Accurate estimates are difficult to obtain due to the inaccessible internal IC node nets.

Logic Simulation Systems. Logic simulation systems are a collection of software utilities. The device design is initially captured and then processed through the software utilities. The general process flow was outlined by Everett (1984) and is shown in Figure 5.

Testability Measurement Utilities. A testability measurement utility can assess the relative testability of a device. The utility infers the difficulty of testing a device through a simple algorithm. The algorithm assigns a testability figure of merit to each device node independently of test vector selection. The SCOAP and COPTR test measurement utilities, cited by Hess (1982) and Kirkland (1984) respectively, employ virtually identical algorithms. The COPTR utility, which is applicable to TEGAS, will be explored.

COPTR uses controllability and observability figures to generate a testability merit figure. The controllability figure measures the relative difficulty of

```
:----------------:     :---------------:
:     System     :<----:    Product    :
:     Design     :     :  Definition   :
:----------------:     :---------------:
        :
        v
:----------------:     :---------------:
:     Logic      :<--->: Testability   :
:     Design     :     :   Analysis    :
:----------------:     :---------------:
        :
        v
:----------------:     :---------------:     :------------:
: Logic-Decision :<--->: Test Pattern  :<--->:   Fault    :
:  Verification  :     :  Generation   :     : Simulation :
:  (Functional   :     :---------------:     :------------:
:   Simulation)  :             :
:----------------:             :
        :                      :
        v                      v
:----------------:     :---------------:
:   Logic Timing :     :    Test       :
:   Verification :     : Pattern Post-:
:----------------:     :  Processing   :
        :              :---------------:
        v                      :
:----------------:             v
:    Physical    :     :---------------:
:     Layout     :     :    Test       :
:----------------:     :   Program     :
        :              : Development   :
        v              :---------------:
:----------------:             :
:  Manufacturing :             :
:----------------:             :
        :                      :
        v                      :
:----------------:             :
:     Test       :<-------------
:----------------:
```

Figure 5. Logic Simulation System Process
(derived from Everett 1984)

setting a particular device node to a desired level. Four figures are computed for each node:

1. combinational 0 controllability,

2. combinational 1 controllability,

3. sequential 0 controllability, and

4. sequential 1 controllability.

The controllability figure represents the quantity of nodes that must be set to control the desired node. Sequential controllability is a number of states into which the device must be set before the desired level appears at the node. The complete analysis must compute both levels for the sequential and combinational modes.

The observability figure represents the quantity of nodes which must be set to permit the observed node's level to propagate to a device output. Observability has both combinational and sequential values, but does not examine levels.

The controllability figures are computed before the observability figures. Nodes which are completely uncontrollable or unobservable are distinguished by assigned merit figures of -1. The node testability figure, also referred to as the test observability figure, is derived from the quantity of additional nodes which must be controlled, their predictability, and the observability of the test node. Hess (1984) defines a node's predictability as a measure of the degree to which internal nodes can be controlled to a known state by primary inputs. Predictability can be considered synonymous with initialization of sequential circuits.

The results of a testability measurement analysis are typically printed in tabular form. The controllability and observability figures of merit are included for each node. Average node controllability, observability, and other summary statistics, are included. Typically, the cost of running a complete testability measurement analysis on a previously captured device design is small. Hnatek (1984) estimated the cost of using a measurement utility to examine a 1000 node circuit in engineering hours and CPU minutes.

Interpretation of Testability Merit Figures. The testability merit figures produced by the testability measurement utilities are not strongly related to the potential fault coverage of the device. Agrawal (1982a) concluded that the SCOAP algorithm provides some correlation between the testability analysis figure of a class of nodes and the difficulty of generating test vectors for the class. However, Agrawal also concluded that the correlation of the testability analysis figure of a specific node and the probability of generating a test vector for the node is much weaker. Specifically, he also suggested that efforts to improve the testability figure of singular nodes with large testability figures are pointless. In one test case, approximately 70% fault coverage of all nodes with high testability figures was achieved after a small number of test vectors was applied. Thus, 70% of such an effort could be wasted.

The probability of detecting a fault is related to the testability figure of the node. Agrawal (1982a) suggested a method of estimating the test generation effort for the device. Assume

$$P(Ti) = e^{-aTi} \qquad (21)$$

where

P(Ti) is the probability of detecting a fault at a node with a testability figure of Ti and

a is a model characterization parameter.

Untestable nodes must substitute a very large number for the commonly used merit figure of -1. If the design has N possible, distinct faults with respective testability measures of T1, T2, T3, ... Ti ... TN and if the fault detection performance of each test vector is statistically independent, then

$$P(i) = [1 - (1 - P(Ti))]^{V} \tag{22}$$

where

P(i) is the probability of detecting the ith fault after V vectors and

V is the number of vectors applied.

The total fault coverage provided by the set of vectors is

$$f(V) = \frac{1}{N} \sum_{i=1}^{N} [1 - (1 - P(Ti))]^{V} \quad \text{or} \tag{23}$$

$$f(V) = 1 - \frac{1}{N} \sum_{i=1}^{N} [1 - P(Ti)]^{V} \tag{24}$$

where

f(V) is the fault coverage provided by V vectors.

Substituting equation (21) into (24) yields

$$f(V) = \frac{1}{N} \sum_{i=1}^{N} [1 - (1 - e^{-aTiV})] \quad . \tag{25}$$

The model characterization constant, a, can be obtained by the following procedure. Assume the ith fault is detected on the Vith vector, then

$$P(Vi) = (1 - P(Ti))^{Vi-1} \; P(Ti) \tag{26}$$

where

P(Vi) is the probability of the ith vector detecting the fault with testability measure Ti.

The joint or maximum probability of detecting the fault in this order is

$$L = \sum_{i=1}^{N} [\, 1 - P(Ti)]^{Vi-1} \; P(Ti) \qquad (27)$$

where

L is the likelihood function.

The maximum probability can be determined by taking the natural logarithm of equation (27),

$$\ln(L) = -a \sum_{i=1}^{N} Ti + \sum_{i=1}^{N} (Vi - 1) \ln [\, 1 - e^{-aTi} \,], \qquad (28)$$

and differentiating to obtain

$$\frac{1}{L} \frac{\partial L}{\partial a} = -\sum_{i=1}^{N} Ti + \sum_{i=1}^{N} \frac{(Vi - 1)\, e^{-aTi}}{1 - e^{-aTi}} = 0 \;, \qquad (29)$$

$$\text{or} \quad \sum_{i=1}^{N} Ti \frac{[\, 1 - (Vi - 1)\, e^{-aTi} \,]}{1 - e^{-aTi}} = 0 \;. \qquad (30)$$

Equation (30) can be used to calculate a. The fault simulation utility output should associate the testability figure of a node with the number of the first vector which detects the fault. Equation (30) is then iteratively solved for the characterization constant. This method will yield the minimum variance unbiased estimator of a as V approaches infinity (Freund 1971).

Logic Fault Simulation Utilities. An alternate method of assessing the testability of a device is by manual fault generation and logic fault simulation. A set of test vectors is manually prepared and the fault coverage of the set is evaluated with a logic fault simulator utility. The associated cost for a 1000 node device is reflected in estimates by Hnatek (1984). The estimate included manual test development times in hundreds of hours and computer fault simulation costs in tens of CPU hours.

Most major logic simulation systems possess a standard fault simulation utility. Breuer (1976) recognized three methods of fault simulation; parallel, concurrent, and deductive (or fault list propagation).

The parallel simulation of many faults can increase the simulation speed. In an example cited by Breuer (1976), an AND gate has two inputs A and B. Each input is represented as the ith bit of a respective register. The output is equivalent to the ith bit of the AND product of the A and B registers. If the width of the registers is W bits, then a single operation can simultaneously perform W simulations of the gate. Thus, as many as W fault simulations may be processed in parallel.

Concurrent simulation is performed by assigning each faulted gate a particular set of input and output values in a super fault list (Breuer 1976). Each fault is assigned a unique position within the super fault list and the list is sorted by fault index. If the fault alters a gate output, the inputs associated with the node are examined and the fault propagates through the fault lists of succeeding gates. Concurrent simulation is event directed, that is, only the portion of the circuit affected by the fault is processed.

Deductive fault simulation processes all faults simultaneously. A storage area reduction is achieved by storing only the faults associated with each gate (Breuer 1976). Generally, the number of faults associated with a specific gate is small. A list of faults which affect each gate is developed by fault list propagation. Logic values which do not agree with the fault free device values are propagated to succeeding gate lists until blocked at gates with controlling values on one or more remaining inputs. The lists of gates with controlling values are examined to insure that propagated fault effects do not alter the controlling values. Only the fault logic values which propagate to a device output are considered detectable (Muehldorf 1981).

Fault Reduction Utilities. The burden of fault simulation can be reduced by preprocessing the design with a fault reduction utility. Fault reduction or, more commonly, fault collapsing utilities combine distinct logic faults into equivalent sets. Breuer (1976) noted that an N input gate has 2(N+1) possible, distinct faults. Consider an N input AND gate which has N possible stuck at 0 input faults. Each input fault is equivalent to the output stuck at 0 fault. Thus, the N stuck at 0 input faults need not be considered. Similarly, a gate output stuck at 1 fault will be detected by any test for a gate input stuck at 1 fault. The gate output stuck at 1 fault is said to dominate the gate input stuck at 1 faults and need not be considered. Thus, the 2(N+1) distinct faults within the gate can be replaced by an equivalent set of N+1 distinct faults. Breuer (1976) also suggests that fault collapsing can substantially reduce the number of simulated faults. Everett (1984) implies fault collapsing can yield fault reductions of up to 30 percent.

Toggle Test Utilities. Toggle test utilities are a computationally inexpensive means of determining circuit controllability and assessing test vector performance. Typically, the user supplies a list of faults associated with the device nodes and a set of test vectors as utility inputs. A fault is considered detectable if the associated node can be driven to the opposite logic value of a stuck at fault.

Toggle tests evaluate the controllability of faulted nodes. The results do not indicate the relative difficulty of controlling a specific node or observing the node response. However, toggle tests can be used to inexpensively estimate test vector performance and determine if fault simulation is warranted. Toggle tests can also determine the regions of the device which are covered by the test vector set.

Timing Analysis Verifier. The internal timing delays of a device are normally proportional. However, process variations and capacitive loading of MOS gates can induce gate timing skew. A logic verifier utility can predict the circuit timing based on user defined circuits and inputs. Functional specifications may be used for repeated blocks. The timing verifier will assess the worst case minimum/maximum delay times for both the rising and falling edges of a node. A single period test is then iterated until the test produces predictable data. That is, the output signal at every gate is consistent with the inputs and the timing definitions.

Timing verifiers are excellent for the examining the synchronous devices of interest. The timing verifier results will detect possible hazard and race conditions. In addition, the timing verifier will detect setup and hold time violations. All aspects of device activity in the clock cycle can be viewed in a relative fashion (Rappaport 1983).

Conclusions. Testability merit figures and fault coverage can be calculated more accurately by logic simulations than by breadboard methods. In addition, the cost of running a testability measurement utility is far less than the cost of manually generating test vectors and performing fault simulations. Better testability measurement utilities, faster fault coverage algorithms, and specialized hardware accelerators are being developed. Thus, logic simulation systems can be expected to improve. In contrast, rough breadboard estimates of fault coverage are unsuitable for the required accuracy of field reject rate calculations. Furthermore, breadboard methods have little promise of future improvement and are becoming increasingly complex as device density increases.

Available Logic Simulation Systems

Although most logic simulation systems operate on general purpose computers, several types are designed to execute on specialized hardware accelerators. Dramatic simulation speed improvements in logic simulation utilities have recently occurred as a result. Another recent improvement is physical modeling. Physical modeling systems characterize higher function VLSI hardware by employing adapter modules containing test components. The adapter modules permit test component interaction with device simulations. Hardware accelerators, physical modelers, and most of the capabilities noted in Table 4 have been incorporated in most major logic simulation systems. Current logic simulation system capabilities have been compiled by Werner (1984b and 1984c) and Everett (1984). Their results are condensed in Table 5. The faults simulation utilities available on major systems have also been compiled. The results are listed in Table 6.

The utilities of particular interest to Tegas system users include two testability measurement packages. The SCOAP utility will be incorporated as an adjunct to TEGAS 6 (Hnatek 1984). The COPTR/ATG utility is presently available with TEGAS products (Kirkland 1983). Kirkland

TABLE 5

LOGIC SIMULATION SYSTEM CAPABILITIES
(DERIVED FROM WERNER 1984c AND EVERETT 1984)

System Manufacturer	Hardware Accelerator Available	Physical Modeler Available	Testability Measurement Utility	Fault Simulator Utility	Automatic Test Vector Generation
Calma/GE	No	No	COPTR*	Texsim Tegas$	p/o COPTR
Daisy	Yes on MegaLogician	PMX	SCOAP	Yes	No
Mentor	Xsim**	Midas	No	Fsim	No
Metheus	No	No	No	Gen Rad Hilo-2	p/o Hilo-2
Silvar-Lisco	No	No	No	Bifas p/o Bimos	No
Valid	Realfast	Realchip***	No	Teradyne LASAR	p/o Hilo-2

Notes
$ includes an automatic fault collapsing utility
* SCOAP will be added to Tegas 6.
** Derived from Zycad.
*** Will link to Realfast.

TABLE 6

LOGIC FAULT SIMULATION UTILITIES
(DERIVED FROM WERNER 1984a)

System Manufacturer	Fault Simulation Utility	Type of Fault Simulation	Estimated Speed# (events/sec)	Memory Required (bytes/ gates)	Timing Verifier
Calma/GE	Tegas-5	Parallel	2000	10K/1000	Yes
	Texsim	-	2000	10K/1000	No
Daisy	Daisy Logic Simulator	Yes*	Logician 1000/ MegaLogician 100,000	40K/1000	Yes**
Mentor	None	-	1000	2M/10K	Yes
Metheus	***	Parallel	1000	50K/1000	Yes**
Silvar-Lisco	Bimos	Parallel$	450	300K/1000	No
Valid	SCALD	Concurrent	1200	60K/1000	Yes
Zycad	LE1001/ LE1002	Serial	500,000/ 1,000,000	not applicable	No

Notes
Logic simulation speed
$ Optional
* Not yet available
** Part of logic simulation utility.
*** Hardware description language

(1983) also noted additional synergistic effects between the COPTR testability analysis utility and an automatic test vector generation utility operating on previously captured design information. The ATG utility will not attempt to generate test vectors for untestable nodes. This additional property of COPTR/ATG utility makes it the preferred testability measurement program for Tegas systems. The forthcoming addition of the SCOAP testability measurement utility to Tegas will enlarge the selection of available testability measurement utilities.

Verification of Testability Design Goals

The device must achieve the testability design goals to realize the field reject rate goals. The testability can be verified by a number of procedures. An efficient approach which Agrawal's (1982a) and Everett's (1984) work suggests is:

1. Apply a testability measurement utility.
2. Alter completely untestable nodes if possible. Some nodes, e.g., those which employ pull up and pull down lines, will always be untestable.
3. If a region of circuitry is highly untestable then:
   3A. verify that the design guidelines have been observed,

- 3B. consider additional design modifications to improve testability, and
- 3C. rerun the testability measurement utility on the modified design and repeat steps 3B and 3C until satisfactory results are obtained.

4. Run a fault collapsing utility, if available.
5. Estimate the test vector generation effort by:
   - 5A. developing a trial number of ATG vectors,
   - 5B. simulating the vectors in a logic fault simulation utility and characterizing the design, and
   - 5C. estimating the remaining test vector generation effort. Agrawal's (1982a) method can be employed.
6. Use the test vector generation estimate to determine whether the design's testability figure should be increased or additional test vectors generated.

The choice between additional test vector generation and increasing the testability figure of the design should favor the lowest cost solution which achieves the desired field reject rate.

Chapter III

TEST VECTOR GENERATION

Approach

Three methods of obtaining and manually generating test vectors will be examined. Eight methods of automatically generating test vectors will also be examined. The capabilities and applications of an available automatic test vector generation logic utility will be discussed. The final steps of test data augmentation and conversion will be considered.

Methods of Obtaining Test Vectors

Muehldorf (1981) and Middleton (1983) identified the following methods of obtaining and generating test vectors:

1. manual methods,

2. purchase,

3. conversion,
4. path sensitization,
5. D algorithm,
6. derivatives of functions,
7. algebraic expression,
8. heuristic algorithms,
9. learn mode,
10. random patterns, and
11. signature analysis.

The first three methods of the list are manual methods. Manual methods are distinguished by the absence of available algorithms. The eight remaining automatic methods can be expressed as algorithms. Several have been incorporated in automatic test vector generation utilities.

Manual Test Vector Generation

The manual methods of test vector generation share the common problems of highly variable risk and methods of implementation. The methods of manual test vector generation include manual mode, purchase and conversion. Each method is examined below.

Manual Mode. The designer prepares test information in a high level description language such as stimulus, tester, or native assembly language. Stimulus language inputs are prepared for fault simulation utilities. Native assembly language, e.g., microprocessor assembly code, can be loaded into RAM/ROM and be executed for immediate testing or hardware learn modes. Tester language can be directly loaded to the ATE tester. The information is converted to tester object language by special purpose compilers. The methods of conversion are shown in Figure 6.

Purchase. If the device has been previously designed or assembled by another manufacturer, device test vector sets may exist. Frequently, the lowest risk and most cost effective solution is to purchase the test vector set. The buyer must carefully determine the fault coverage afforded by the purchased test vector set. Potential vendors include the original device vendor, consultants, and ATE manufacturers.

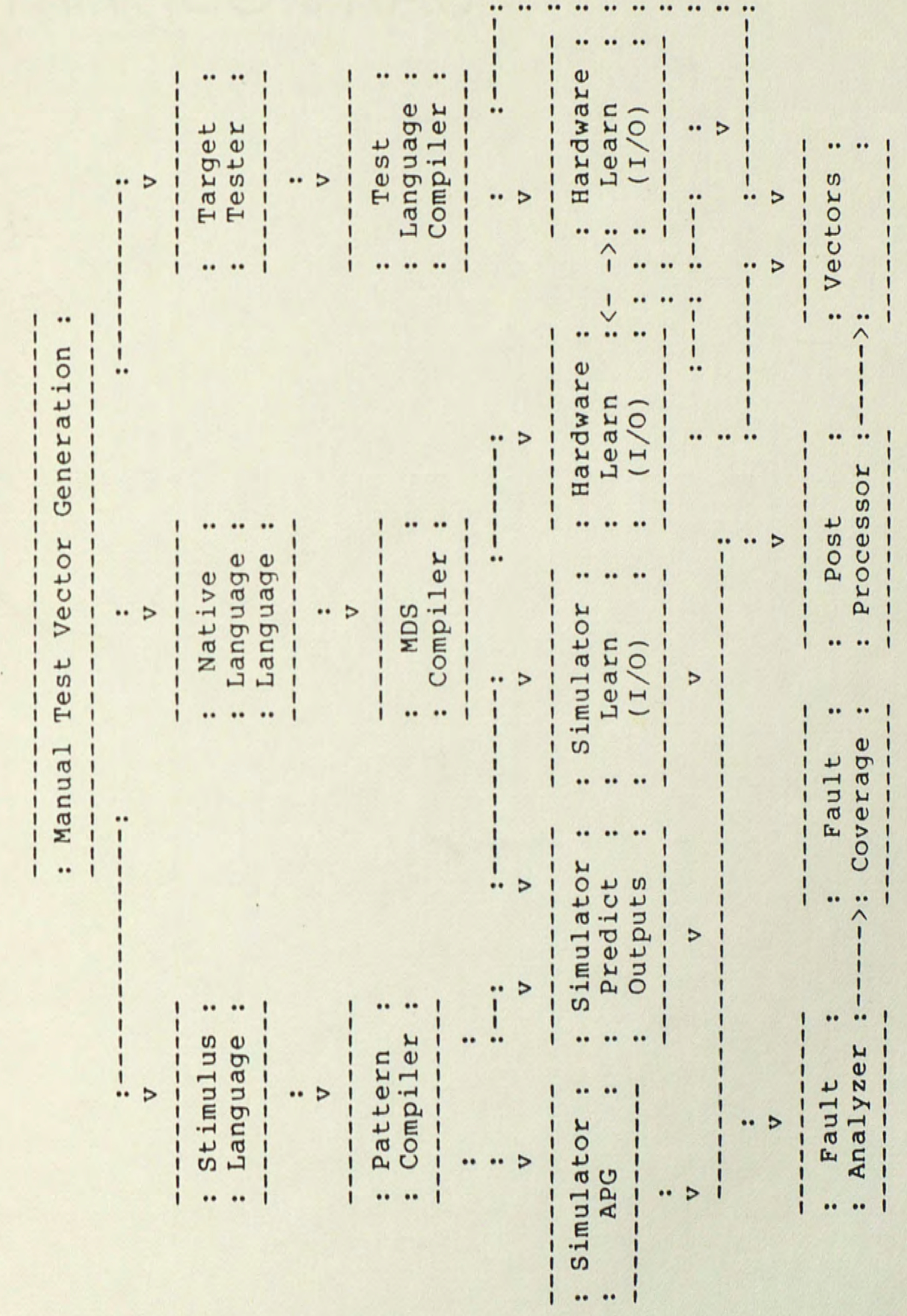


Figure 6. Manual Method of Test Vector Generation
(derived from Middleton 1983)

<u>Conversion</u>. Frequently, test vector sets are obtained from another test unit. A conversion between the tester languages is often required. Although the conversion process may be automated, the wide variations in test methods often necessitate manual postprocessing steps. Several ATE vendors provide translators which semi-automatically translate other tester languages to the target tester language (Middleton 1983).

Automatic Test Vector Generation

The automatic test vector generation methods can be implemented in an algorithmic fashion. The majority of available utilities implement combinations of algorithms. Often, the utilities employ heuristic and iterative algorithms. Thus, the results are highly dependent on the particluar algorithm implementation.

<u>Path Sensitization</u>. The path sensitization algorithm is implemented by:

1. provoking the faulted gate input to be tested with a logic value inverse to the modeled stuck at fault,

2. creating a forward drive path by which the gate output signal can be propagated to a device output, and

3. verifying that the output is only a function of the signal change resulting from the stuck at fault.

The path sensitization procedure has two primary disadvantages. The search for a forward drive path resembles a tree search. The associated path propagation algorithms make arbitrary choices at the branch nodes. When a conflict is encountered, the algorithm must backtrace and remake decisions at the branch nodes until a suitable solution is obtained or the search tree is exhausted. The second disadvantage is that alternate test vectors may not be detected. That is, test vectors for faults with reconvergent paths may not be detected because the tree search algorithms are following single rather than multiple paths (Breuer 1976).

D Algorithm. The D algorithm overcomes the problems associated with the path sensitization method. The D algorithm employs a multivalued logic calculus which behaves in accordance with the rules of Boolean algebra for a Boolean variable (Roth 1980). The term D denotes a signal which has a high value at a fault free node and a low value at a faulted node. Similarly, the logical inverse of D denotes a signal with a low value at a fault free node and a high value at a faulted node. See Figure 7.

Multiple paths can be searched by propagating the effects of the fault through the device in a parallel manner. However, only one fault at a time can be propagated. Furthermore, an untestable fault initiates a large, fruitless computational effort which ceases only when all search possibilities have been exhausted.

Derivatives of Functions. The derivatives of Boolean functions can be used to generate test vectors. Susskind (1973) noted the following partial derivative procedure. Assume a Boolean expression of the form,

$$z = Z(X1, X2, X3, \ldots, Xi, \ldots Xn) \qquad (31)$$

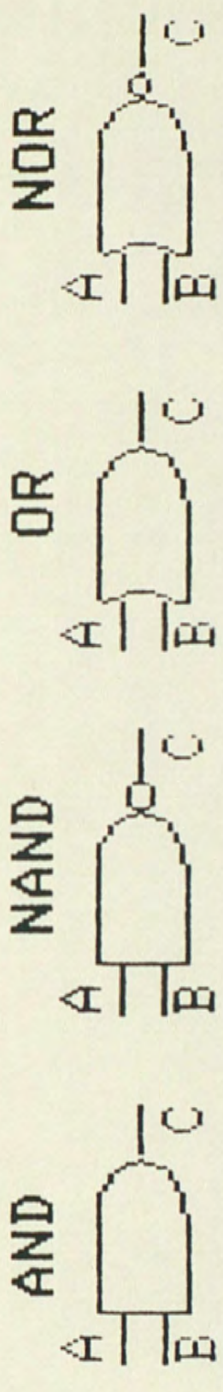
where

Z is an n input, single output Boolean function,

z is the output of the Boolean function, and

Xi is the ith input of the Boolean function.

The EXCLUSIVE OR operation, also referred to as the ring sum or the Boolean difference of z with respect to x, is expressed as



	AND			NAND			OR			NOR		
	A	B	C	A	B	C	A	B	C	A	B	C
PROP. CUBE	D	1	D	D	1	$\bar{D}$	D	0	D	D	0	$\bar{D}$
	1	D	D	1	D	$\bar{D}$	0	D	D	0	D	$\bar{D}$
	$\bar{D}$	1	$\bar{D}$	$\bar{D}$	1	D	D	0	D	$\bar{D}$	0	$\bar{D}$
	1	$\bar{D}$	$\bar{D}$	1	$\bar{D}$	D	0	$\bar{D}$	$\bar{D}$	0	$\bar{D}$	D
TEST CUBE	1	1	D	1	1	D	0	0	D	0	0	D

Figure 7. Elements of the D Algorithm

$$\frac{dz}{dXi} = Z(X1,X2,\ldots,0,\ldots Xn) \oplus Z(X1,X2,\ldots,1,\ldots Xn) \quad (32)$$

A sensitive path from the ith input to the output exists if

$$1 = Z(X1,X2 \ldots,0, \ldots Xn) \oplus Z(X1,X2 \ldots ,1, \ldots Xn). \quad (33)$$

Thus, a sensitive path from Xi to z can be constructed if the other input variables can be selected in accordance with Equation (33).

The formal procedure can be stated as follows. Given a fault on lead i which is provoked if and only if the Boolean function P is true, then the test for the fault can be found by determining the solutions to

$$Pi \frac{dz}{dXi} = 1 \quad (34)$$

where

Pi is the ith input of the Boolean function P.

This method is not restricted to stuck at fault models. Consider the circuit of Figure 8 with the function,

$$z = a\,\overline{b} + \overline{b}\,\overline{c}\,\overline{d} \quad , \quad (35)$$

then the tests which sensitize input a must satisfy

$$\frac{dz}{da} = (\overline{b} + \overline{b}\,\overline{c}\,\overline{d}) \oplus (\overline{b}\,\overline{c}\,\overline{d}) = 1 \quad \text{and} \quad (36)$$

$$= \overline{b}\,c + \overline{b}\,d = 1. \quad (37)$$

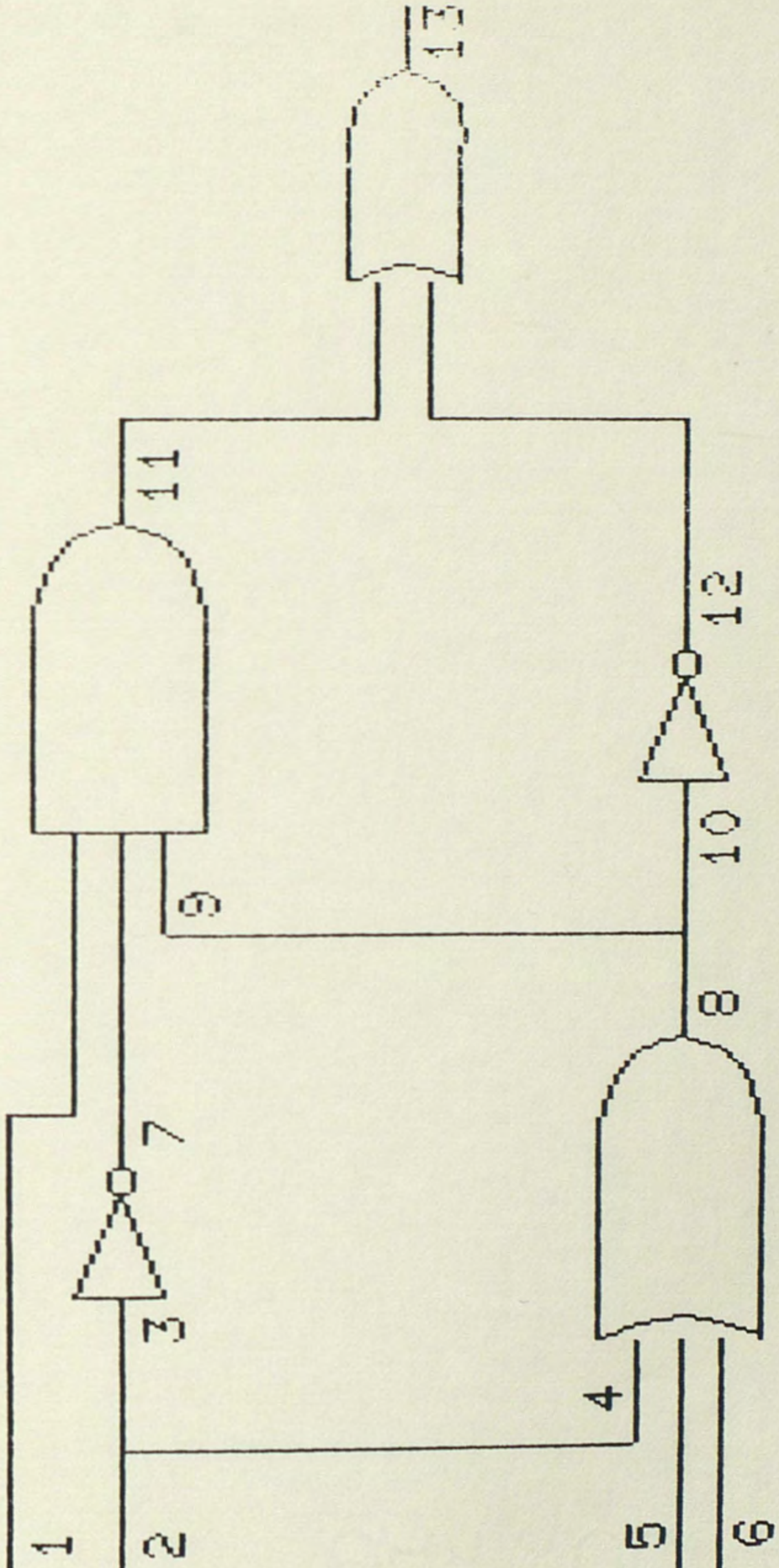


Figure 8. Example Circuit

The path sensitizing test vectors which satisfy Equation (37) are

$$Z(a,0,1,X) = Z(a,b,c,d) \quad (38)$$

and

$$Z(a,0,X,1) = Z(a,b,c,d) \quad (39)$$

where

X is the don't care state.

The derivative method suffers from two problems. First, the algorithm must manipulate formidable Boolean equations. The second problem is the inability to handle multiple faults.

Algebraic Expression. The algebraic expression technique employs subscripted variables which retain the circuit topographical information. In this method, also referred to as Poage's method, the Boolean variables must be subscripted according to the device nets in which they appear. Susskind (1973) introduced the term SPOOF, Structure Parity Observing Output Function, for the expressions. A sensitized path can be extracted by the following procedure.

Consider the circuit of Figure 8. The lead 9, standard Boolean expression is

$$z = b + c + d \tag{40}$$

where

z is the circuit response at lead 9 and

b, c, d are the input variables.

The lead 9 SPOOF expression is

$$S_{9} = b_{2,4,8,9} + c_{5,8,9} + d_{6,8,9} \tag{41}$$

where

S_{9} is the SPOOF response at lead 9 and

$b_{2,4,8,9}$ is the SPOOF variable with a path list of 2 - 4 - 8 - 9.

Similarly, the lead 12 SPOOF expression is

$$S_{12} = \overline{b}_{\overline{2},\overline{4},\overline{8},\overline{10},12} * \overline{c}_{\overline{5},\overline{8},\overline{10},12} * \overline{d}_{\overline{6},\overline{8},\overline{10},12} \quad . \tag{42}$$

The output SPOOF expression is

$$S_{13} = S_{11,13} + S_{12,13} \qquad \text{or} \tag{43}$$

$$\begin{aligned} S_{13} = {} & a_{1,11,13}\,\overline{b}_{\overline{2},\overline{3},7,11,13}\,b_{2,4,8,9,11,13} + \\ & a_{1,11,13}\,\overline{b}_{\overline{2},\overline{3},7,11,13}\,c_{5,8,9,11,13} + \\ & a_{1,11,13}\,\overline{b}_{\overline{2},\overline{3},7,11,13}\,d_{6,8,9,11,13} + \\ & \overline{b}_{\overline{2},\overline{4},\overline{8},\overline{10},12,13}\,\overline{c}_{\overline{5},\overline{8},\overline{10},12,13}\,\overline{d}_{\overline{6},\overline{8},\overline{10},12,13} \quad . \end{aligned} \tag{44}$$

Identical terms which contain different path lists are considered distinct. The first product term of Equation (44) contains a term and its inverse. For the purpose of determining test vectors, the equation is not reduced to lowest form. Clearly, if lead 1 has a stuck at 0 fault then the terms which contain a 1 in the path list become zero. The terms which remain are:

$$S_{13}^{1sa0} = \bar{b}_{\bar{2},\bar{4},\bar{8},\overline{10},12,13}\ \bar{c}_{\bar{5},\bar{8},\overline{10},12,13}\ \bar{d}_{\bar{6},\bar{8},\overline{10},12,13} \quad . \qquad (45)$$

where

S_{13}^{1sa0} is the lead 13 response with lead 1 stuck at 0.

The algebraic expression method can process multiple faults. The test method can also be extended to develop fault dictionaries and shorted run tests (Flomenhoft 1973). The principal disadvantage of the method is the large computational effort associated with manipulating the Boolean expressions (Chang 1974).

Signature Analysis. Signature analysis is implemented by applying a pseudo-random, repeatable set of test vectors to an initialized DUT. Frequently, the test vectors are generated by LFSRs (Frohwerk 1977). During the test sequence, the response of the device is compressed by a second LFSR. The final result is compared to a known correct response. Alternately, the number of level transitions on each output can be determined. Susskind

(1981) has shown the transition count is the lowest order Walsh coefficient of the DUT's response. The acceptability of the device is normally inferred from the single Walsh coefficient.

Susskind (1981) also proposed an improved method of inferring the DUT's condition. Two of the Walsh coefficients of the DUT's response are evaluated and compared to the coefficients of a correct model. The test will detect any pin fault, e.g., an input or output stuck at level condition. The correct device response can be determined through simulation or, more commonly, by direct comparison with a correctly functioning device. A fault dictionary, which relates the device's signature to known defects, can be compiled. The dictionary can serve as a fault location guide.

Learn Mode. The learn mode consists of two methods. The first method compares a device under test to a known, correctly functioning device, e.g., hardware learn. The hardware learn method is particularly important if the device's responses have not been previously calculated. In the second method, the DUT is compared to a software model, e.g., software learn.

Two variations of hardware learn exist. The first variation assumes the device's response is unknown. User prepared test vectors must be applied to an initialized device. The device must be initialized to a repeatable, predetermined state. The test vectors are applied in sequential fashion and a list of correct responses is assembled. The device is reinitialized after each new response is obtained and the test is repeated until the next new response is encountered.

The second variation of hardware learn is primarily intended for microprocessors. It requires a native assembly language description of the device. The device's object program is stored in a RAM/ROM adapter within the ATE. The device is then operated and the tester observes the device operation, e.g., read/writes to the RAM/ROM. The observations are used to construct a test program written in the ATE's native language. Several ATE vendors market utilities which perform this variation of the hardware learn mode.

The software learn mode is characterized by the use of a previously developed set of test vectors and anticipated device responses. The test vectors are applied and the device responses are compared to the anticipated device responses. Often, the device responses are the results of a fault simulation utility. One variation of the software learn mode is of particular importance to microprocessors. The device's assembly language program is compiled using the microprocessor development system. The resulting binary object program is simulated in a corresponding memory model and is interconnected with a device model. State patterns from the model are recognized by an ATE program and converted by a postprocessing program. The postprocessing output consists of ATE tester native language commands.

Heuristic Algorithms. Heuristic algorithms employ high level languages to automate the production of test vectors. Often, the production rules are deduced from a functional model or description of the device. High level procedural languages, expansions, e.g., Macro-T, Pascal-T, etc. are frequently employed (Middleton 1983). Recent efforts have been directed at extensions to CHDL (Son 1982). The high level constructs of CHDL are tabulated and guide the automatic generation of behavioral test vectors.

Random Patterns. Random patterns are generated and applied to a model of the DUT until simulation results indicate sufficient fault coverage has been achieved. The random patterns are usually generated by a seeded random number generator. The cost associated with the method is low. However, the achieved fault coverage is highly variable unless Agrawal's (1982b) method is employed. Many ATG utilities employ variations of the random pattern generation algorithm.

Kirkland (1984) noted additional synergistic effects between a testability measurement utility and a random pattern generation utility operating on a previously captured design. The ATG portion of the utility does not attempt to generate vectors for untestable nodes. Agrawal (1982b) noted that the first few test vectors detect a great number of faults. Thus, it is possible to use random test vectors to obtain a relatively high degree of fault coverage.

Sequential Circuits

Sequential networks are distinguished by the inclusion of memory elements. A pattern of test vectors is required for adequate testing. There are three principal methods of generating test vector patterns for sequential circuits (Susskind 1973). The first method is to verify the functional characteristics of the device. The second method is to translate the given sequential network into a related iterative combinational network. The last method is to verify the sequential network's state table.

Available Test Generation Utilities

Many logic simulation systems incorporate test generation utilities. The utilities available with the major systems are shown in Table 5. The test generation capabilities of the available Tegas utility are representative and will be examined.

The Tegas test generator has two modes. The path sensitization test mode proceeds through three phases which successively attempt to develop test vectors. During the first phase, the test generator proceeds by leveling the device circuitry. That is, each element within the device is assigned a number starting at one at the input pin and ending with n at the output pin. Feedback path information is also gathered during this process. The test generator will then attempt to generate vectors which detect stuck at faults on the device outputs. Other node stuck at faults which are detected will also be recorded.

During the second phase of the path sensitization mode, the test generator attempts to detect only the remaining undetected faults. The test generator proceeds by attempting to drive nodes to the desired value. No effort is made to propagate the gate output to an observable point. The third phase is similar to the second except that the test generator attempts to propagate the signal to a device output.

The second mode of the Tegas test generator employs heuristic generators which implement four strategies. The four strategies are: random, start, check, and DQUB. The second test mode also employs a path sensitization technique.

The random pattern generator develops a user specified number of test vectors. The start strategy initially fills the test vector with don't care states. In each consecutive pattern, the most significant don't care is replaced by a low logic level until all inputs are low levels. The process is continued by replacing the low logic levels with high logic levels and then replacing the high levels with low levels.

The check test generator produces checkerboard vector sets. The first vector of the checkerboard vector set is constructed by shifting a high logic level into the most significant bit of a zero vector. The next vector has a low level followed by a high level shifted into the most significant bits. The shifting is repeated for succeeding vectors until the last vector is filled with alternating low and high levels. The process is then repeated with high and low input levels exchanged.

The last method, DQUB, is similar to the start method except that the vectors can be modified to include more than one change per vector. This method does not utilize don't care states.

Suggested Applications

The Tegas test vector generation utilities can be applied in number of schemes. The path sensitization mode of the test vector generator can be applied to small devices. However, the large computational effort associated with the mode may prohibits its application to large designs.

The random test generation mode and Agrawal's (1982b) method can effectively be employed for larger devices. Manually generated test vectors can probably be used in conjunction with this method. The faults which are detected by the manually generated patterns must be eliminated from the faults considered by Agrawal's method.

The estimated costs of using the path sensitization mode must be compared to the costs of randomly generating the test vectors. The cost of the random generation effort can be estimated by characterizing the device with only a few hundred vectors. Subsequent fault simulation costs must also be considered for each method.

Test Data Augmentation and Conversion

The test vectors obtained are processed through a final Test Data Augmentation and Conversion, TDAC, step (Muehldorf 1981). The step is composed of two tasks. The first task consists of converting the logic tests into tester native languages. The second task consists of augmenting the logic tests with parametric tests which check/verify the voltage/current properties of the DUT I/O pins. Typically, the parametric tests are performed by applying a constant voltage/current to a pin and observing the resulting current/voltage. The value obtained is compared to previously defined limits. International Microelectronic Products, a gate array vendor, performs only DC parametric testing on devices. The tests reveal power bus shorts and improperly functioning I/O pins but do not detect logic faults. The vendor achieves surprisingly good results (Rappaport 1984).

Chapter IV

CONCLUSIONS

CMOS LSI faults can be divided into two types, logical and parametric. The three test methods which exist are DC, AC (or parametric), and clock rate testing. Five viable test techniques, which were discussed, exhibit a great diversity of characteristics. Thus, the designer must carefully consider each design's requirements and test technique characteristics before making a selection.

The testability design goal need not be 100 percent fault coverage. Agrawal's method can be employed to determine the most cost effective tradeoff between field reject rate and device fault coverage. Frequently, the required fault coverage is significantly lower, e.g., 85 to 95 percent.

Design rules can help the designer avoid common mistakes. The design rules can be applied during the block and detail hardware design phases. Test technique selection, which occurs in the block design phase, should be based on five, design dependent criteria. The criteria are: the device architecture, the desired fault coverage, compatibility with the test strategy of the next higher assembly level, available test generators/analyzers, and the remaining device capacity.

Logic simulation systems are better than breadboards. The gap between these simulation methods will continue to widen with the development of improved software and hardware for logic simulation systems. Presently, a number of logic simulation systems are available. Most systems contain a testability measurement utility. The testability measurement utilities should be employed and the results processed as inputs to Agrawal's method. A limited number of test vectors, e.g., 100 to 300, should be generated by a random method. Agrawal's method can then be used to estimate the remaining test vector generation effort. Remedial design efforts can be implemented to improve the device testability if the test generation effort is excessive. These steps form the basis of a procedure which verifies the testability design goal can be met and estimates the test generation effort.

A large variety of test generation methods are available. However, only a few methods are available in ATVG utilities. They are computationally expensive and employ unsophisticated algorithms. The TEGAS test vector generation utility, which employs a path sensitization method, should be applied to small devices.

Most logic simulation systems include a random test pattern generator. The random test pattern generator can provide a high degree of fault coverage and supports Agrawal's method of estimating the test generation effort. This method should be applied to larger devices. Future cost improvements in fault simulation hardware and software will make random pattern generation even more attractive.

APPENDICES

APPENDIX A

DEFINITION OF TERMS AND ABBREVIATIONS

AC TESTING: Also known as dynamic or parametric testing, verifies the time related behavior of the device and the magnitudes of output voltage and current levels.

ALU: Arithmetic Logic Unit

ATE: Automatic Test Equipment

ATG: Automatic Test Generation

ATVG: Automatic Test Vector Generation

AVLSI: Advanced Very Large Scale Integration

BILBO: Built-In Logic Block Observer

CLOCK RATE TESTING: Similar to DC testing but occurs at clock frequencies near the device maximum. Clock rate testing is performed to reduce device test time and to prevent data loss in time dependent circuits such as MOS memories.

CHDL: Computer Hardware Description Language

CMOS: Complementary Metal Oxide Semiconductor

COMPLETE TEST: A test covering all stuck-at-faults that can be accessed.

CONTROLLABILITY: The ability to force a selected circuit node to a desired logic state.

DC TESTING: Also known as static or functional testing, consists of applying test vectors to the device and analyzing the corresponding steady state outputs to determine whether the functional behavior is correct.

DEVICE: A general term for an item of interest such as an LSI chip.

DUT: Device Under Test

FAILURE: The occurrence of an inability of a device to perform according to specification.

FAULT: A physical state or condition of a device which may cause failure.

FAULTED MACHINE (MODEL): A logic network modeled with one or more fault conditions.

FDM: Functional Data Modules

FUNCTIONAL TEST: A test which examines the ability of a device to operate according to the functional specifications.

GOOD MACHINE (MODEL): A logic network modeled without a fault.

HILDO: Highly Integrated Logic Device Observer

HOMING SEQUENCE: A series of input vectors which, when applied to a sequential circuit, bring it into a previously defined state.

IC: Integrated Circuit

I/O: Input/Output

LDL: Logic Design Language

LFSR: Linear Feedback Shift Register

LOGIC FAULT: A fault which causes the logic function of a device to be changed to some other function. A typical logical fault causes the circuit signal to be fixed at a constant value.

LOGIC SHORT FAULT: A fault in logic circuitry in which a short circuit exists between logic nets.

LOGIC TEST: A test for the logic or switching function of a device. The test may consist of many test vectors.

LSI: Large Scale Integration

LSSD: Level Sensitive Scan Design

MOS: Metal Oxide Semiconductor

MSI: Medium Scale Integration

NET: A group of interconnected circuit nodes.

OBSERVABILITY: The ability to propagate the logic state of a selected circuit node to an accessible test point.

PARAMETRIC FAULT: A parametric fault typically alters the magnitude of a circuit parameter causing a change in circuit speed, current, or voltage.

PATTERN: A set of logic vectors describing the state of input or output modes of a device. Typically, expressed as a vector applied to a device for the purpose of simulation or testing.

PLA: Programmable Logic Array

RAM: Random Access Memory

READOUT SEQUENCE: A pattern sequence, applied to the input of a sequential circuit, which propagates the state of the device to an accessible test point.

ROM: Read Only Memory

RTL: Register Transfer Language

SENSITIZED PATH: The path from the failure site to an accessible test point.

SIMULATION: Exercising a descriptive model of a logic network on a computer to analyze its functional behavior.

SPOOF: Structure Parity Observation Output Function

SSI: Small Scale Integration

STRUCTURAL TEST: A test based on the physical structure or layout of a device.

STUCK-AT-FAULT: A signal which becomes fixed at a constant value.

TEST: A procedure which examines the ability of a device to conform to specification.

TESTABILITY: The percentage of detectable faults that can possibly occur in a model which is accessible to testing compared with the faults which can possibly occur.

TEST COVERAGE: The percentage of possible faults for a device for which test vectors are provided.

TESTABILITY MEASUREMENT UTILITY: A logic simulation utility which infers the difficulty of testing a circuit node by a simple algorithm.

TEST COVERAGE: The percentage of possible faults for a device for which test vectors are provided.

TDAC: Test Data Augmentation and Conversion

TEST PATTERN: Logic vectors or states applied to the inputs of a logic network and performing one or more logic tests.

TEST SEQUENCE: A series of test vectors applied in a specific order.

TEST VECTOR: A logic state applied to the input of a logic network.

TPFF: Test Point Flip-Flop

TPG: Test Pattern Generation

VLSI: Very Large Scale Integration

APPENDIX B

TEST TECHNIQUE

(extracted from Buehler 1982)

The Test Circuit: A Two-bit Adder with Ripple Carry

For ease of analysis, the five techniques in the study were compared by considering only the testing of the combinational network in the classical finite-state machine shown in Figure 9. To keep the network simple yet meaningful, we chose the ripple carry adder with no carry-in shown in Figure 10. This figure will be used again in subsequent circuits. The adder adds two numbers, A and B, each represented by two bits vectors, a=(a1,a0) and b=(b1,b0), to produce the sum S represented by the vector s=(s1,s0). As seen in Figure 10, the adder consists of a half adder whose inputs are a0 and b0, and whose outputs are s0 and c1. The carry out of the half adder c1 is an input along with a1 and b1 to the full adder which produces the carry out c2 and the sum bit s1. The relevant switching expressions for the half adder are

$$s0 = a0 + b0 \quad (46)$$

$$c1 = a0 * b0 \quad (47)$$

and for the full adder are

$$s1 = (a1 + b1) + c1 \quad (48)$$

$$c2 = (a1 + b1) * c1 + a1 * b1 \quad (49)$$

Figure 11 shows an implementation for the above switching expressions that can be realized in hardware using NAND and XOR gates.

To efficiently test for the structural integrity of the adder circuit, a minimum test set was found that detects all single stuck at faults. For the two bit adder implementation shown in Figure 11, tests were developed assuming a stuck-fault model at each of 26 fault points indicated by X's. That is, for each of 16 possible faults, the output was observed when the circuit was

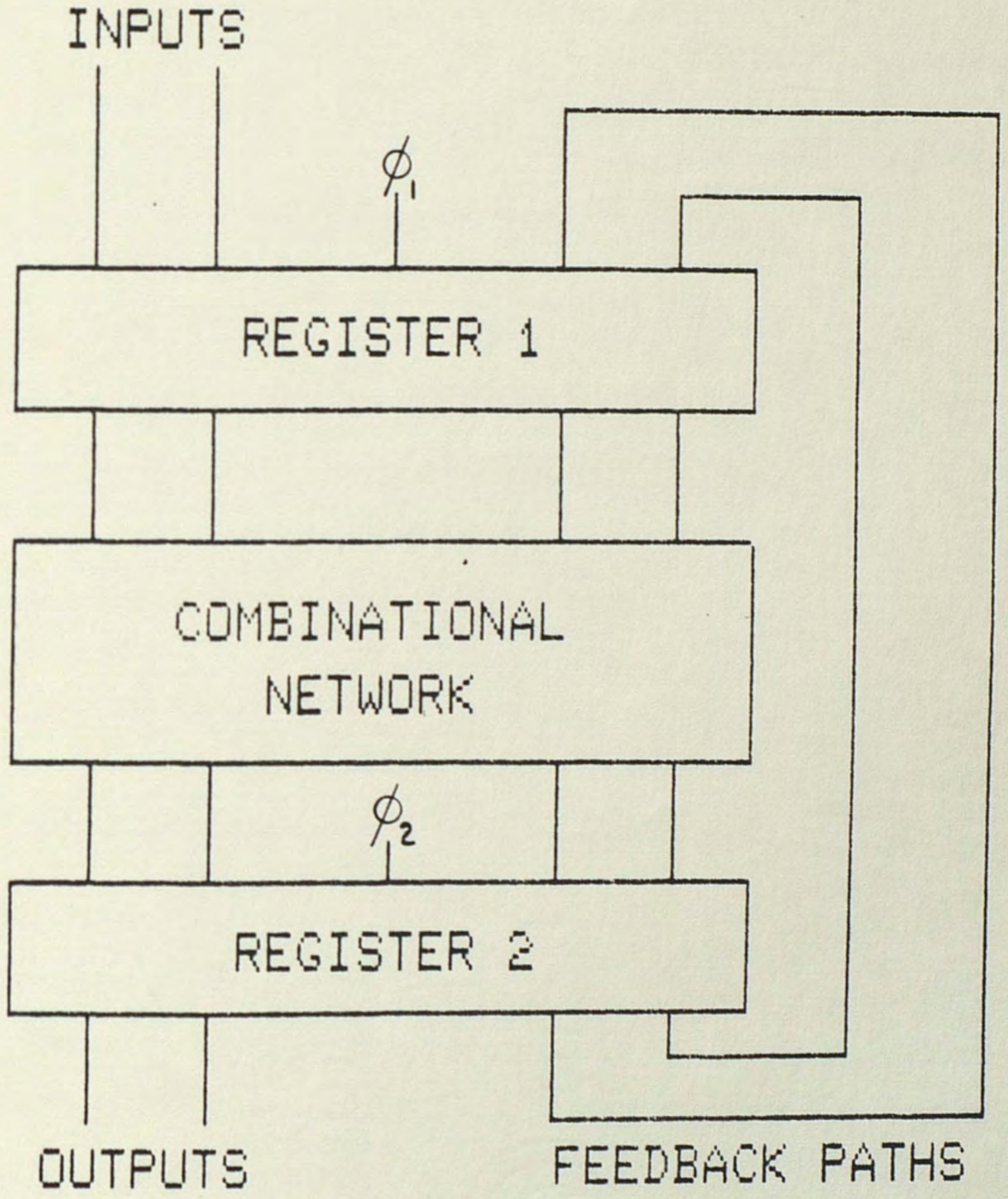


Figure 9. Finite State Machine

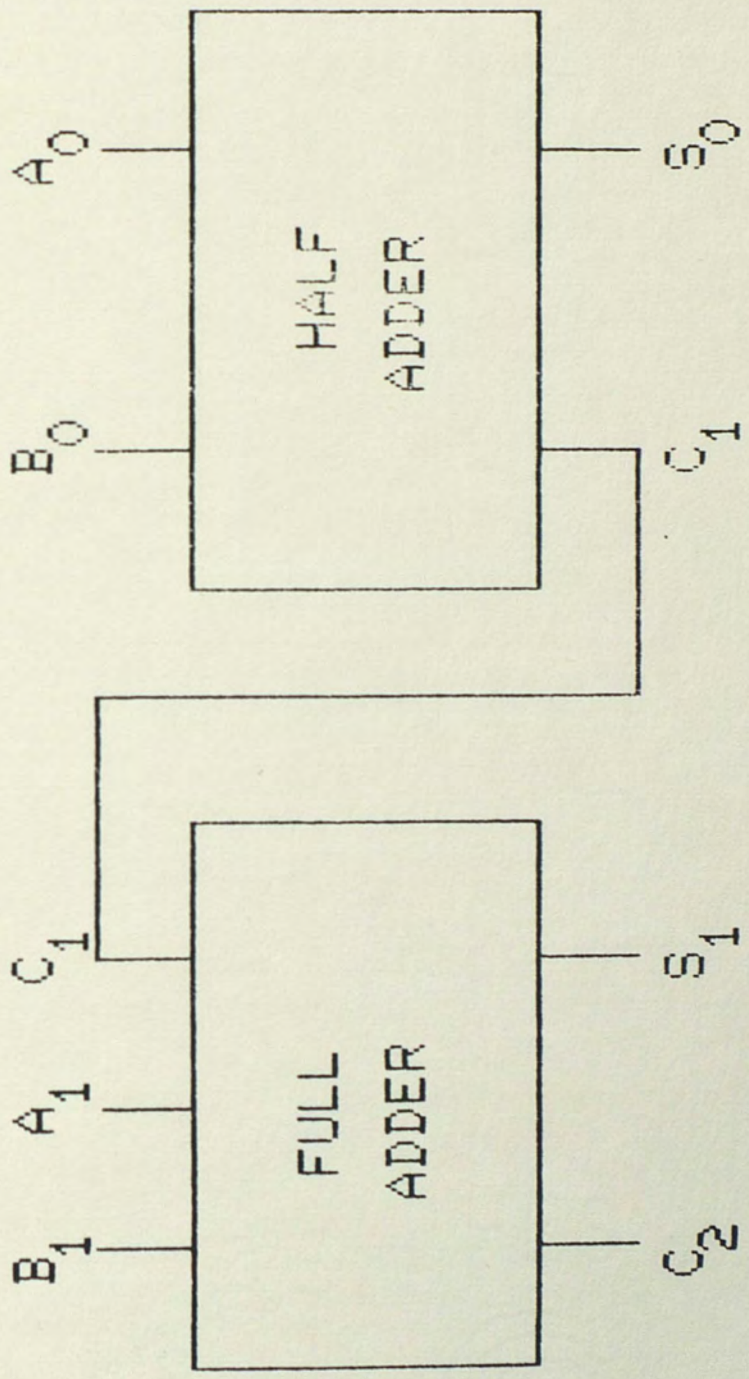


Figure 10. Block Diagram of Adder Circuit

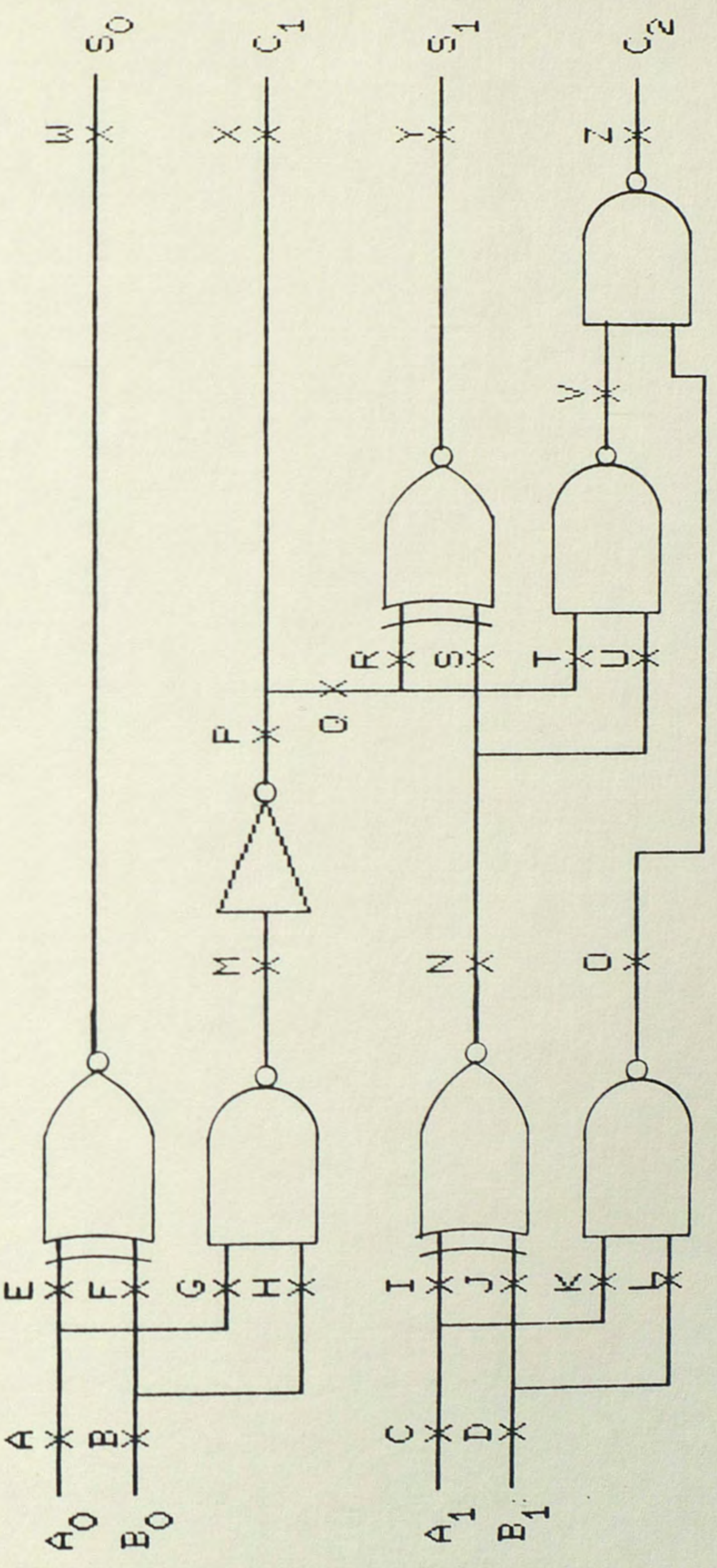


Figure 11. Gate Level Representation of Adder

faulted by both a stuck-at-one and a stuck-at-zero fault at each fault point. It will be assumed that when a point is faulted, the point is disconnected from upstream signals and is either s-a-1 or s-a-0 with respect to downstream signals. For example, a fault at E does not affect the A to G signal path.

The result of a stuck fault analysis for the adder circuit shown in Figure 11 is given in Figure 12, which shows the output at nodes Z,Y,X,W (GOOD) for each of 16 possible tests. The stuck faults are indicated by the notation listed in the left-hand column. For example, (A/0) indicates that node A is stuck-at-zero, (A/1) implies that node A is stuck-at-one, etc. A fault is detected by a specific test when the output differs from the good machine output; these are indicated by the asterisks in Figure 12.

In the following analysis, a test set for the adder circuit is determined that detects 100 percent of the stuck faults. The test set can be obtained by inspection of Figure 12. The process can be simplified by first eliminating faults that produce identical output for all 16 tests. By inspecting Figure 12 or from a knowledge of indistinguishable faults at each gate, the following fault groups are detectable by the same test set:

(a) G/0, H/0, M/1, P/0
(b) K/0, L/0, O/1
(c) T/0, U/0, V/1
(d) O/0, V/0, Z/1
(e) M/0, P/1

This process, known as fault collapsing, reduces the number of faults listed in Figure 12 from 52 to 42.

From this reduced list, a set of five tests [<u>sic</u>] was found as follows. Inspection of a collapsed fault list reveal that some faults are detected by only a few tests. For example, U/1 is detected only by test T3, so this test must be included in the test set. As an intermediate step, a test set was determined for those faults detected by four or fewer tests. When the remaining faults were examined, this test set was found sufficient to detect all faults. The test set for 100 percent fault coverage of the adder circuit consists of five tests: T3, T5, T7, T10, and T12.

```
ADD 9      10-JUN-1981 19:16

   TEST     0    1    2    3    4    5    6    7
   INPUT A 0    1    0    1    0    1    0    1
   INPUT B 0    0    1    1    0    0    1    1
   INPUT C 0    0    0    0    1    1    1    1
   INPUT D 0    0    0    0    0    0    0    0
ZYXW(GOOD)0000 0001 0001 0110 0100 0101 0101 1010

   (A 0 )   0000 000* 0001 0*** 0100 010* 0101 ****
   (A 1 )   000* 0001 0001 0110 010* 0101 **** 1010
   (B 0 )   0000 0001 0001 0*** 0100 0101 010* ****
   (B 1 )   000* 0*** 0001 0110 010* **** 0101 1010
   (C 0 )   0000 0001 0001 0110 0*00 0*01 0*01 **10
   (C 1 )   0*00 0*01 0001 **10 0100 0101 0101 1010
   (D 0 )   0000 0001 0001 0110 0100 0101 0101 1010
   (D 1 )   0*00 0*01 0001 **10 **00 **01 **01 1*10
   (E 0 )   0000 000* 0001 011* 0100 010* 0101 101*
   (E 1 )   000* 0001 0001 0110 010* 0101 010* 1010
   (F 0 )   0000 0001 0001 011* 0100 0101 010* 101*
   (F 1 )   000* 000* 0001 0110 010* 010* 0101 1010
   (G 0 )   0000 0001 0001 0**0 0100 0101 0101 ***0
   (G 1 )   0000 0001 0001 0110 0100 0101 ***1 1010
   (H 0 )   0000 0001 0001 0**0 0100 0101 0101 ***0
   (H 1 )   0000 0**1 0001 0110 0100 ***1 0101 1010
   (I 0 )   0000 0001 0001 0110 0*00 0*01 0*01 **10
   (I 1 )   0*00 0*01 0001 **10 0100 0101 0101 1010
   (J 0 )   0000 0001 0001 0110 0100 0101 0101 1010
   (J 1 )   0*00 0*01 0001 **10 0*00 0*01 0*01 **10
   (K 0 )   0000 0001 0001 0110 0100 0101 0101 1010
   (K 1 )   0000 0001 0001 0110 0100 0101 0101 1010
   (L 0 )   0000 0001 0001 0110 0100 0101 0101 1010
   (L 1 )   0000 0001 0001 0110 *100 *101 *101 1010
   (M 0 )   0**0 0**1 0001 0110 ***0 ***1 ***1 1010
   (M 1 )   0000 0001 0001 0**0 0100 0101 0101 ***0
   (N 0 )   0000 0001 0001 0110 0*00 0*01 0*01 **10
   (N 1 )   0*00 0*01 0001 **10 0100 0101 0101 1010
   (O 0 )   *000 *001 0001 *110 *100 *101 *101 1010
   (O 1 )   0000 0001 0001 0110 0100 0101 0101 1010
   (P 0 )   0000 0001 0001 0**0 0100 0101 0101 ***0
   (P 1 )   0**0 0**1 0001 0110 ***0 ***1 ***1 1010
```

Figure 12. Fault Analysis of Adder Circuit

```
  TEST     8    9    10   11   12   13   14   15
  INPUT A 0    1    0    1    0    1    0    1
  INPUT B 0    0    1    1    0    0    1    1
  INPUT C 0    0    0    0    1    1    1    1
  INPUT D 1    1    1    1    1    1    1    1
ZYXW(GOOD)0100 0101 0101 1010 1000 1001 1001 1110

  (A 0 )  0100 010* 0101 **** 1000 100* 1001 1***
  (A 1 )  010* 0101 **** 1010 100* 1001 1*** 1110
  (B 0 )  0100 0101 010* **** 1000 1001 100* 1***
  (B 1 )  010* **** 0101 1010 100* 1*** 1001 1110
  (C 0 )  0100 0101 0101 1010 **00 **01 **01 1*10
  (C 1 )  **00 **01 **01 1*10 1000 1001 1001 1110
  (D 0 )  0*00 0*01 0*01 **10 **00 **01 **01 1*10
  (D 1 )  0100 0101 0101 1010 1000 1001 1001 1110
  (E 0 )  0100 010* 0101 101* 1000 100* 1001 111*
  (E 1 )  010* 0101 010* 1010 100* 1001 100* 1110
  (F 0 )  0100 0101 010* 101* 1000 1001 100* 111*
  (F 1 )  010* 010* 0101 1010 100* 100* 1001 1110
  (G 0 )  0100 0101 0101 ***0 1000 1001 1001 1**0
  (G 1 )  0100 0101 ***1 1010 1000 1001 1**1 1110
  (H 0 )  0100 0101 0101 ***0 1000 1001 1001 1**0
  (H 1 )  0100 ***1 0101 1010 1000 1**1 1001 1110
  (I 0 )  0100 0101 0101 1010 1*00 1*01 1*01 1*10
  (I 1 )  0*00 0*01 0*01 **10 1000 1001 1001 1110
  (J 0 )  0*00 0*01 0*01 **10 1*00 1*01 1*01 1*10
  (J 1 )  0100 0101 0101 1010 1000 1001 1001 1110
  (K 0 )  0100 0101 0101 1010 *000 *001 *001 *110
  (K 1 )  *100 *101 *101 1010 1000 1001 1001 1110
  (L 0 )  0100 0101 0101 1010 *000 *001 *001 *110
  (L 1 )  0100 0101 0101 1010 1000 1001 1001 1110
  (M 0 )  ***0 ***1 ***1 1010 1**0 1**1 1**1 1110
  (M 1 )  0100 0101 0101 ***0 1000 1001 1001 1**0
  (N 0 )  0*00 0*01 0*01 **10 1000 1001 1001 1110
  (N 1 )  0100 0101 0101 1010 1*00 1*01 1*01 1*10
  (O 0 )  *100 *101 *101 1010 1000 1001 1001 1110
  (O 1 )  0100 0101 0101 1010 *000 *001 *001 *110
  (P 0 )  0100 0101 0101 ***0 1000 1001 1001 1**0
  (P 1 )  ***0 ***1 ***1 1010 1**0 1**1 1**1 1110
```

Figure 12.--Continued

```
    TEST      0    1    2    3    4    5    6    7
    INPUT A 0    1    0    1    0    1    0    1
    INPUT B 0    0    1    1    0    0    1    1
    INPUT C 0    0    0    0    1    1    1    1
    INPUT D 0    0    0    0    0    0    0    0
ZYXW(GOOD)0000 0001 0001 0110 0100 0101 0101 1010

    (Q 0 )  0000 0001 0001 0*10 0100 0101 0101 **10
    (Q 1 )  0*00 0*01 0*01 0110 **00 **01 **01 1010
    (R 0 )  0000 0001 0001 0*10 0100 0101 0101 1*10
    (R 1 )  0*00 0*01 0*01 0110 0*00 0*01 0*01 1010
    (S 0 )  0000 0001 0001 0110 0*00 0*01 0*01 1*10
    (S 1 )  0*00 0*01 0*01 0*10 0100 0101 0101 1010
    (T 0 )  0000 0001 0001 0110 0100 0101 0101 *010
    (T 1 )  0000 0001 0001 0110 *100 *101 *101 1010
    (U 0 )  0000 0001 0001 0110 0100 0101 0101 *010
    (U 1 )  0000 0001 0001 *110 0100 0101 0101 1010
    (V 0 )  *000 *001 *001 *110 *100 *101 *101 1010
    (V 1 )  0000 0001 0001 0110 0100 0101 0101 *010
    (W 0 )  0000 000* 000* 0110 0100 010* 010* 1010
    (W 1 )  000* 0001 0001 011* 010* 0101 0101 101*
    (X 0 )  0000 0001 0001 01*0 0100 0101 0101 10*0
    (X 1 )  00*0 00*1 00*1 0110 01*0 01*1 01*1 1010
    (Y 0 )  0000 0001 0001 0*10 0*00 0*01 0*01 1010
    (Y 1 )  0*00 0*01 0*01 0110 0100 0101 0101 1*10
    (Z 0 )  0000 0001 0001 0110 0100 0101 0101 *010
    (Z 1 )  *000 *001 *001 *110 *100 *101 *101 1010
```

Figure 12.--Continued

TEST	8	9	10	11	12	13	14	15
INPUT A	0	1	0	1	0	1	0	1
INPUT B	0	0	1	1	0	0	1	1
INPUT C	0	0	0	0	1	1	1	1
INPUT D	1	1	1	1	1	1	1	1
ZYXW(GOOD)	0100	0101	0101	1010	1000	1001	1001	1110
(Q 0)	0100	0101	0101	**10	1000	1001	1001	1*10
(Q 1)	**00	**01	**01	1010	1*00	1*01	1*01	1110
(R 0)	0100	0101	0101	1*10	1000	1001	1001	1*10
(R 1)	0*00	0*01	0*01	1010	1*00	1*01	1*01	1110
(S 0)	0*00	0*01	0*01	1*10	1000	1001	1001	1110
(S 1)	0100	0101	0101	1010	1*00	1*01	1*01	1*10
(T 0)	0100	0101	0101	*010	1000	1001	1001	1110
(T 1)	*100	*101	*101	1010	1000	1001	1001	1110
(U 0)	0100	0101	0101	*010	1000	1001	1001	1110
(U 1)	0100	0101	0101	1010	1000	1001	1001	1110
(V 0)	*100	*101	*101	1010	1000	1001	1001	1110
(V 1)	0100	0101	0101	*010	1000	1001	1001	1110
(W 0)	0100	010*	010*	1010	1000	100*	100*	1110
(W 1)	010*	0101	0101	101*	100*	1001	1001	111*
(X 0)	0100	0101	0101	10*0	1000	1001	1001	11*0
(X 1)	01*0	01*1	01*1	1010	10*0	10*1	10*1	1110
(Y 0)	0*00	0*01	0*01	1010	1000	1001	1001	1*10
(Y 1)	0100	0101	0101	1*10	1*00	1*01	1*01	1110
(Z 0)	0100	0101	0101	*010	*000	*001	*001	*110
(Z 1)	*100	*101	*101	1010	1000	1001	1001	1110

Figure 12.--Continued

Off-line Test Techniques

Self-Oscillation

For the adder of Figure 11, the self-oscillation test circuit is simple to implement as shown in Figure 13. In the add mode (sigma =1), data are applied to the adder and outputs obtained in the usual manner. In the oscillation mode (sigma=1), the feedback path between c2 and b0 is activated, and the circuit oscillates at a frequency determined by the gate delays and stray capacitances. As seen in Figure 13, the c2 bit is inverted, fed back and inserted at b0. The path of oscillation through the adder is illustrated by the heavy line in Figure 14. Oscillations will stop if a faults occurs on the oscillation path or if a gate is not operational.

For oscillation to occur, certain gates in the adder and feedback path must be sensitized. For

$$ob_1 a_1 b_0 a_0 = 00101, \qquad (50)$$

the circuit oscillates between tests T5 and T7 listed in Figure 12. For

$$= 01001, \qquad (51)$$

the circuit oscillates between tests T9 and T11. The path of oscillation is illustrated in Figure 14 by the heavy line. Oscillations will cease if a fault occurs on the oscillation path of if a gate is not properly sensitized. The oscillation is is illustrated by the two additions shown in Figure 15. For the sensitizing condition

$$= 00101, \qquad (52)$$

the c2 bi is inverted and inserted at b0, thus sustaining the oscillations.

Results for a fault analysis of the circuit, shown in Figure 14, are listed in Table 7. For the four oscillation conditions listed, 80 percent of the faults in the adder and feedback circuits can be detected by reverting to the add mode and by applying tests T14 and T3.

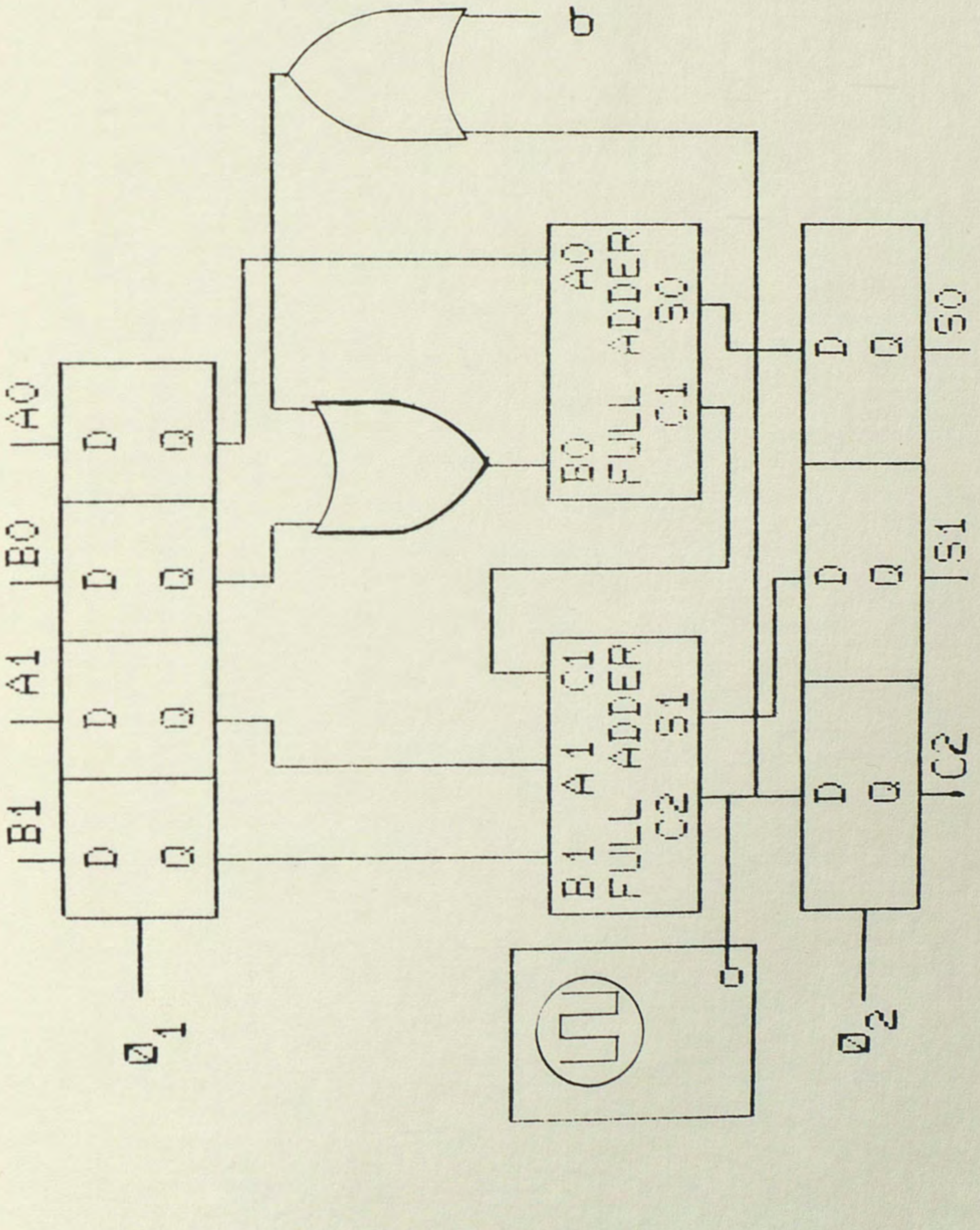


Figure 13. Self-Oscillation Test Circuit

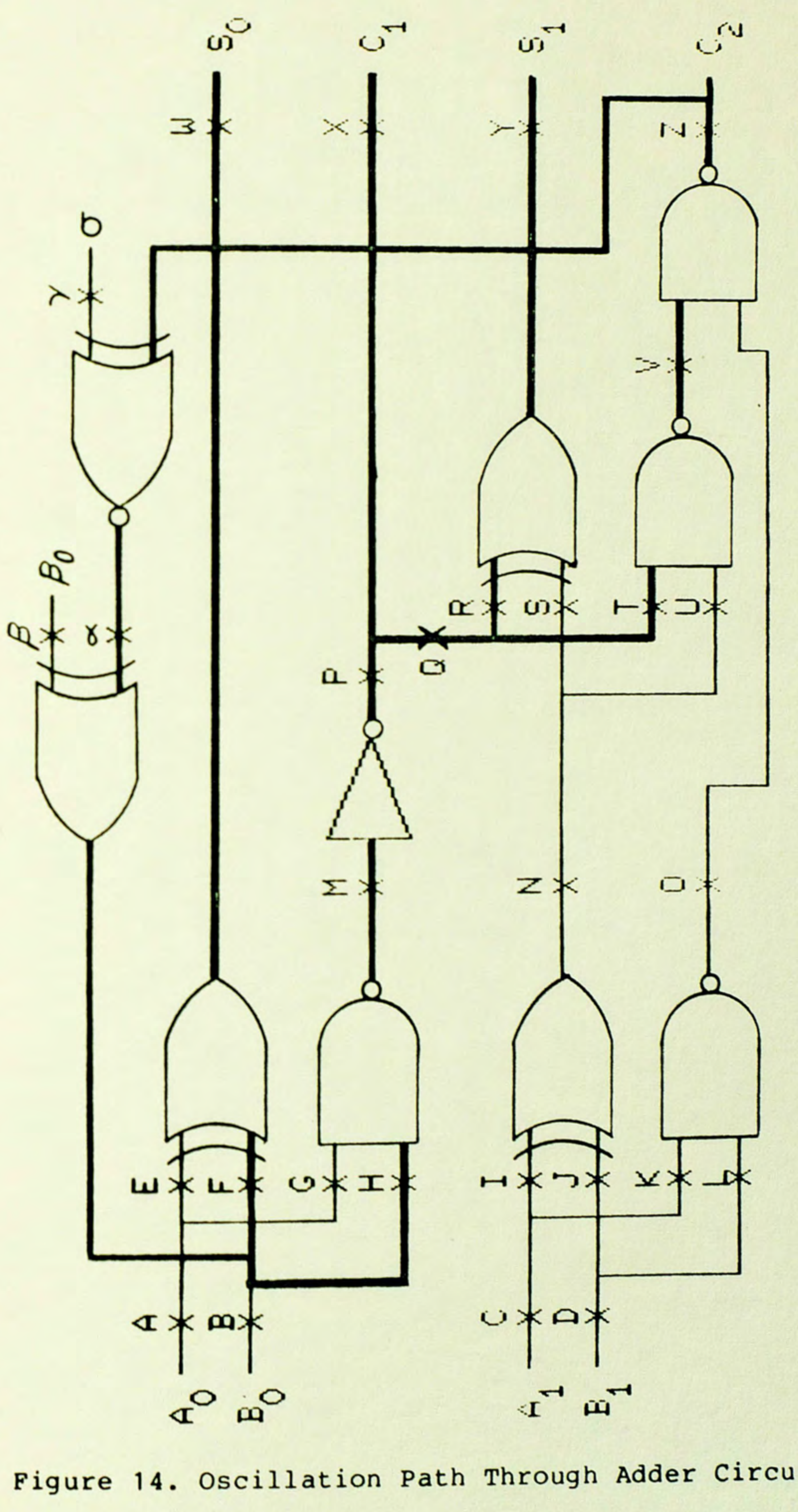


Figure 14. Oscillation Path Through Adder Circuit

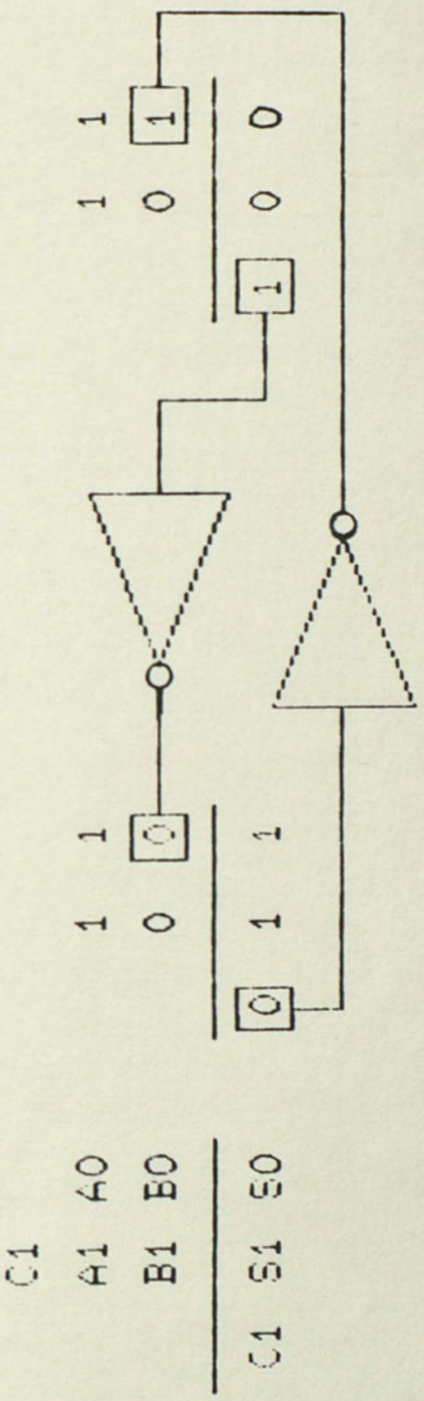


Figure 15. Oscillation Sequence for the
Self-Oscillation Test Circuit

TABLE 7

TEST SEQUENCE AND FAULTS DETECTED FOR
THE SELF-OSCILLATION TEST CIRCUIT

MODE	INPUT B A B A 1 1 0 0	SIGMA	FAULT DETECTED	EFFECT OF FAULT
OSC. AT Z	T =0101 5	0	A/0 B/0 B/1 C/0 D/1 G/0 H/0 H/1 I/0 J/1 L/1 M/0 M/1 N/0 O/0 P/0 P/1 T/0 T/1 U/0 V/0 V/1 Z/0 Z/1	STOP OSC.
OSC. AT Y	T =0101 5	0	R/0 R/1 Y/0 Y/1 S/0	STOP OSC. Y IN PHASE WITH Z
OSC. AT W	T =0101 5	0	F/0 F/1 W/0 W/1 E/0	STOP OSC. W IN PHASE WITH Z
OSC. AT Z	T =1001 9	0	C/1 D/0 I/1 J/0 K/1	STOP OSC.
ADD	T =1110 14	1	A/1 E/1 G/1 K/0 L/0 N/1 O/1 S/1 /0 /0	ZYW / 101
ADD	T =0011 3	1	U/1	ZYW / 101

The self-oscillation test circuit is simple and is easily tested for faults. This technique has been used successfully to test high-speed gallium arsenide ICs, an application for which commercial test equipment with the required speed is unavailable. Long (1980) has used the technique to verify the operation of 5x5 and 8x8-bit parallel multipliers operating around 200 MHz.

Self-Comparison

The basic philosophy in a self-comparison scheme is to partition the circuit under test in such a manner that similar functional elements can be coerced into producing identical outputs for a given set of input values in a fault-free environment. Figure 16 shows how self-comparison is applied to our [sic] adder example. The figure shows a comparator that continuously monitors the outputs of the half and full adder. Forcing equality in the outputs of these modules may be accomplished in at least two ways.

Circuit reconfiguration is the simplest way to obtain equal output values. This scheme requires that, during the test mode, the carry out from the half-adder c1 be disconnected from the carry-in of the full adder. The full adder carry-in is set to the value zero, thereby causing the full adder to behave as if it were a half adder. Inputs (a1,b1) and (a0,b0) are set equal, and the output of the comparator is observed. There are three basic problems with this approach. First, considerable overhead is needed to accomplish reconfiguration. Second, it is necessary to verify that the reconfiguration circuit is functional, which may not be possible. Finally, since the basic circuit is changed to perform testing, it is not possible to check all paths through the "real" configuration. For example, it will not be possible to determine whether the carry-out c1 of the half adder is correctly connected to the carry-in of the full adder.

The other method of creating equalities in the output variables is to study the associated switching functions and to carefully analyze the interaction of the modules in the operational configuration. The goal is to find a set of input values for which the modules being compared will output identical results. This approach requires more time to develop a test set but has the advantage of testing an undisturbed circuit. On the disadvantage side,

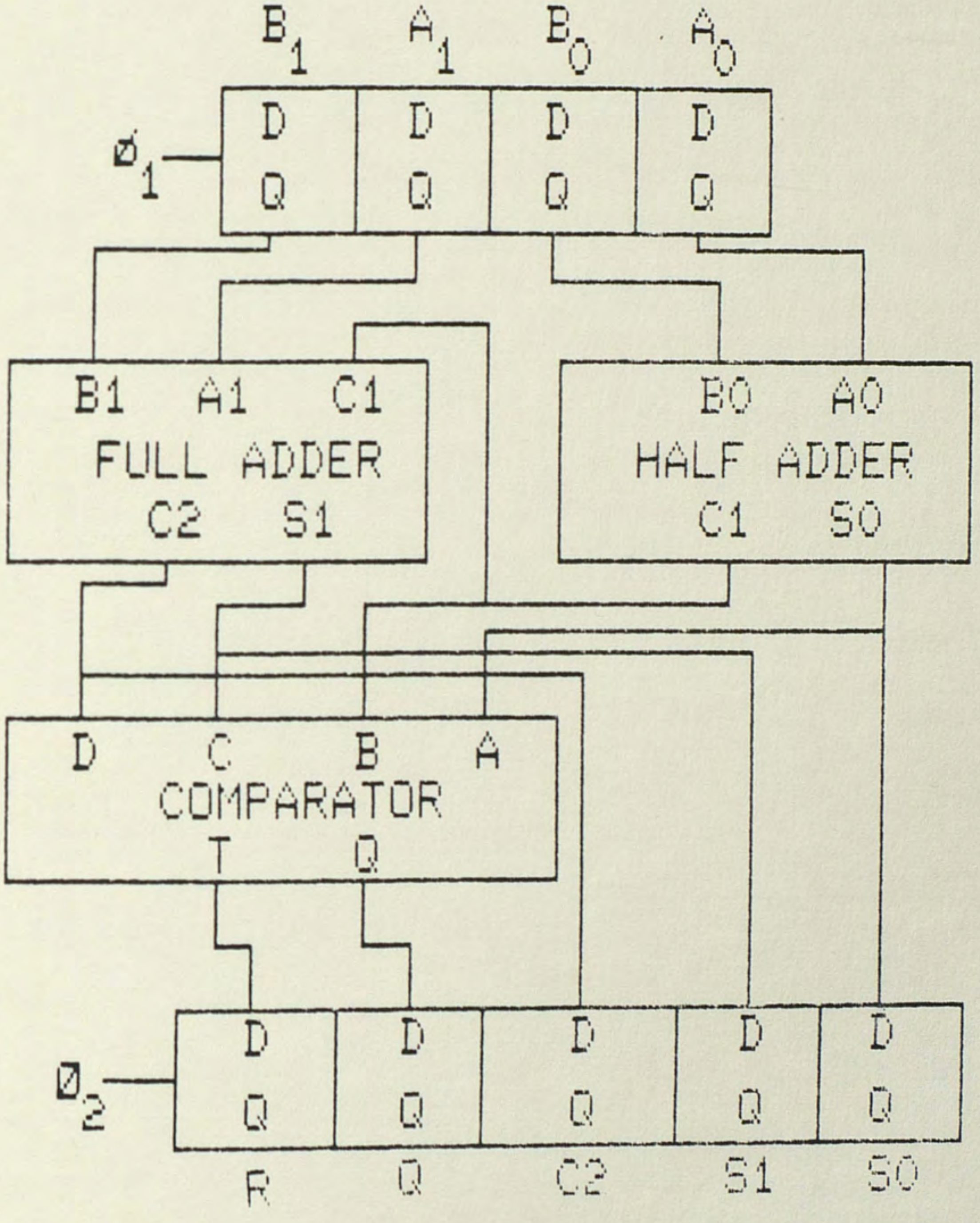


Figure 16. Self-Comparison Test Circuit

however, it may be difficult to generate a sufficient test set to sensitize all paths through both modules.

The hardcore of this testing method is the comparator. That is, this circuit must be operating correctly in order to successfully perform testing. It would seem reasonable, therefore, to require a procedure to test the comparator. This last point is simply a restatement of the famous question "who checks the checker?" To a large degree this question can be answered by designing a checker that checks itself. These circuits are called totally self-checking checkers and require special coding of their inputs such that these inputs fall into two sets: G or good and B or bad (refer to Figure 17). Further, the output codewords formed by these circuits are elements of one of two sets: g and b. The operation of a TSC checker when no internal faults are present performs a mapping from

$$G \rightarrow g \tag{53}$$

and

$$B \rightarrow b. \tag{54}$$

In the case where an internal fault is present, there must exist at least one element

$$g' \quad G \tag{55}$$

such that

$$g' \rightarrow b \quad . \tag{56}$$

It is essential to note that it is impossible to determine from the output of a TSC checker whether the source of a faults was the checker of the circuit being checked. For this reason it will always be assumed that the checker is an integral part of the circuit being checked, and therefore any fault indication implies that the unit is bad. Figure 18 shows a typical TSC comparator. Input codewords in the set

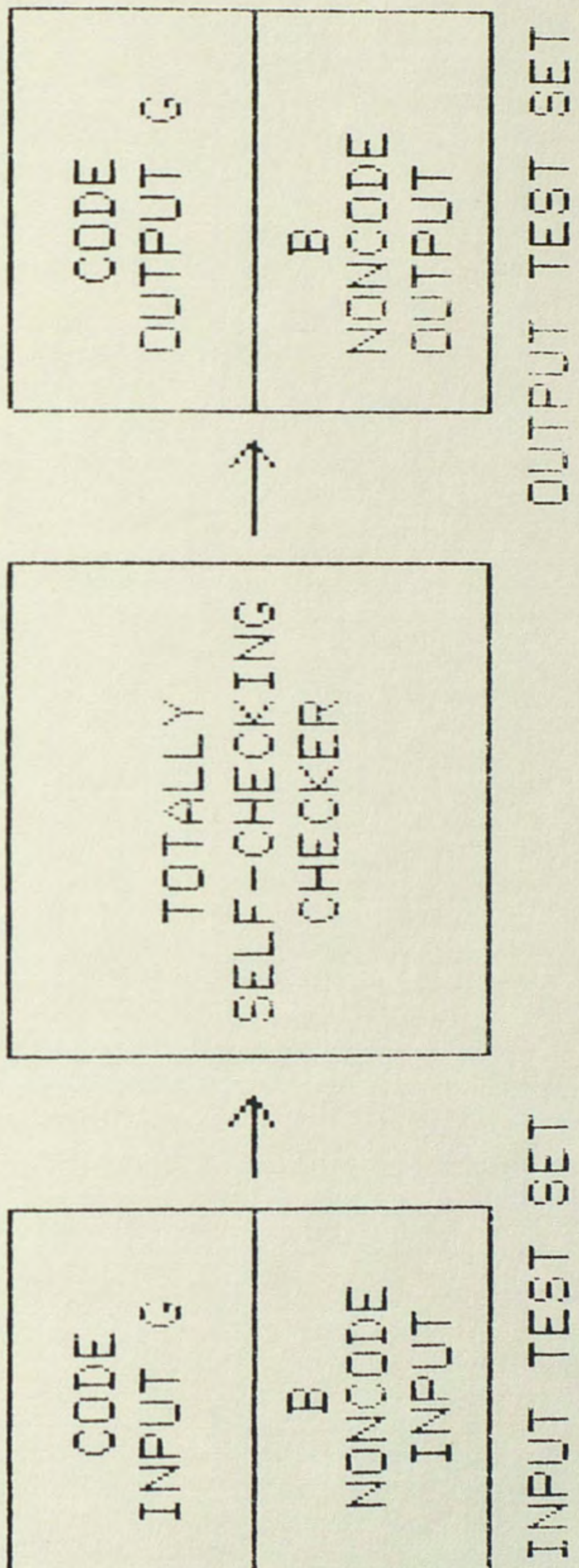


Figure 17. Operation of a Totally
Self-Checking Checker Circuit

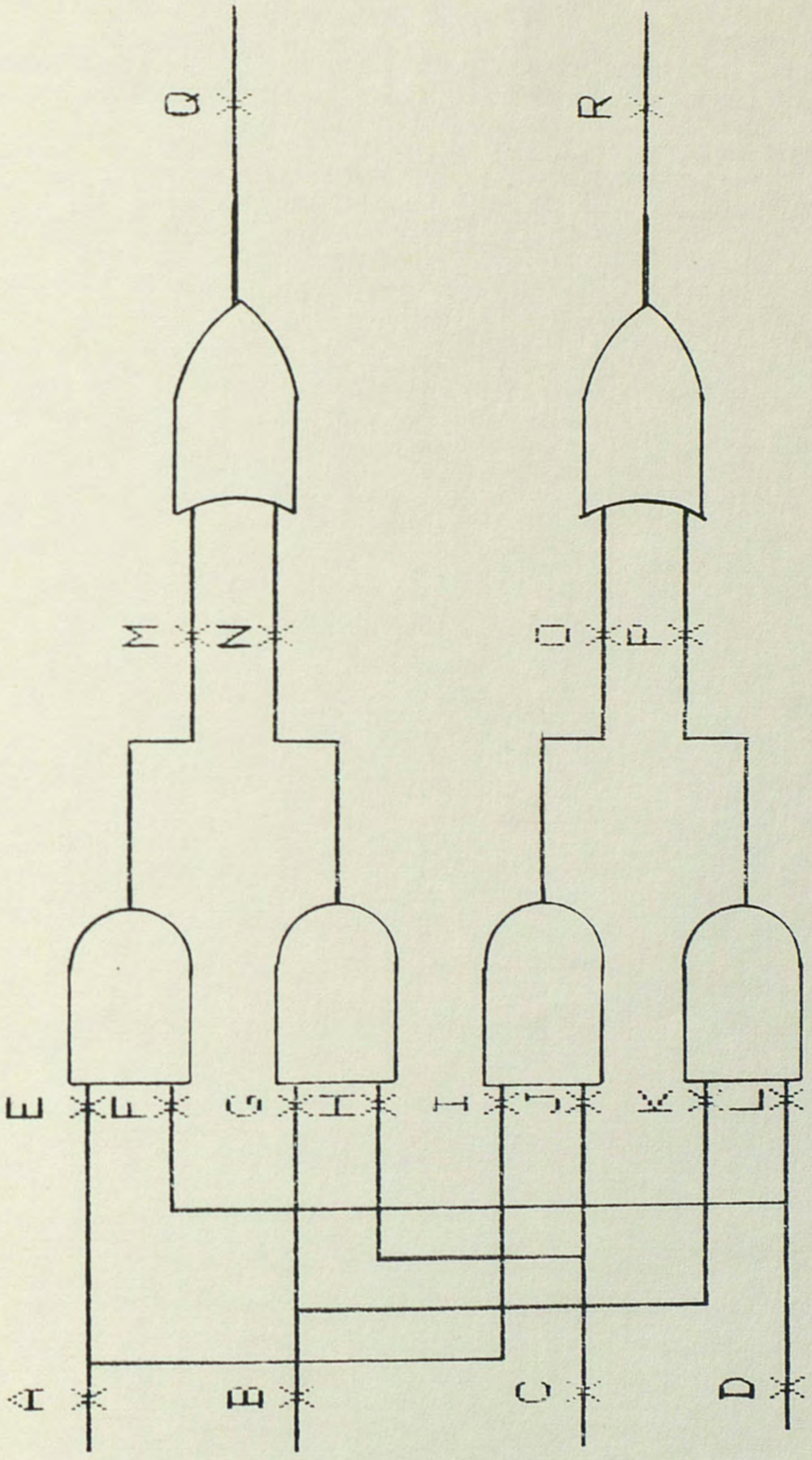


Figure 18. A Totally Self-Checking Comparator

[a / b and c = d] (57)

fall into the set G. All other codewords are considered bad. The outputs

[q / r] (58)

are elements of g; conversely,

[q = r] (59)

belong to b.

A fault analysis for the comparator shown in Figure 18 is given in Figure 19. An examination of Figure 19 indicates that valid codewords are obtained for tests T5, T6, T9, and T10. Notice that the requirements mentioned above for good codewords hold only for these tests.

The test set needed to detect all single faults in the adder circuit is developed to be compatible with the codeword requirements of the comparator. (Recall that the adder circuit is shown in Figure 10, and its fault analysis is given in Figure 12.) An examination of Figure 12 indicates that nine tests, T3, T5, T6, T7, T9, T10, T11, T13, and T14, are candidates because they form valid input codewords for the comparator. Before determining the test set, fault responses that produce codewords

(Z / Y) (60)

and

(X / W) (61)

must be eliminated from Figure 12. These responses would lead to the conclusion that the adder is fault free when in fact it is faulty. An intermediate test set

```
COM 5       20-AUG-1981 10:49
 TEST      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 INPUT A 0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
 INPUT B 0  0  1  1  0  0  1  1  0  0  1  1  0  0  1  1
 INPUT C 0  0  0  0  1  1  1  1  0  0  0  0  1  1  1  1
 INPUT D 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1
RQ(GOOD)00 00 00 00 00 10 01 11 00 01 10 11 00 11 11 11

 (A 0 ) 00 00 00 00 00 *0 01 *1 00 0* 10 1* 00 ** 11 11
 (A 1 ) 00 00 00 00 *0 10 *1 11 0* 01 1* 11 ** 11 11 11
 (B 0 ) 00 00 00 00 00 10 0* 1* 00 01 *0 *1 00 11 ** 11
 (B 1 ) 00 00 00 00 0* 1* 01 11 *0 *1 10 11 ** 11 11 11
 (C 0 ) 00 00 00 00 00 *0 0* ** 00 01 10 11 00 *1 1* 11
 (C 1 ) 00 *0 0* ** 00 10 01 11 00 *1 1* 11 00 11 11 11
 (D 0 ) 00 00 00 00 00 10 01 11 00 0* *0 ** 00 1* *1 11
 (D 1 ) 00 0* *0 ** 00 1* *1 11 00 01 10 11 00 11 11 11
 (E 0 ) 00 00 00 00 00 10 01 11 00 0* 10 1* 00 1* 11 11
 (E 1 ) 00 00 00 00 00 10 01 11 0* 01 1* 11 0* 11 11 11
 (F 0 ) 00 00 00 00 00 10 01 11 00 0* 10 1* 00 1* 11 11
 (F 1 ) 00 0* 00 0* 00 1* 01 11 00 01 10 11 00 11 11 11
 (G 0 ) 00 00 00 00 00 10 0* 1* 00 01 10 11 00 11 1* 11
 (G 1 ) 00 00 00 00 0* 1* 01 11 00 01 10 11 0* 11 11 11
 (H 0 ) 00 00 00 00 00 10 0* 1* 00 01 10 11 00 11 1* 11
 (H 1 ) 00 00 0* 0* 00 10 01 11 00 01 1* 11 00 11 11 11
 (I 0 ) 00 00 00 00 00 *0 01 *1 00 01 10 11 00 *1 11 11
 (I 1 ) 00 00 00 00 *0 10 *1 11 00 01 10 11 *0 11 11 11
 (J 0 ) 00 00 00 00 00 *0 01 *1 00 01 10 11 00 *1 11 11
 (J 1 ) 00 *0 00 *0 00 10 01 11 00 *1 10 11 00 11 11 11
 (K 0 ) 00 00 00 00 00 10 01 11 00 01 *0 *1 00 11 *1 11
 (K 1 ) 00 00 00 00 00 10 01 11 *0 *1 10 11 *0 11 11 11
 (L 0 ) 00 00 00 00 00 10 01 11 00 01 *0 *1 00 11 *1 11
 (L 1 ) 00 00 *0 *0 00 10 *1 11 00 01 10 11 00 11 11 11
 (M 0 ) 00 00 00 00 00 10 01 11 00 0* 10 1* 00 1* 11 11
 (M 1 ) 0* 0* 0* 0* 0* 1* 01 11 0* 01 1* 11 0* 11 11 11
 (N 0 ) 00 00 00 00 00 10 0* 1* 00 01 10 11 00 11 1* 11
 (N 1 ) 0* 0* 0* 0* 0* 1* 01 11 0* 01 1* 11 0* 11 11 11
 (O 0 ) 00 00 00 00 00 *0 01 *1 00 01 10 11 00 *1 11 11
 (O 1 ) *0 *0 *0 *0 *0 10 *1 11 *0 *1 10 11 *0 11 11 11
 (P 0 ) 00 00 00 00 00 10 01 11 00 01 *0 *1 00 11 *1 11
 (P 1 ) *0 *0 *0 *0 *0 10 *1 11 *0 *1 10 11 *0 11 11 11
 (Q 0 ) 00 00 00 00 00 10 0* 1* 00 0* 10 1* 00 1* 1* 1*
 (Q 1 ) 0* 0* 0* 0* 0* 1* 01 11 0* 01 1* 11 0* 11 11 11
 (R 0 ) 00 00 00 00 00 *0 01 *1 00 01 *0 *1 00 *1 *1 *1
 (R 1 ) *0 *0 *0 *0 *0 10 *1 11 *0 *1 10 11 *0 11 11 11
```

Figure 19. Fault Analysis for a Totally
Self-Checking Comparator

(T3, T7, T11, T13) (62)

was determined by identifying those faults detected by a single test. A test set (T5 and T9) was then determined for the faults not covered by the previous test. The test set for 100 percent fault coverage of the adder circuit consists of seven tests (T3, T5, T7, T9, T11, T13, and T14). In the self comparison test mode, these seven test are generated, and if

$$q \neq r, \qquad (63)$$

then the adder and comparator are fault free.

Partition

The partition approach to testing is shown in Figure 20. In the add mode (sigma=0), the multiplexer reconfigures the logic so that the full adder and half adder can be tested independently of each other. In the figure, this is accomplished by connecting c13 to c11 and c14 to c12 when sigma equals 1 and c13 to c01 and c14 to c02 when sigma equals 0.

The partition principle states that the number of tests required to exhaustively test the subcircuits of a combinational circuit is fewer than the number of test required to exhaustively test the entire circuit. For example, the half adder with three input requires

$$2^2 = 4 \text{ tests}, \qquad (64)$$

and the full adder with three inputs requires

$$2^3 = 8 \text{ tests}. \qquad (65)$$

Since the test of the half adder and full adder can be performed in parallel, only eight tests are required to exhaustively test the adder circuitry. This amount of testing is less than the number of tests needed to exhaustively test the unpartitioned adder with four inputs, which requires

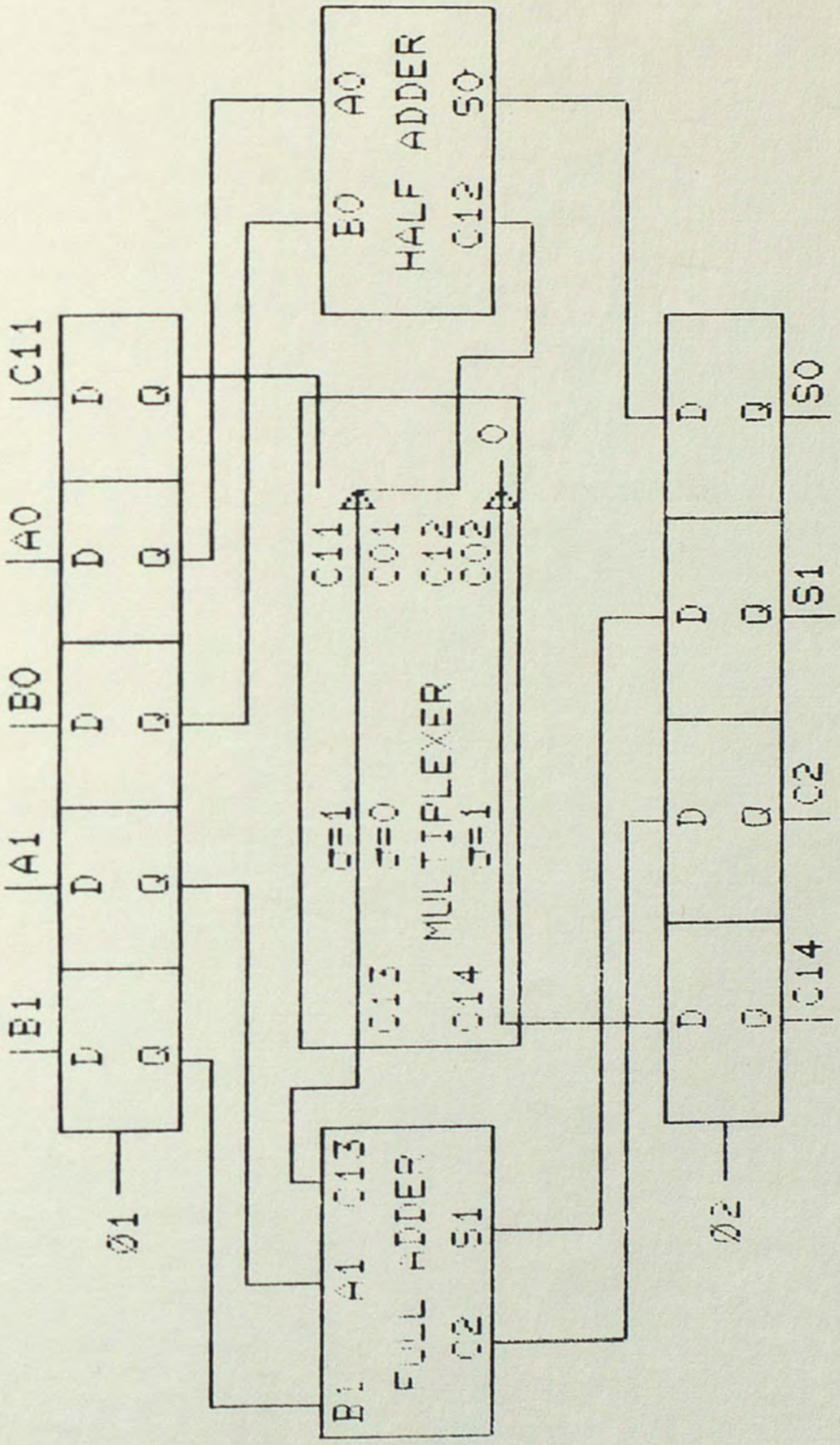


Figure 20. Partition Test Circuit

$$2^4 = 16 \text{ tests.} \qquad (66)$$

When a structural test of the adder and multiplexer was derived, it was discovered that only five tests were needed for 100 percent fault coverage. The multiplexer used in this study is shown in Figure 21. The fault analysis for the entire circuit consists of combining the tests needed to detect 100 percent of the faults in the half adder, the multiplexer, and the full adder. The fault analysis for the multiplexer is given in Figure 22, and the fault analysis for the half adder and full adder can be derived from Figure 12. The fault analysis (see Figure 23) shows that five tests are required for 100 percent fault coverage.

When applied to the testing of a four bit ALU, the partitioning approach dramatically reduced the number of tests. The four bit ALU studied was a commercially available 74181 with 14 control and data inputs. Such a combinational circuit requires

$$2^{14} = 16{,}384 \text{ tests} \qquad (67)$$

to be exhaustively tested, but partitioning the circuit into five parts and testing identical parts in parallel reduced the number of tests required to only 356. This circuit could be partitioned without additional circuitry.

Scan Path

The scan path approach to testing allows access to internal circuit nodes without unduly increasing chip pin count. The technique relies on a special test mode which reconfigures on-chip flip-flops into a shift register. Test patterns can be serially shifted through the reconfigured flip-flops to control various combinational networks and shift out their response vectors. Figure 24 shows an application of scan path testing to the adder circuit in which the test control variable, sigma, operates a multiplexer at the input of each flip-flop. In normal operation, sigma equals 0 and the multiplexer selects parallel load data. The shift mode is selected when sigma equals 1, and the multiplexers are configured

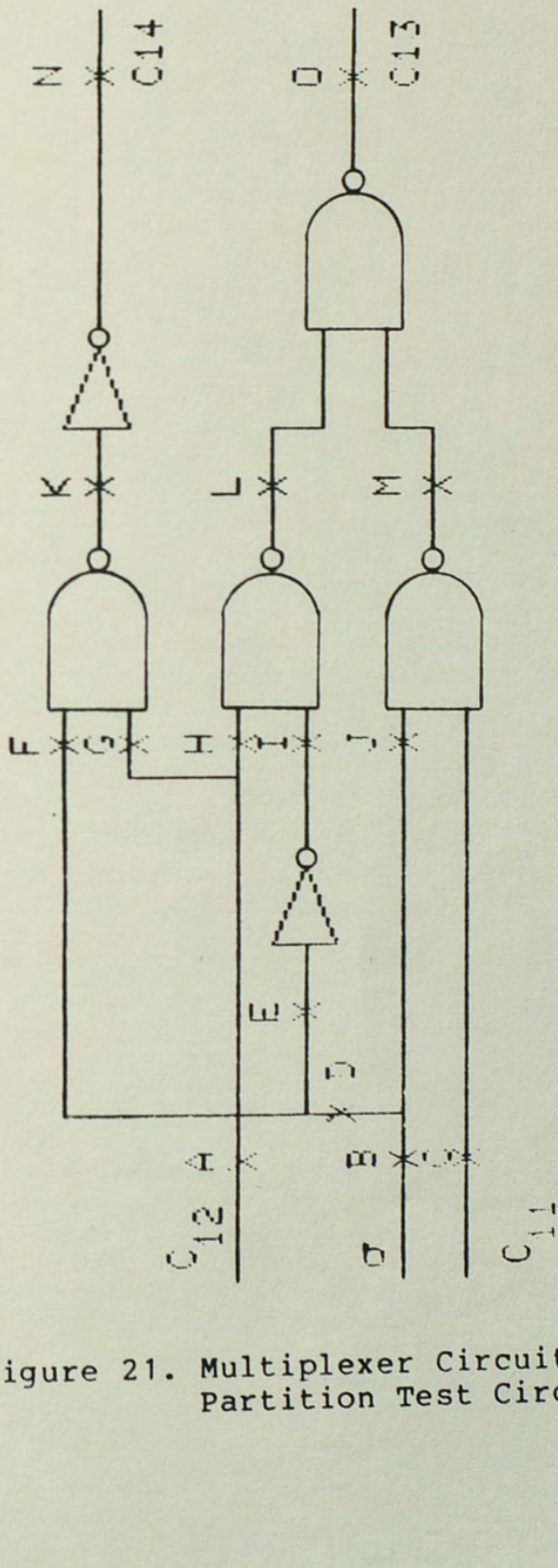


Figure 21. Multiplexer Circuit for the Partition Test Circuit

```
MUX 1       20-AUG-1981 10:46

 TEST      0   1   2   3   4   5   6   7
 INPUT A 0   1   0   1   0   1   0   1
 INPUT B 0   0   1   1   0   0   1   1
 INPUT C 0   0   0   0   1   1   1   1
ON(GOOD)00  10  00  01  00  10  10  11

 (A 0 ) 00  *0  00  0*  00  *0  10  1*
 (A 1 ) *0  10  0*  01  *0  10  1*  11
 (B 0 ) 00  10  00  **  00  10  *0  1*
 (B 1 ) 00  **  00  01  *0  1*  10  11
 (C 0 ) 00  10  00  01  00  10  *0  *1
 (C 1 ) 00  10  *0  *1  00  10  10  11
 (D 0 ) 00  10  00  **  00  10  10  1*
 (D 1 ) 00  **  00  01  00  **  10  11
 (E 0 ) 00  10  00  *1  00  10  10  11
 (E 1 ) 00  *0  00  01  00  *0  10  11
 (F 0 ) 00  10  00  0*  00  10  10  1*
 (F 1 ) 00  1*  00  01  00  1*  10  11
 (G 0 ) 00  10  00  0*  00  10  10  1*
 (G 1 ) 00  10  0*  01  00  10  1*  11
 (H 0 ) 00  *0  00  01  00  *0  10  11
 (H 1 ) *0  10  00  01  *0  10  10  11
 (I 0 ) 00  *0  00  01  00  *0  10  11
 (I 1 ) 00  10  00  *1  00  10  10  11
 (J 0 ) 00  10  00  01  00  10  *0  *1
 (J 1 ) 00  10  00  01  *0  10  10  11
 (K 0 ) 0*  1*  0*  01  0*  1*  1*  11
 (K 1 ) 00  10  00  0*  00  10  10  1*
 (L 0 ) *0  10  *0  *1  *0  10  10  11
 (L 1 ) 00  *0  00  01  00  *0  10  11
 (M 0 ) *0  10  *0  *1  *0  10  10  11
 (M 1 ) 00  10  00  01  00  10  *0  *1
 (N 0 ) 00  10  00  0*  00  10  10  1*
 (N 1 ) 0*  1*  0*  01  0*  1*  1*  11
 (O 0 ) 00  *0  00  01  00  *0  *0  *1
 (O 1 ) *0  10  *0  *1  *0  10  10  11
```

Figure 22. Fault Analysis for the Multiplexer

HALF ADDER

A	B	W	P
1	0	1	0
0	1	1	0
1	1	0	1

MULTIPLEXER

a	b	c	n	o
1	0	0	0	1
1	1	0	1	0
0	0	1	0	0
0	1	1	0	1

FULL ADDER

Q	C	D	Y	Z
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1

PARTITION TEST CIRCUIT

A	B	b	c	C	D	W	X	Y	Z
1	1	0	0	0	0	0	0	1	0
1	1	1	0	1	0	0	1	1	0
1	0	0	1	0	1	1	0	1	0
1	0	0	1	1	1	1	0	0	1
0	1	1	1	1	0	1	0	0	1

Figure 23. Test Sequence for 100 Percent Fault Coverage for the Partition Test Circuit

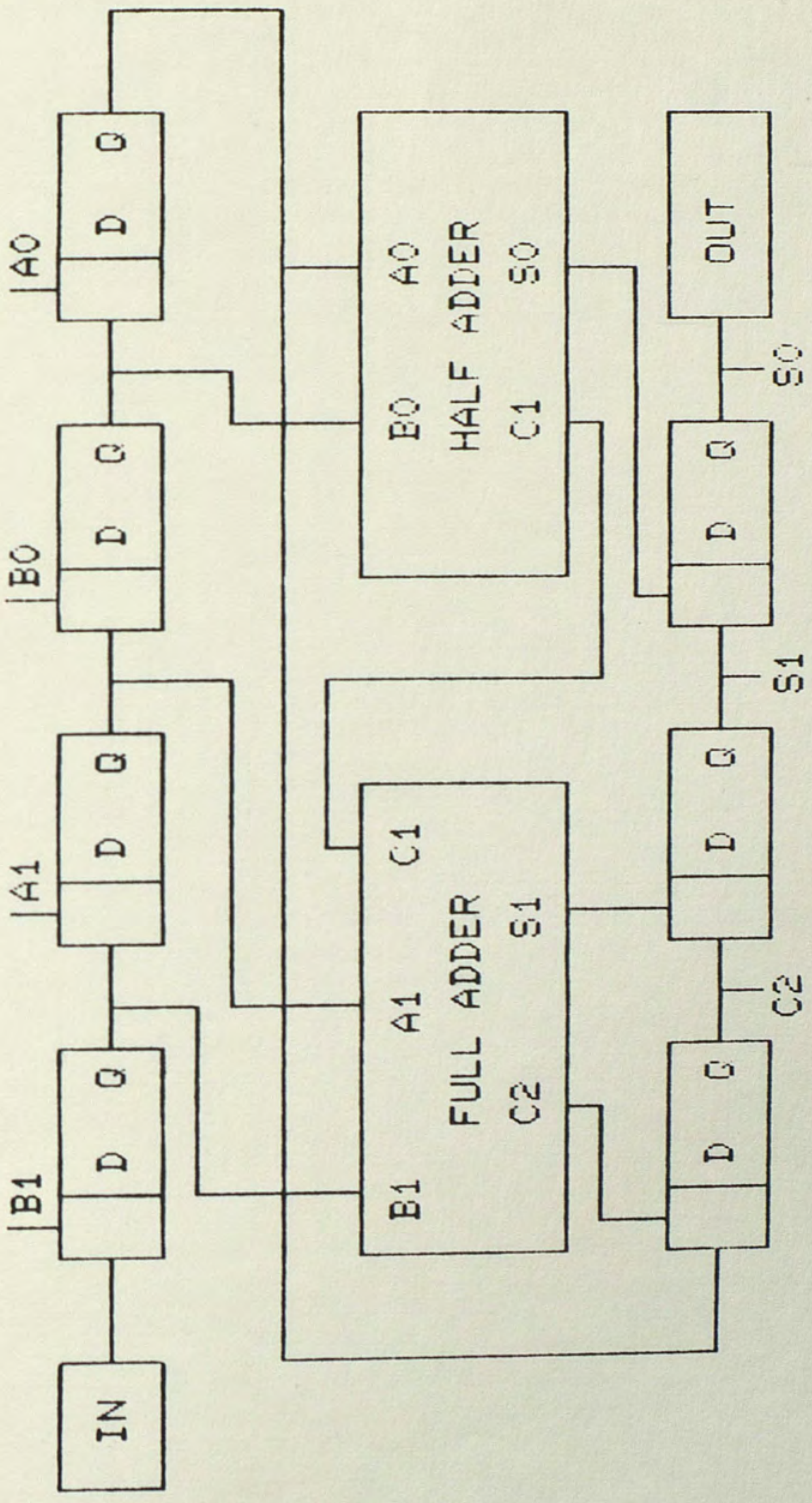


Figure 24. Scan Path Test Circuit

to shift data from the input pin, through the flip-flops, to the output pin.

The connection of the output Q4 of the input register to the serial input of the output register is crucial to the scan path methodology. This links the registers in series and provides a means of verifying the correct operation of the flip-flops prior to inserting vectors to check the combinational network. Patterns may be shifted through the register and the output checked. Having this capability is important for large combinational networks containing many registers. It allows the registers to be linked together, forming a long shift register with one input and one output.

Once the flip-flops have been checked, the adder in Figure 24 is probed by first setting the value of sigma to 1. Data are applied at the serial input port and fill the input register in four clock pulses. At the fifth clock pulse, sigma is set to 0, and the output of the adder is loaded into the output register. During the next four clock pulses, sigma is again set to 1, and the output register is shifted out.

Although test time can be very long in the scan path method because of the serial nature of the test data I/O, hardware overhead in practice is relatively small (typically less than 20 percent). So far, the technique has been applied to relatively small circuits (<4000 gates). In VLSI size chips, it may be necessary to provide a sophisticated architecture to implement the scan path. Users of this method could probably utilize the technique of major and minor loops employed in bubble memories to reduce the time needed to shift patterns in and out of the chip.

BILBO

The built-in logic block observer is the last testing method we will examine. It is based on the use of an on-chip test generator and analyzer. As such, it is well suited to VLSI, since test vectors and responses need not be sent to input-output pins, for only the signature of the test is observed externally.

The BILBO method makes use of two linear feedback shift registers. One LFSR creates a pseudorandom data pattern that is used as input to a combinational network.

The output of that network is compressed into a signature by the second LFSR. This signature can then be compared with an expected signature also stored on-chip.

Figure 25 is an example of the BILBO approach. As in the scan-path method, a control variable sigma directs the chip to enter either a self-test mode (sigma=1) or normal operation mode (sigma=0). When sigma equals 1, multiplexers at the front end of each flip-flop select serial operation. Exclusive OR gates in the serial paths implement a polynomial division of the data in the adder input and output paths. The input LFSR implements division by the primitive polynomial

$$X^4 + 1 \qquad (68)$$

to generate a pseudorandom sequence of length

$$2^4 - 1 = 15 \, . \qquad (69)$$

(Clearly the input register can never be in the all 0 state, because this state can never be excited.) The output register is similarly configured into an LFSR that implements a hashing function on the adder outputs.

The input and output registers must be preset with a "seed" value prior to beginning the testing to guarantee that the signature generated by the hashing function is produced under the same initial conditions as the stored signature. For example, if the input register is preset to all 1's and the output register is cleared, a transition through the next 14 states will, with high probability, result in

$$c_2 s_1 s_0 = 100 \qquad (70)$$

if no faults are present in either the LFSRs or the adder.

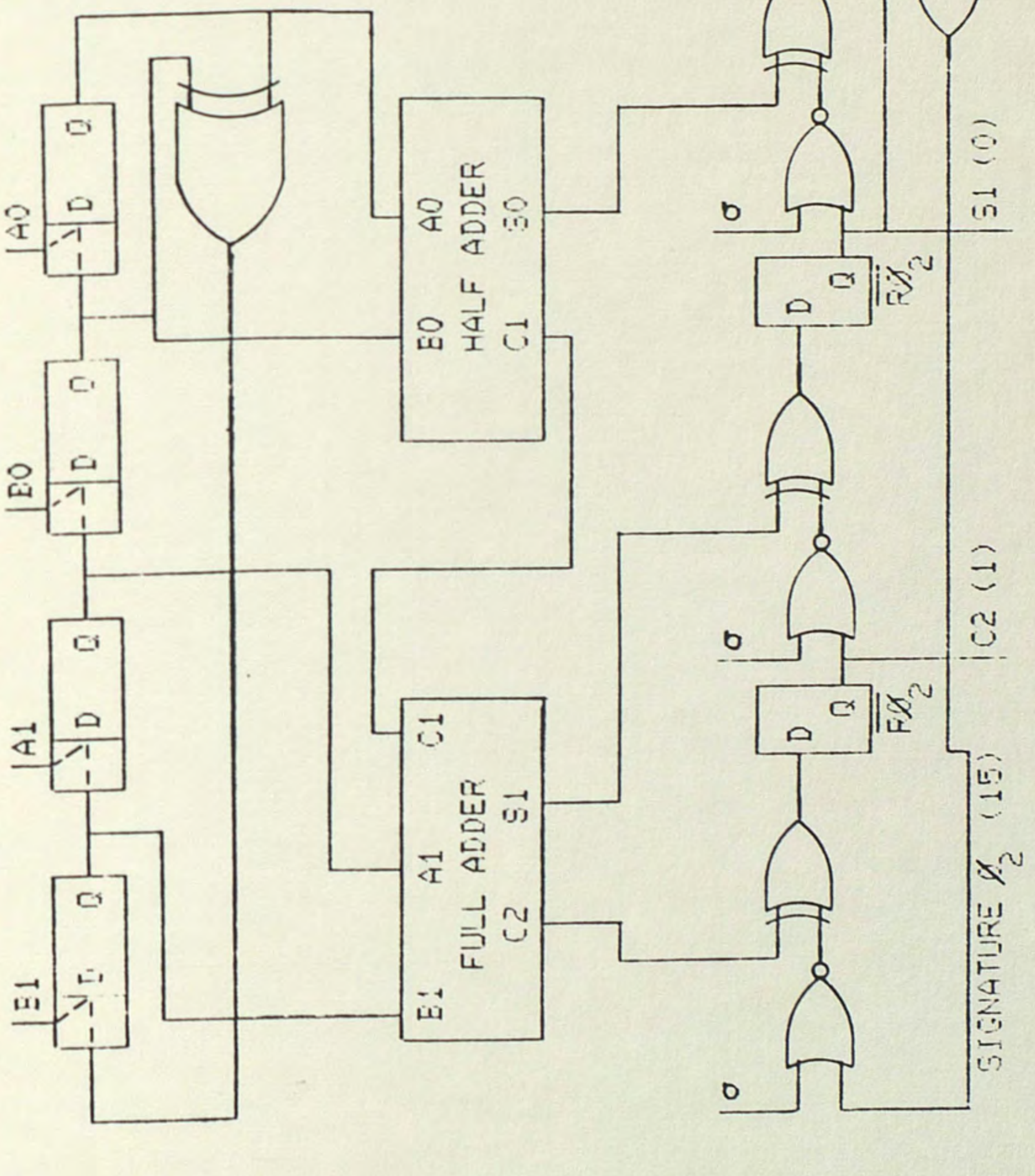


Figure 25. BILBO Test Circuit

APPENDIX C

HILDO TEST TECHNIQUE

The HILDO technique is based on a single diagnostic register which functions as the device's test generator and analyzer. The technique is easily implemented in VLSI devices because the number of test control pins is small and only the diagnostic register signature is observed externally.

The HILDO technique employs an on-board register as a LFSR. The feedback circuitry is constructed from the device circuitry. Thus, the register compresses the device circuitry's response into a characteristic signature. The signature is then compared with the expected signature which can be stored on the device.

Figure 26 is an example of the HILDO technique. A test control input (sigma/X2) directs the device to enter the self-test mode or normal operation. The register is preset to a repeatable, seed value prior to the beginning of the test. The clock is then advanced for a predetermined number of cycles. The final diagnostic register contents are compared with the expected result.

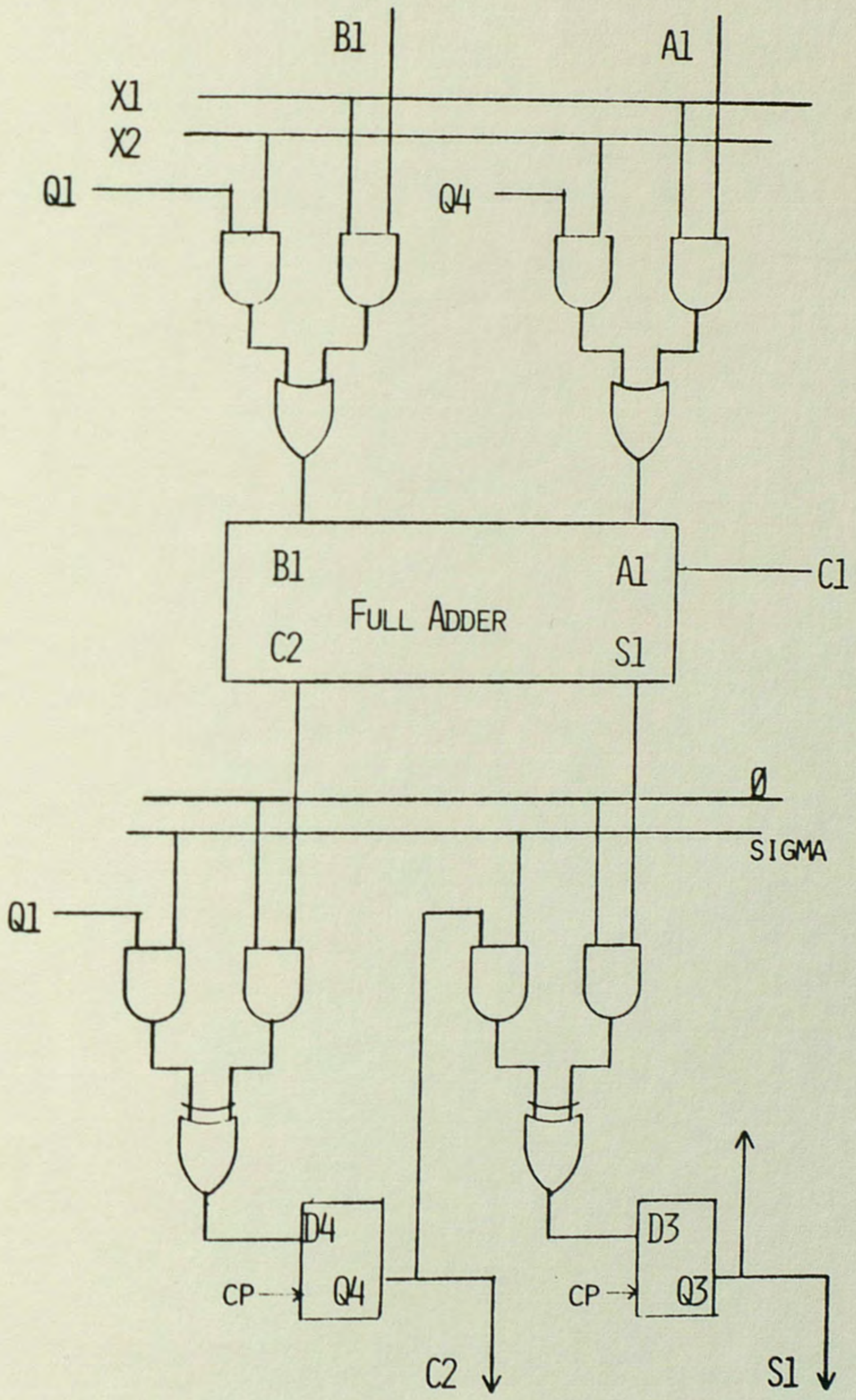


Figure 26. HILDO Test Circuit

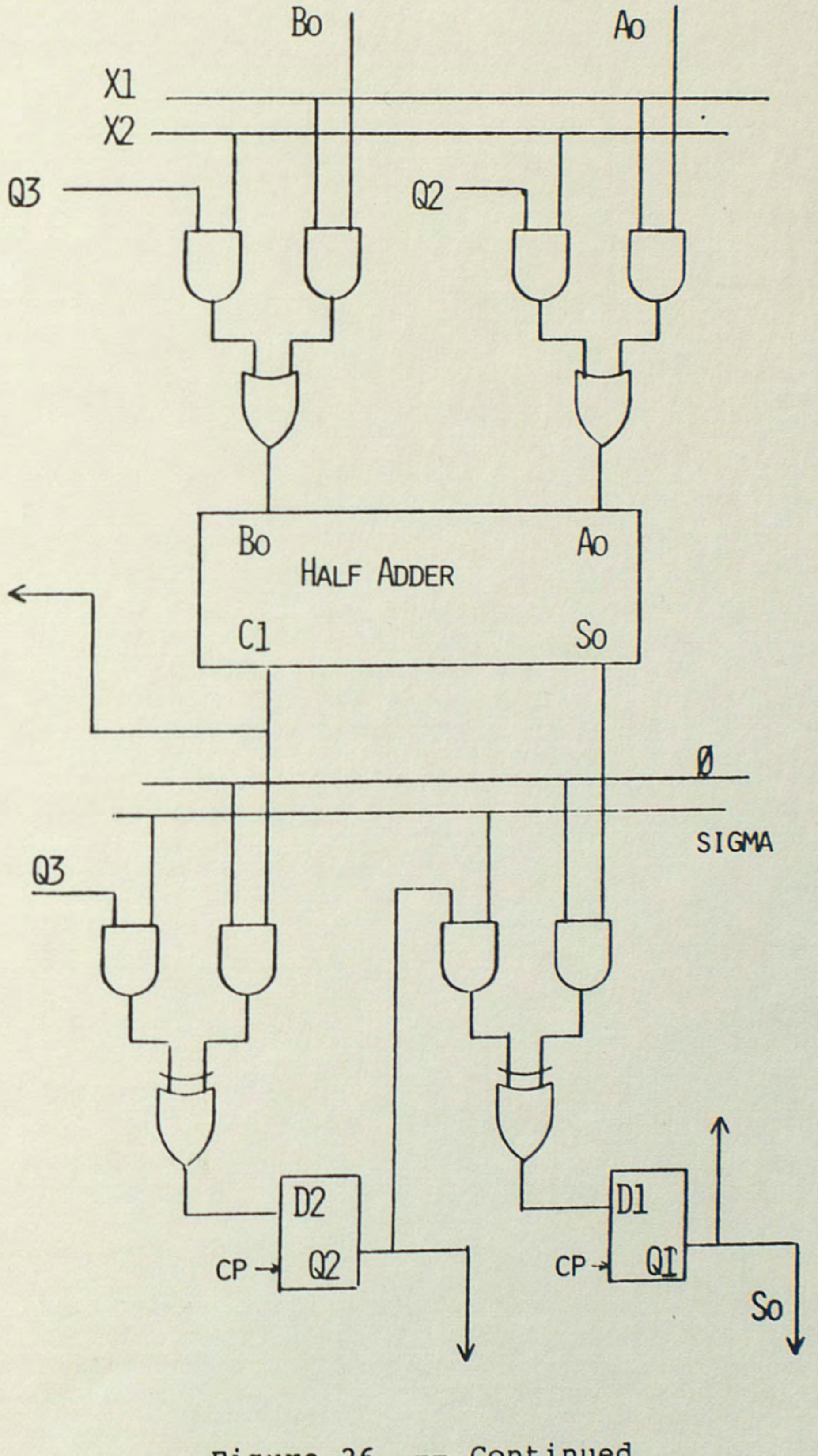


Figure 26. -- Continued

The HILDO diagnostic register states, for the two bit adder circuit of Figure 11, are shown in Table 8. The HILDO register states were generated by a simple computer program. If the device is functioning correctly, then the HILDO register will generate the sequences shown in Table 8.

All but one fault can be detected by initializing the diagnostic register to

$$Q(1,1,1,0) = Q(q_1,q_2,q_3,q_4) \tag{71}$$

where

$$Q(q_1,q_2,q_3,q_4)$$

is the output vector of the individual registers.
The fault coverage, predicted from Figure 12 is 97.6 percent. The remaining fault can be covered in a single additional test. The additional test consists of initializing the diagnostic register to

$$Q(0,0,1,1) = Q(q_1,q_2,q_3,q_4) \tag{72}$$

which will provide complete fault coverage.

The HILDO technique requires the addition of 36 gates to the device circuitry. However, the gates associated with the second register, which is required by most other techniques, can be eliminated. The second register

TABLE 8 HILDO STATE TRANSITIONS

	Register Vector				Input Vector				Adder Vector			
	Q4	Q3	Q2	Q1	I1	I2	I3	I4	A1	A2	A3	A4
The new seed is	0	0	0	0	0	0	0	0	0	0	0	0
Repeat Vector	0	0	0	0								
The new seed is	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	0	0	1	0	0	0	0	1	0	0	0	1
	0	0	0	0	0	0	0	0	0	0	0	0
Repeat Vector	0	0	0	0								
The new seed is	0	0	1	1	1	0	0	1	0	1	0	1
	1	1	0	0	0	1	1	0	0	1	0	1
Repeat Vector	0	0	1	1								
The new seed is	0	1	0	0	0	0	1	0	0	0	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
	1	1	0	0	0	1	1	0	0	1	0	1
Repeat Vector	0	0	1	1								
The new seed is	0	1	0	1	1	0	1	0	0	1	0	1
	1	1	1	1	1	1	1	1	1	1	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	0	1	1	0	0	0	1	1	0	1	1	0
	0	1	0	1	1	0	1	0	0	1	0	1
	1	1	1	1	1	1	1	1	1	1	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								

TABLE 8 -- CONTINUED

	Register	Vector			Input		Vector		Adder		Vector	
	Q4	Q3	Q2	Q1	I1	I2	I3	I4	A1	A2	A3	A4
The new seed is	0	1	1	1	1	0	1	1	1	0	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	1	0	0	0	0	1	0	0	0	1	0	0
	0	0	0	0	0	0	0	0	0	0	0	0
Repeat Vector	0	0	0	0								
The new seed is	1	0	0	1	1	1	0	0	1	0	0	0
	0	1	0	0	0	0	1	0	0	0	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
	1	1	0	0	0	1	1	0	0	1	0	1
Repeat Vector	0	0	1	1								
The new seed is	1	0	1	0	0	1	0	1	0	1	0	1
	0	0	0	0	0	0	0	0	0	0	0	0
Repeat Vector	0	0	0	0								
The new seed is	1	0	1	1	1	1	0	1	1	0	0	1
	0	1	0	0	0	0	1	0	0	0	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
	1	1	0	0	0	1	1	0	0	1	0	1
Repeat Vector	0	0	1	1								

TABLE 8 -- CONTINUED

	Register Vector				Input Vector				Adder Vector			
	Q4	Q3	Q2	Q1	I1	I2	I3	I4	A1	A2	A3	A4
The new seed is	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	1	1	0	1	1	1	1	0	1	0	0	1
	0	1	1	1	1	0	1	1	1	0	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	1	1	1	0	0	1	1	1	1	0	1	0
	1	1	0	1	1	1	1	0	1	0	0	1
	0	1	1	1	1	0	1	1	1	0	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								
The new seed is	1	1	1	1	1	1	1	1	1	1	1	0
	0	0	0	1	1	0	0	0	0	1	0	0
	1	1	0	0	0	1	1	0	0	1	0	1
	0	0	1	1	1	0	0	1	0	1	0	1
Repeat Vector	1	1	0	0								

represents approximately 20 gates. If the register gate reduction is included, the overall gate count of the adder circuit employing the HILDO technique is 16 gates. The 16 gates represent approximately 48 additional transistors.

LIST OF REFERENCES

Agrawal, V.D and Mercer, M.R. 1982a. Testability measures - What do they tell us? IEEE International Test Conference Proceedings. p. 391-396. Los Angeles: IEEE Press.

Agrawal, V.D, Seth, S.C, and Agrawal, P. 1982b. Fault coverage requirement in production testing of LSI circuits. IEEE Journal of Solid-State Circuits. SC-17: 57-61.

Anderson, D.A. and Metze, G. 1973. Design of totally self-checking check circuits for m-out-of-n codes. IEEE Transactions of Computers. C-22: 263-269.

Barbacci, M.R. 1981. The symbolic manipulation of computer descriptions: An ISPS simulator, p. 4-32. Pittsburgh: Carnegie-Mellon University.

Barzilai, Z. and Rosen, B.K. 1983. Comparison of AC self-testing procedures. IEEE International Test Conference Proceedings. p. 89-94. Los Angeles: IEEE Press.

Beh, C.C., Arya, K.H., Radke, C.E., and Torku, K.E. 1982. Do stuck fault models reflect manufacturing defects? IEEE International Test Conference Proceedings. p. 35-42. Los Angeles: IEEE Press.

Beucler, F.P. and Manner, M.J. 1984. HILDO: The highly integrated logic device observer. VLSI Design. V: 88-96.

Bhavsar, D.K. and Heckelman, R.W. 1981. Self-testing by polynomial division. IEEE International Test Conference Proceedings. p. 208-216. Los Angeles: IEEE Press.

Breuer, M.A. and Friedman, A.D. 1976. Diagnosis and reliable design of digital systems, p. 56-59; 174,175; 224-273. Rockville, Maryland: Computer Science Press.

Buehler, Martin G. and Sievers Michael W. 1982. Off-line, built-in test techniques for VLSI circuits. IEEE Transactions on Computers. C-31: 69-82.

Chang, H.Y, Manning, E., and Metze, G. 1974. Fault diagnosis of digital systems, New York: John Wiley and Sons, 1970; reprint ed., p. 49-54. New York: Krieger.

Eichelberger, E.B. and Williams, T.W. 1978. A logic design structure for LSI testability. Journal of Design Automation Fault Tolerant Computing 2: 165-178.

Everett, C. 1984. Testability analysis and fault simulation spearhead CAT's entry into CAE environment. Electronic Design News, 29: 107-117.

Flomenhoft, M.J., Si, S.C., and Susskind, A.K. 1973. Algebraic techniques for finding tests for several fault types. IEEE Fault Tolerant Computing Conference Proceedings. p. 227-235. Los Angeles: IEEE Press.

Freund, John E. 1971. Mathematical statistics, 2nd ed., p. 266-270. New Jersey: Prentice-Hall Inc.

Frohwerk, R. 1977. Signature analysis: A new digital field service method. Hewlett-Packard Journal, 28: 2-8.

Galiey, J., Crouzet, Y., and Vergniault, M. 1980. Physical versus logical fault models MOS LSI circuits: Impact on their testability. IEEE Transactions on Computers, C-29: 527-531.

General Electric 1981. "Draft Standard 1981". (Typewritten.) p. 1-20.

Hess, R.D. 1982. Testability analysis: an alternative to structured design for testability. VLSI Design III: 22-29.

Hill, F.J. and Peterson, G.R. 1978. Digital systems: hardware organization and design, 2nd ed., p. 131-145. New York: John Wiley and Sons.

Hnatek, E. 1984. A merger of CAD and CAT is breaking the VLSI bottleneck. Electronics, 57: 129-134.

Kidder, T. 1981. The soul of a new machine, p. 136-138, 258-259. New York: Avon.

Kirkland, T. and Flores, V. 1983. Software checks testability and generates tests of VLSI design. Electronics, 56: 120-124.

Konemann, B., Mucha, J., and Zwiehoff, G. 1980. Built-in test for complex digital integrated circuits. IEEE Journal of Solid-State Circuits, SC-15: 315-319.

Levendel, Y.H. and Menon, P.R. 1982. Test generation algorithms for computer hardware description languages. IEEE Transactions on Computers, C-31: 577-588.

Long, S.I. and Eisen, F. H. 1980. Ion implanted GaAs I.C. process technology. Rockwell International Quarterly Technical Report 11. 22:7-16.

McClusky, E.J. and Bozorigui-Nesbat, S. 1980. Design for Autonomous Test. IEEE International Test Conference Proceedings. p. 15-21. Los Angeles: IEEE Press.

Middleton, T. 1983. Functional test vector generation for digital LSI/VLSI devices. IEEE International Test Conference Proceedings. p. 682-691. Los Angeles: IEEE Press.

Muehldorf, E.I. and Savkar, A.D. 1981. LSI logic testing - an overview. IEEE Transactions on Computers, C-30: 1-17.

Rappaport, A. 1984. Simple functional tester verifies chip performance. Electronic Design News, 29: 151-162.

Rappaport, A. 1983. Hands-on timing verifier validates IC design techniques. Electronic Design News, 28: 147-162.

Reddy, S.M, Reddy, M.K., and Kuhl, J.G. 1983. On testable design for CMOS logic circuits. IEEE International Test Conference Proceedings. p. 435-445. Los Angeles: IEEE Press.

Rose, C.W., Ordy, G.W., and Drongowski, P.J. 1984. N.mPc: A study in university-industry technology transfer. IEEE Design and Test, 1: 44-55.

Roth, P. 1980. Computer logic, testing, and verification, p. 59-71. Rockville, Maryland: Computer Science Press.

Son, K. and Fong, J.Y.O, 1982. Automatic behavioral test generation. IEEE International Test Conference Proceedings. p. 161-165. Los Angeles: IEEE Press.

Susskind, Alfred K., 1981. Testing by verifying Walsh coefficients. IEEE International Test Conference Proceedings. p. 206-209. Los Angeles: IEEE Press.

Susskind, Alfred K., 1973. Diagnostics for logic networks. IEEE Spectrum. 10: 40-47.

Werner, J., and Beresford, R. 1984a. A system engineer's guide to simulators. VLSI Design V: 27-49.

Werner, J. ed., 1984b. CAE systems: A status report. VLSI Design V: 40-49.

Werner, J. ed., 1984c. Physical model for logic simulations. VLSI Design V: 62-67.

Williams, T.W. and Parker, K.P. 1982. Design for testability - a survey. IEEE Transactions on Computers, C-31: 2-15.