City Research Online

City St George's, University of London

Citation: Lala, P. K. (1976). The Diagnosis of Solid and Intermittent Faults in Logic Circuits. (Unpublished Doctoral thesis, The City University)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/37937/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

THE DIAGNOSIS OF SOLID AND INTERMITTENT FAULTS IN LOGIC CIRCUITS

PARAG KUMAR LALA MSc

Thesis Submitted For The Degree Of
Doctor of Philosophy

THE CITY UNIVERSITY
1976

Tennessee Williams

CONTENTS

Page No.

ABSTRACT v

1. INTRODUCTION 1

1.1 Automated Design and Analysis of Digital Systems 3

1.1.1 Evolution of Design Automation 3

1.1.2 Design Automation of Digital Systems 4

1.2 Reliability in System Design 7

1.2.1 The Importance of Reliability 7

1.2.2 Failures in a Digital System 11

1.3 Redundancy Applications in Digital System Design 15

1.3.1 What is Redundancy? 15

1.3.2 Static Redundancy 15

1.3.3 Dynamic Redundancy 18

1.4 Maintainability of a Digital System 19

1.4.1 The Crucial Role of Fault-diagnosis 19

1.4.2 Non-classical Faults in Digital Systems 21

1.4.3 Intermittent Faults 23

2. DIGITAL LOGIC SIMULATION 25

2.1 General Review of Logic Simulation 25

2.2 Timing Model in Logic Simulation 29

2.3 Coding of the Circuit Structure for Simulation 30

2.3.1 Combinational circuit model 30

2.3.2 Development of sub-routines for function evaluation 33

2.3.3 Synchronous sequential circuit model 35

Page No.

2.4 Fault Simulation in Logic Circuits 43

2.4.1 Techniques for fault simulation 43

2.4.2 Fault insertion 44

3. DIAGNOSIS OF SOLID FAULTS IN COMBINATIONAL LOGIC CIRCUITS 49

3.1 Introduction 49

3.2 Test Generation for Diagnosing Solid Faults in Combinational Logic Circuits 49

3.2.1 Truth table method 49

3.2.2 One-dimensional path sensitization 50

3.2.3 D-Algorithm 52

3.2.4 Equivalent normal form method 57

3.2.5 Boolean difference 60

3.3 Fault Indistinguishability in Combinational Circuits 61

3.3.1 Use of fault-folding in deriving an indistinguishable fault set 62

3.4 Multiple Faults in Combinational Logic Circuits 64

3.5 An Algorithm for Deriving a Fault Detection Test Set for Combinational Circuits 67

3.5.1 Application of the algorithm to generate a fault-detection test set for a combinational logic network 69

3.5.2 Computer simulation of the algorithm 78

3.6 Application of the Fault Detection Test Set in Locating Multiple Faults in Combinational Circuits 110

3.6.1 Programme development for the faulted and fault-free simulation of combinational logic circuits 112

3.6.2 A circuit example 114

4. DIAGNOSIS OF SOLID FAULTS IN SEQUENTIAL LOGIC CIRCUITS 118

4.1 Introduction 118

4.1.1 Circuit testing approach 118

Page No.

4.1.2 Machine identification approach 119

4.1.3 Notations and definitions 121

4.2 Checking Experiments for Sequential Circuits 124

4.2.1 A computer-aided design procedure for homing experiments 124

4.2.2 Programme development for automatic transfer sequence generation 135

4.3 A Method for Test Sequence Generation of Sequential Circuits based on Behaviour and Structure 136

4.3.1 Application of fault-folding to a synchronous sequential circuit 136

4.3.2 An algorithm for checking sequence generation of synchronous sequential circuits 138

4.3.3 An example of the application of the algorithm 141

4.3.4 Computer simulation of the checking sequence generating algorithm 146

4.3.5 Computer-aided generation of a test sequence: a circuit example 149

4.4 Fault Location in Synchronous Sequential Circuits 154

4.4.1 Programme development for fault simulation in synchronous sequential circuits 154

4.4.2 A Procedure for Locating Faults in Synchronous Sequential Circuits 161

4.4.3 A circuit example 165

4.4.4 Computer simulation of the fault-locating procedure with an example 169

4.5 Fault-folding Techniques in Asynchronous Sequential Circuit Testing 177

4.5.1 Test generation 177

4.5.2 A circuit example 179

5. INTERMITTENT FAULT DETECTION IN LOGIC CIRCUITS 184

5.1 A Probabilistic Model of Intermittent Fault Ocurrence 184

Page No.

5.1.1 Intermittent fault model 184

5.1.2 Simulation of intermittent faults 186

5.2 Poisson Distribution in Intermittent Fault Detection 188

5.2.1 The Poisson distribution 188

5.2.2 Detection of intermittent faults 189

5.2.3 Programme development for the evaluation of test-time from a given probability of failure 190

5.2.4 Example of the application of the Poisson distribution in intermittent fault detection 192

5.3 Detection of Intermittent Faults in Combinational Circuits 196

5.3.1 Detection procedure 196

5.3.2 A circuit example 198

5.4 Detection of Intermittent Faults in Synchronous Sequential Circuits 201

5.4.1 Test sequence generation 201

5.4.2 Detection procedure 204

6. PROJECT DISCUSSION AND CONCLUSION 216

6.1 Diagnosis of Solid Faults 216

6.1.1 Combinational circuits 216

6.1.2 Sequential circuits 219

6.2 Diagnosis of Intermittent Faults 221

6.3 Conclusion 223

APPENDIX A. Random events in time : Poisson distribution 228

REFERENCES 232

ACKNOWLEDGEMENTS 241

ABSTRACT

This thesis is concerned with the test generating procedures for logic circuits using a "black-box" approach. By a black-box approach it is meant that the internal probing of a circuit is not permitted.

The project is divided into two phases : development of new procedures for diagnosing permanent faults and the use of these procedures for detecting intermittent faults. The basis of the procedures is the 'Fault-folding' concept, which is used to form fault equivalence classes for a given combinational circuit by "folding" faults towards the primary inputs.

An algorithm is presented for finding a minimal set of tests for the detection and location of single and multiple faults in a combinational logic network.

Synchronous sequential circuits in a particular state behave like a combinational circuit with feedback paths acting as primary inputs. Fault-folding is used to find the minimum number of states at which the circuit is to be tested, to test cover all faults in the circuit. A checking sequence is then developed to test the circuit at the required states, thus taking into account both behaviour and structure of the network. A fault-location procedure is suggested which locates faults within an equivalent class.

Fault detection procedures for asynchronous sequential circuits are also described. Asynchronous circuits are transformed into synchronous circuits and tests are generated for these versions of the circuits.

Finally a procedure, based on probability theory, is developed to detect a well-behaved intermittent fault in a logic network. The

procedure employs repeated applications of a test or a test sequence. The time-period, during which a test or a test sequence is repeatedly applied, depends on the probability of detection desired and is derived from the Poisson distribution.

Algorithms described in the thesis are computer-programmed using FORTRAN IV; computational results are reported.

The programme used for automatic test generation of combinational circuits can handle circuits having up to 15 primary inputs, 10 primary outputs and 95 basic gates. The corresponding programme for synchronous sequential circuits is applicable to circuits having up to 50 gates and 16 internal states.

The main objective in developing the programmes in the thesis, is to prove that the algorithms described are suitable for computer-programming; no particular emphasis has been placed on the efficiency of the resulting software.

1. INTRODUCTION

Over the past 25 years, electronics has progressed from the era of thermionic valves to discrete solid state devices and from thence to the widespread use of integrated circuits. As can be seen from Fig.1.1 this has improved the overall reliability of electronic equipment for a given size by several orders of magnitude.

However by a process akin to one of Parkinson's laws, the complexity of electronic circuits has increased such that the overall reliability has not improved to the same extent.

Thus the diagnosis and location of faults in integrated circuit chips, particularly M.S.I. and L.S.I. becomes increasingly important, particularly as access can be obtained only to input and output connections.

In this thesis a study has been made of methods for solid and intermittent fault diagnosis and location in combinational and sequential logic circuits. Computer simulation techniques have been extensively utilised.

It commences with a section on design automation and reliability studies. This is followed by a discussion of digital logic simulation.

Chapter three deals with the diagnosis of solid faults in combinational logic circuits.

The next chapter is on fault diagnosis in sequential logic circuits.

Many faults occur only for a part of the total operating time i.e. they are intermittent. The location and diagnosis of such faults in both combinational and sequential circuits is considered in chapter five.

The final chapter is a discussion and conclusion of the whole topic.

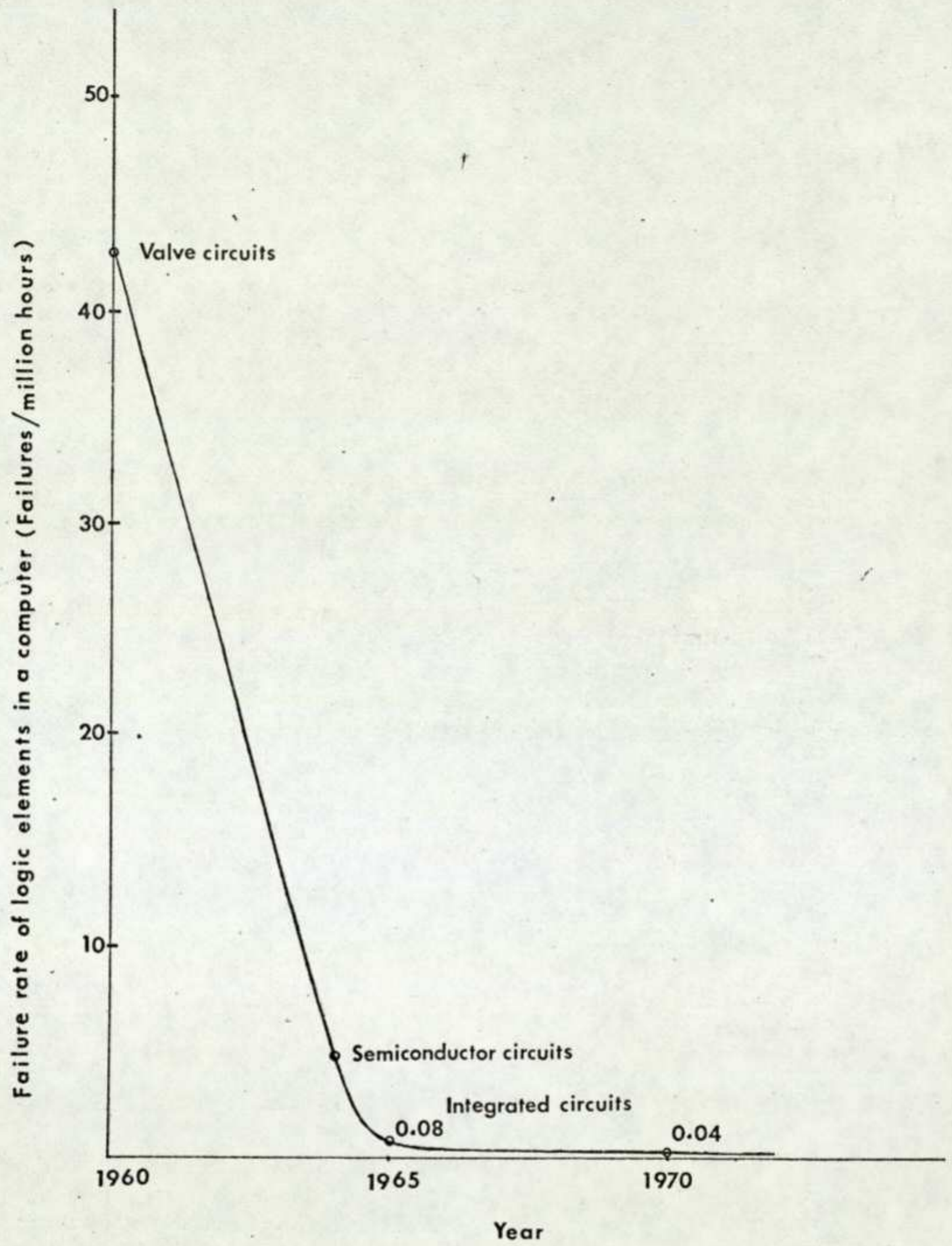


Fig 1.1 Improvement in reliability with development of integrated circuits

1.1 Automated Design and Analysis of Digital Systems

1.1.1 Evolution of Design Automation

The difficult and tedious job of designing and analysing complex digital systems has led to the increasing application of computers in these areas. This is normally termed 'Design Automation'.

The evolution of Design Automation is seen in many references (1) (2) (3).

The strategic goals of a Design Automation system are:-

(i) Possible reduction in time and cost between the start of design and completion of fabrication.

(ii) Improvement in documentation, checking and analysis capabilities.

(iii) Symbiosis of man and computer as a means of obtaining better solutions than either one can produce alone.

The first use of "Design Automation' was made during the design of the second generation of computers by defining the logical design, as a separate and distinct preliminary step, followed by the details of the physical design. Early examples of design automation from each of these areas are,

(1) the logic design file development

(2) the back panel wiring development.

In recent years the field has grown considerably and has made large scale integrated (LSI) electronics feasible for commercial productions. With the introduction of LSI, the cost of digital logic has come down to a level which was unthinkable in the days of wired discrete components. But LSI has also introduced difficult new problems such as design verification and part testing which without Design Automation would exceed the permissible product development times and costs. It is for this reason that most manufacturers have developed quite sophisticated Design Automation systems.

1.1.2 Design Automation of Digital Systems

Design Automation commences with system design which involves defining the machine's operational specifications in the form of cost, size, reliability, ease of maintenance, and operator-machine interface. This is the phase in which the user's requirements for a digital system are first translated into the technical language of the engineer.

During the next stage the proposed system is divided into various logical blocks; the next task is to prove, by using simulation systems like GPSS (4) and SIMSCRIPT (5), that all of the logical blocks work together harmoniously.

The logical blocks are then sub-divided into smaller blocks and the design specification is evaluated in terms of the sub-block logical functions using register transfer language. Each designer or design team is then assigned with the responsibility of physical design and implementation of a sub-block.

The next phase of the Design Automation is to turn the defined function of each sub-block into solid design; this can be done by converting the register transfer description of the sub-block, into Boolean equations or logic diagrams using synthesizing routines (6) (7).

Once a solid design has been obtained, the simulation function of the Design Automation can be used to test out the logic circuitry of the sub-blocks. Simulation techniques have become an important design automation tool. Simulators become indispensible where increased assurance of design correctness before the manufacturing is an economic necessity. Despite the growing importance of higher level functional representation of digital systems, most of the simulation for Design Automation is done using logic gates as the basic elements. The main reason for using gate level simulators is to achieve fast computation

and they are extremely valuable in digital system design.

The next process in the design is to transfer the logic design from a schematic to realizable hardware. At this stage the partitioning problem arises, which consists of breaking up an entire logic schematic into smaller subsets of elements according to some packaging rules. Depending on the hardware sub-system level, these subsets correspond to IC chips and printed circuit boards. The objectives of partitioning include: limiting the number of terminals and interconnections, using as many standard parts as possible, and satisfying signal propagation requirements among other things.

On the circuit board itself, the designer faces the problem of properly placing the components. Usually the objective is to minimize the total interconnecting distance between them in order to ease the problem of completing the wire-routing and also to help satisfy other constraints such as signal delay and cross-talk. Once the components are placed, they must be properly interconnected. This is the problem of routing. Over the last few years many successful automated routing systems have been developed; however these techniques worked satisfactorily for relatively simple and regular layouts only. As components and devices were packed more closely, fewer automatic routing procedures were able to reach 100 percent completion. Experience is proving that a designer can rapidly complete routing procedures that would be very expensive to finish using complete automatic techniques.

A comprehensive review of the current theory and practice in the areas of partitioning, placement and routing can be found in (8).

Fault-diagnosis has become a separate and very important area of Design Automation. Despite the fact that digital circuits have become increasingly reliable and cheaper, there is still a growing interest in techniques for detecting and locating failures in complex digital net-

works. One of the reasons for the increased emphasis on diagnostics result from the formidable problem faced during the production testing of LSI circuits. As a result of the system design and partitioning objectives, LSI circuits have a much lower pin to gate ratio than a design of equivalent logic complexity that uses first generation ICs or MSI circuits (9). Besides the problem of testing of logic boards has become more complicated because the tendency nowadays is to design boards with increased logical functions which in turn need thousands of gates and flip-flops to realize.

Two basic approaches are used in testing digital systems: functional and structural (10). Functional tests determine whether the unit under test performs the desired logical function and the structural tests are designed to assure that the individual components (AND, OR etc.) are operating correctly. Functional testing never became very popular because it is based on heuristics and usually did not yield a complete test of a given network. On the other hand structural tests are particularly suitable for acceptance testing before the major units of the system are assembled, and for off-line repair. In this thesis only the structural testing will be of concern.

All the functions performed by Design Automation techniques are centred around a central data base. The function of a data base is to control and disburse design data in an organised fashion. Design Automation processes usually start with separate programmes for placement, routing, artwork generation etc.; each having its own, unique data file. As the need for interchanging data increases, programmes are written to map all or part of the data from one certain file into another and a data file network develops. Later, a main data base serves as the vehicle for interchanging between data-files. Finally in the ultimate system, the programmes themselves operate directly from the single,

central data base. A central-data based design automation system (3) is shown in Fig.1.2, in which the various application programmes (rectangular boxes) communicate with each other, as necessary, via the central data base.

1.2 Reliability in System Design

1.2.1 The Importance of Reliability

Present day electronic devices and systems are becoming more and more sophisticated and complex. As complexity and sophistication increase, so do performance requirements, reliability requirements and above all, costs. A fundamental problem in estimating reliability may be described as follows: one has a complex device or system such as a space-craft, computer or missile at some stage of design and development and is concerned whether or not it will function in a given environment, in a prescribed manner, and for a given period of time. Whether the system operates successfully or not depends, of course, on a very large number of factors, some of which are under the control of the developer and others of which are not. Among these, one can include such factors as the design itself, parts and components used, the manufacturing process, the environment and human factors. Performance of a given system, under given conditions, for a given period of time, can be considered as a chance event - namely, the outcome of the event is undetermined until it has actually occurred. Hence, if one wishes to make any statements prior to the occurrence of this event, they will have to be probabilistic in nature - and it becomes natural to consider the reliability of a system as an unknown design parameter which is considered as the probability that a given system will function under specified conditions for a specified period of time. If one thinks of R(t), sometimes called the 'reliability function', as the probability

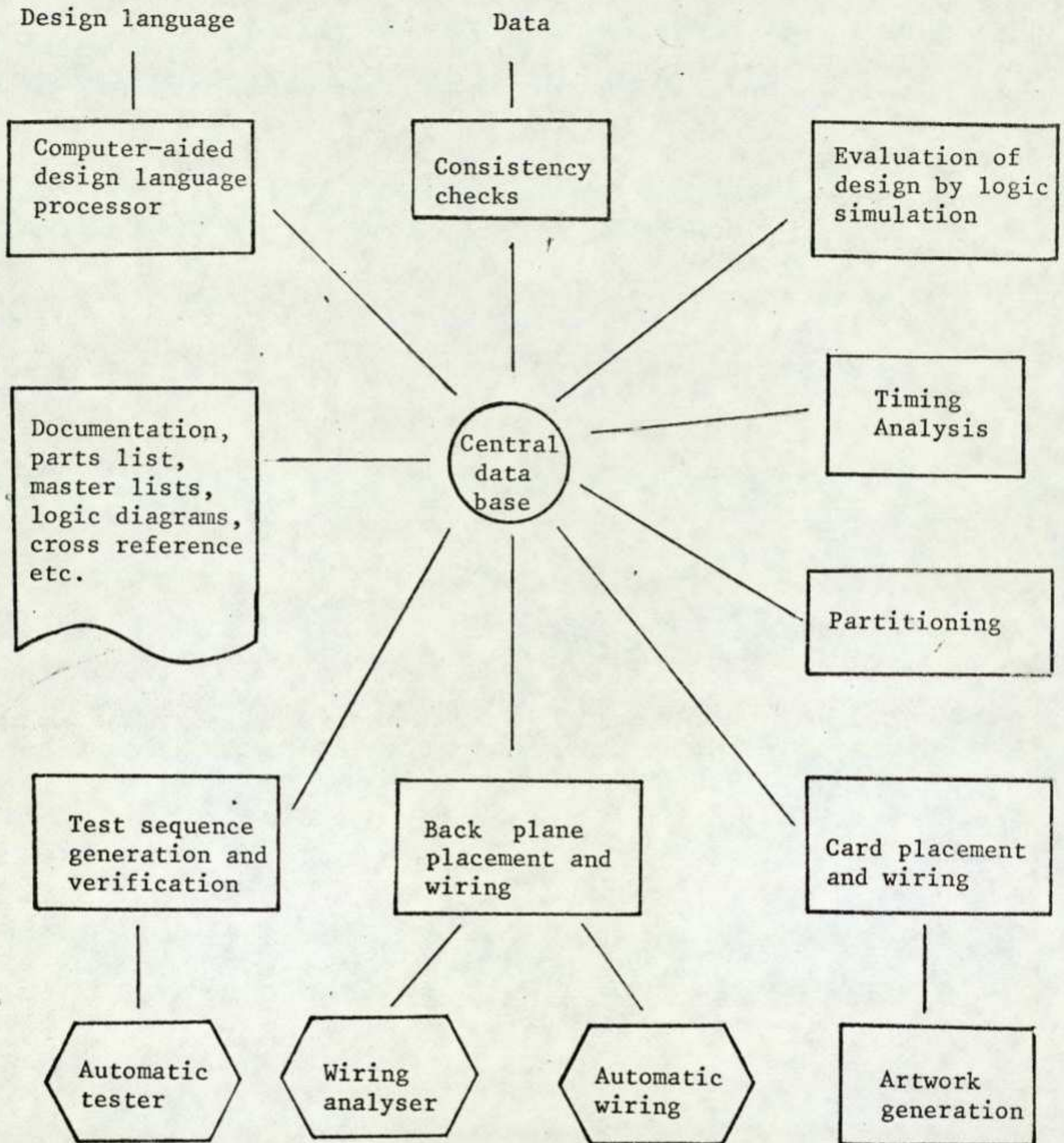


Fig.1.2 A Central-data based Design Automation System

that a certain system operates satisfactorily in the interval from 0 to t then, $F(t) = 1-R(t)$ is the probability that the system will fail to operate satisfactorily in the interval from 0 to t. The probability density function for a random variable T which measures times to failure for a certain system, is given by

$$f(t;a) = ae^{-at}, \ a>0, \ t>0$$
$$= 0 \quad \text{elsewhere.}$$

The expression f(t;a) is called the 'exponential distribution' and the random variable T having this distribution is called the 'exponential variable', each with parameter 'a'. The parameter 'a' when used in connection with reliability is called the 'Failure rate', defined as the percentage of failures in a given time. It can be shown that the mean of the distribution of T, $E(T) = 1/a$. In application to reliability problems, 1/a is often called the 'Mean time between failures' (MTBF); the average time a system will run between failures. Since

$$F(t) = \int_0^t ae^{-ax}\,dx = 1-e^{-at}, \ a>0, \ t>0$$

it follows that,

$$R(t) = 1-F(t) = e^{-at} \quad a>0, \ t>0$$

or using m=1/a, the alternative expression for the reliability function is,

$$R(t) = e^{-t/m}.$$

As t increases, R(t) decreases (Fig.1.3) and when t=m, the MTBF, the reliability is only 36.8%.

The reliability of a digital system may be improved by several ways. Some of these are

(i) The system and the circuits should be as simple as possible,

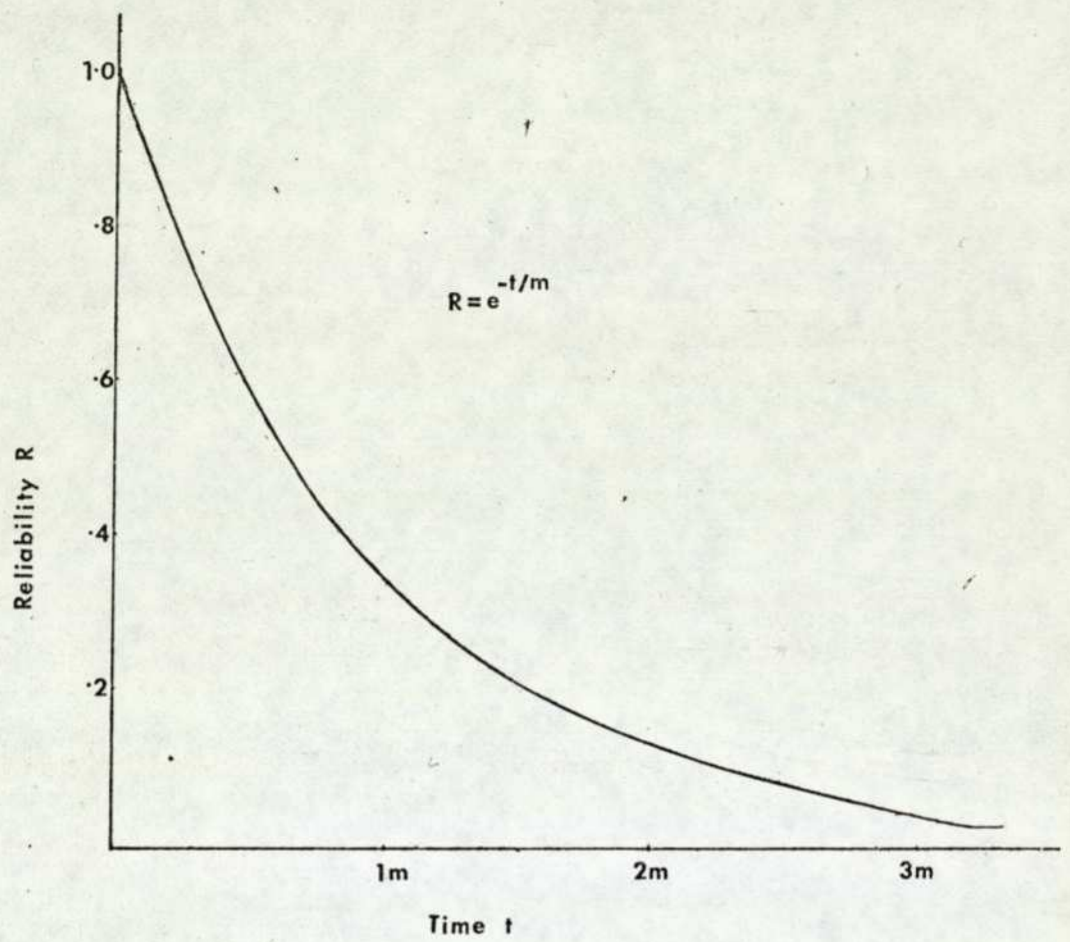


Fig 1.3 Reliability curve

consistent with achieving the design objectives.

(ii) Component reliability should be very high. As an example, consider an equipment containing N_i non-redundant components each having a failure rate λ_i. Then the overall failure rate

$\lambda_{ov} = N_i\lambda_i$ ..... (1.1).

Suppose that N_i = 1000 and an overall failure rate of 1%/1000 hours can be tolerated. Then it can be seen from equation (1.1) that λ_i = .001%/1000 hours, which is very difficult to realize. In Fig.1.4, the overall failure rate for a circuit using discrete components is .325%/1000h. The same circuit when designed as an integrated circuit has an overall failure rate of .01%/1000h.

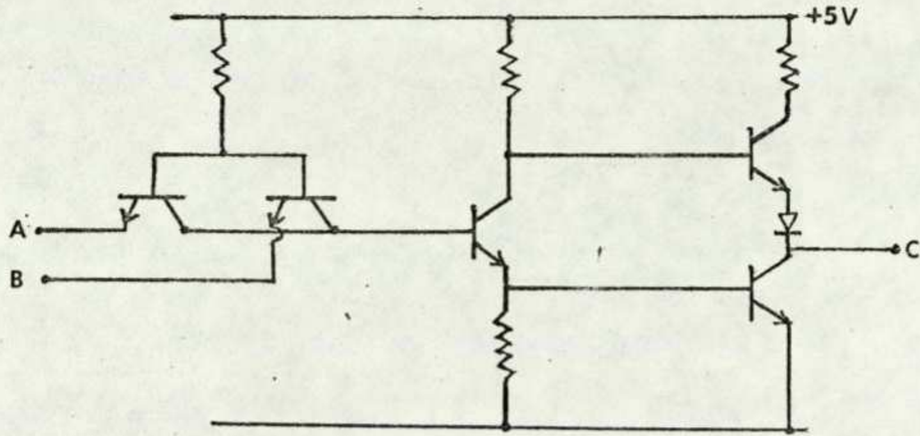
(iii) Components should be subjected to environmental tests which the system itself is likely to endure.

(iv) The interconnection between the IC chips should be kept to a minimum because in most integrated circuit systems, failures occur at the interconnections.

(v) The system should be packaged to screen out the expected forms of external interference.

1.2.2 Failures in a Digital System

It is almost certain that in a complex system, even though it was designed to achieve the highest reliability, any arbitrary number of components will fail before the end of the life time of the system causing it to malfunction. In a recent survey of component reliability the following pattern was observed - capacitors are more susceptible to failures than resistors, discrete semiconductors are less reliable than ICs; CMOS and ECL logic families are more vulnerable to failures than standard TTL, and both large-scale and medium-scale ICs can have very high failure rates (11).



	λ %/1000 hrs.	
Transistor	.008	× 5
Diode	.005	× 1
Resistor	.02	× 4
Connection	.101	× 20

$$\lambda = \Sigma N_k \cdot \lambda_k = .325\,\%/1000\,\text{hrs}$$

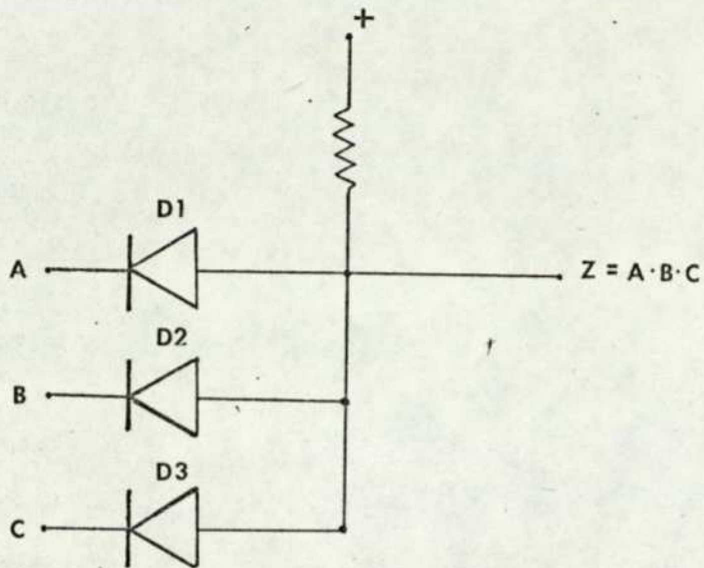
Fig 1.4 NAND gate

Component failures in digital systems may affect voltages, currents, pulse shape or circuit delays, or appear as logic faults. A logic fault occurs when logic values at one or more points in the circuit become opposite to the specified values. Logic faults can be classified into two groups; (a) solid and (b) intermittent.

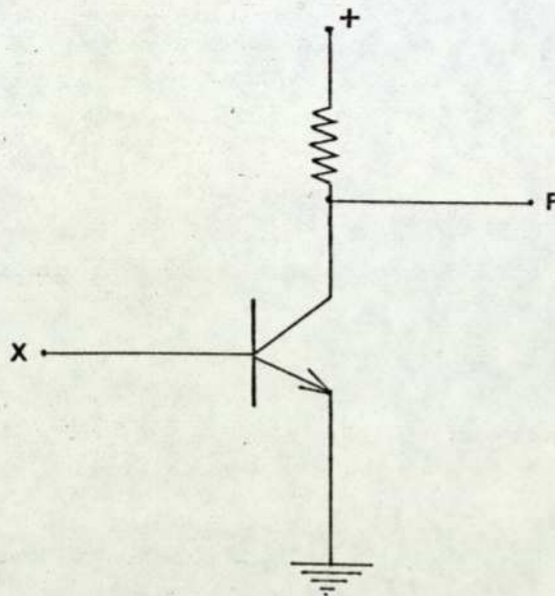
Solid faults are caused by permanent component failure and make the logic value at a certain point remain constant. On the other hand, intermittent faults which exist during certain time intervals and disappear in others result from the marginal operation of the component. The main causes of marginal operation are critical circuit timing, close design tolerances, irregular physical structure of components, interferences like irregular power supply, stray electromagnetic radiation and severe environments (12) (13) (14).

To illustrate the effect of a permanent component failure, consider the 3-input AND gate circuit shown in Fig.1.5a. If the circuit is fault-free, Z = A.B.C. However when D1 is opened, the function becomes Z = B.C and it seems as though the input A was absent. In other words, input A will appear to be permanently stuck-at-one (s-a-1). Similarly in the INVERTER circuit of Fig.1.5b, if the base or the collector of the transistor is permanently open, the output F is stuck-at-1 (assuming high voltage level = logic 1, low voltage level = logic 0) independent of the input state at X; and if there is a short-circuit between collector and emitter, the output F is stuck-at-0 (s-a-0) irrespective of the input state at X. The "stuck-at" fault model is the most accepted fault model for representing faults in logic circuits. There are two fundamentally different ways of coping with faults in digital systems (15).

(i) The system should be designed in such a way that it should remain operational despite hardware failures, in other words the system



a. A three-input AND gate



b. An INVERTER

Fig 1.5

should contain enough extra (redundant) hardware so that the failures cannot affect the system performance.

(ii) Procedures for testing the correct operation of the system should be provided so that if there are faults in the system they can be detected and located (diagnosis).

1.3 Redundancy Applications In Digital System Design

1.3.1 What is Redundancy?

Complex digital systems are used in applications e.g. deep-space exploration which require reliability levels that cannot be achieved with non-redundant design. This may be primarily because some of the units which make up the system are insufficiently reliable. Hence more than one copy of the same units are incorporated into the system so that if one copy fails there is one similar copy which will carry on functioning, thus failure of the whole system is avoided. This technique is called 'redundancy' and, as a definition , it is the provision of more than one means of accomplishing a given function. Redundancy is not meant to be a replacement but rather a supplement to the other principles of reliable design. Although reliability improves with redundancy, use of extra hardware increases the probability of failure, hence maintenance and check-out become more critical. Techniques for redundancy application can be classified into two broad groups. Static redundancy and dynamic redundancy (16).

1.3.2 Static Redundancy

Static redundancy or masking approach uses a "massive" replication of each component or circuit to two or more copies. All copies are permanently connected and receive power. Component failures or logic faults are automatically corrected by the presence of other copies of the same item. The main advantages of static redundancy are instantaneous

fault correction and simplicity of design operation. Hence it is useful mainly in such applications as the space-craft guidance, where correct operation is required for a relatively short period of time, and repair is impossible.

The form of masking generally in use today was described originally by Von Neumann (17). His idea of using multiple copies of a circuit, each of which receive identical inputs and taking the result that comes out of a majority of these as correct, was the basis of what is now known as 'Triple Modular Redundancy (TMR)' technique. Essentially TMR consists of dividing a non-redundant circuit into several modules, triplicating them, and inserting majority gates (voters) between them. This can be generalised to an N-tuple modular redundant system (NMR) where the modules are replicated N times (where N = 2n+1 with n an integer). At least n+1 out of the N units have to be operational for the system to function (18). In Fig.1.6, the non-redundant network (a) is taken and the module is triplicated and connected with triplicated voters (b). System failure is defined to occur if a TMR system has two or more errors in any of the triplicated output lines.

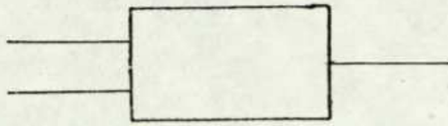
Tyron further developed Von Neumann's basic idea in his development of 'Quadded logic' (19). These three principles form the basis of "quadding":-

(i) All logic appears in quadruplicate

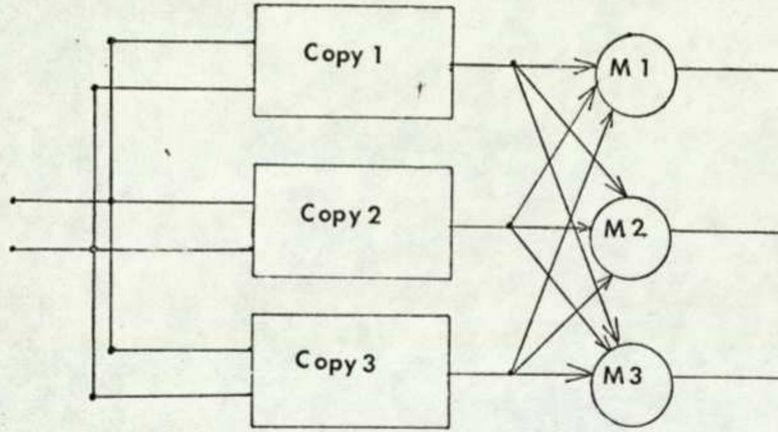
(ii) Every error is corrected within one or two levels of propagation

(iii) Correction is accomplished by mixing "bad" signals with "good" signals from neighbouring units.

Fig.1.7 shows how these ideas can be used to mask the faults in a hypothetical circuit consisting of two single-input AND gate in series with one single-input OR gate. The example shows an error in the first AND gate; this propagates to two errors through the OR gates, and is then

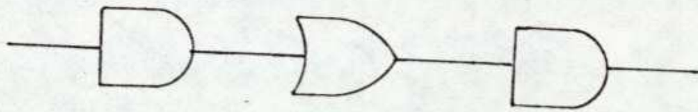


a. Non-redundant network

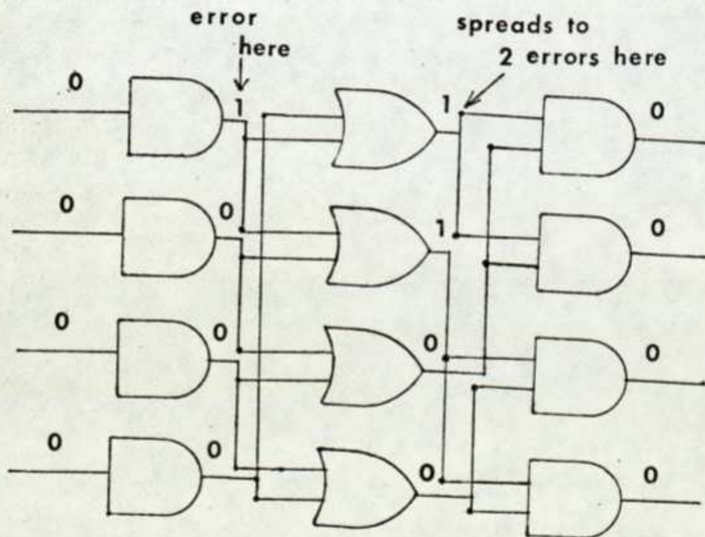


b. Network of (a) triplicated and connected with triplicated voters

Fig I.6



a. Non-redundant circuit



b. Circuit of (a) after quadding

Fig I.7

corrected by the correct signals in the final stage of the AND gate. Error-correcting capabilities of the quadded structure result from the specific interconnection patterns used in the network and their determination may involve some trial and error. The generalisation of the "quadding" concept has been called "interwoven logic" by Pierce (20). Another scheme which is conceptually similar to quadded logic and known as "Radial logic" was proposed by Kalaschka (21).

Although all of these methods increase a network's reliability they have certain drawbacks. For example, majority logic schemes use three times the number of gates and interconnecting wires not taking into account the voters, and quadded logic uses four times the number of gates and interconnecting wires. A new technique, which has certain advantages over the previously described methods, called 'Dotted logic' has been proposed by Freeman and Metze (22). In comparison with other redundancy schemes, it uses less number of elements and redundant interconnections between levels; speed of the network is increased because of the elimination of gates and their inherent delays. In addition to correcting single faults, dotted schemes may easily be extended to cover multiple faults.

1.3.3 Dynamic Redundancy

Dynamic redundancy requires two separate actions. The presence of a fault is first detected, then the recovery action either removes the fault or corrects it. The redundancy used is selective rather than massive. It has certain advantages over its static counterpart (23).

(i) Power is required by only one copy of each replaceable item in a replacement system.

(ii) Fault-isolation is provided between the sub-systems.

(iii) Circuit-related problems like increased fan-out and fan-in

requirements are eliminated.

(iv) The system fails only when all spares of one type are exhausted.

(v) Spare units can be tested by means of a precomputed diagnostic programme.

Interesting examples of extensive dynamic redundancy with software and human support are the ESS systems (24). The experimental JPL-STAR computer (23), intended for self-contained multiyear space missions, was probably the first operational computer with fully hardware-controlled dynamical redundancy. It employs a special Test-And-Repair Processor (TARP) module to control recovery and self-repair.

The concept of using algebric codes (which have been successfully used for the transmission of digital information over noisy communication channels) for improving the reliability of digital computers has been advanced by Armstrong (25), Ray-chaudhuri (26) and others. In a computing system it is more economical to use error-correcting codes, as compared with other redundancy techniques, in the areas of memory, arithmetic unit and input/output devices (27).

1.4 Maintainability Of A Digital System

1.4.1 The Crucial Role of Fault-diagnosis

Fault-tolerant design can mask the occurrence of a limited number of random errors as they occur and provide error-free operations for some fixed period of time (i.e. as new faults occur, the likelihood of the majority of the replicated modules producing a false output increases). Eventually there is a need in these systems to replace the faulty modules. In other systems not employing large-scale fault-tolerant design, immediate human participation is required in the diagnosis of faults and in taking corrective actions. This is of utmost importance for satisfactory system availability. The availability of a system is defined by the

formula:

$$\text{availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

where MTBF = mean time between failures

MTTR = mean time to repair.

If the MTTR can be reduced, availability will increase and the utilization of the system will be more economical. The MTTR can be reduced by a good maintenance policy which enables the time taken to carry out routine procedures and to diagnose and repair a fault to be kept low. A system where failures are rapidly diagnosed may be considered to be more "reliable" than a system which has lower failure rates but where failures remain hidden for a long period and consequently a lengthy system down-time is needed for fault isolation (28). Another reason for increased interest in Fault-diagnosis is the improved production efficiency a digital system manufacturer may obtain if automatic testing methods are used to 'debug' circuit boards and sub-systems.

Much work has been done in finding efficient procedures for fault-diagnosis in digital systems. Fault-diagnosis is concerned with the detection and location of faults. Normally a set of input patterns or tests is applied to the system to detect the existence of faults and then their locations are determined within some degree of resolution. The widespread use of MSI and LSI circuits, coupled with the trend towards larger boards, made manual test generation very expensive and rather difficult to up-date for design changes. Besides in LSI circuits, where the gate density is very high, it is impossible to verify manually the failures detected by a given test pattern. Therefore, over the years, a number of techniques, for the automatic generation of test patterns and subsequent verification of their effectiveness, have been developed, such as:

(1) One-dimensional path-sensitizing

(2) D-Algorithm

(3) Equivalent normal form

(4) Boolean difference

These topics will be reviewed and discussed later in the thesis.

1.4.2 Non-classical Faults in Digital Systems

All test generation procedures mentioned in the previous section are based on the assumption that faults in a digital system can be modelled by the stuck-at or classical faults. Although the "stuck-at" fault model probably covers the most important class of faults, there are certain types of fault e.g. shorted diode (emitter) and bridge faults when the model cannot be applied.

To indicate the 'shorted-diode' effect, consider the circuit in Fig.1.8. If all the gates are faultless, i.e. no shorts and A=0, B=C=1 then E=0 and F=1. But if D2 is shorted, F will be forced to "0" since D3 is forward-biased. The result is that the shorted-diode will go undetected unless A=0 and when it is detected, the fault will be attributed to gate F.

A bridging fault occurs when the outputs of two gates are connected accidently and wired-logic is performed at the connection (29). For example consider the circuit in Fig.1.9, assuming that the circuit was implemented with DTL and positive logic was used. If the outputs of the two NAND gates are connected by the fault, shown in Fig.1.9a, both outputs become equal to the AND of the NAND-gate outputs, shown in Fig. 1.9b. Since E and F are not distinguishable terms from a hardware standpoint, they present testing problems.

Friedman (30) has shown that all non-stuck type of faults can be detected by a standard path-sensitization test procedure for stuck-at faults by adding simple constraints to it. Henceforth, a 'fault' will

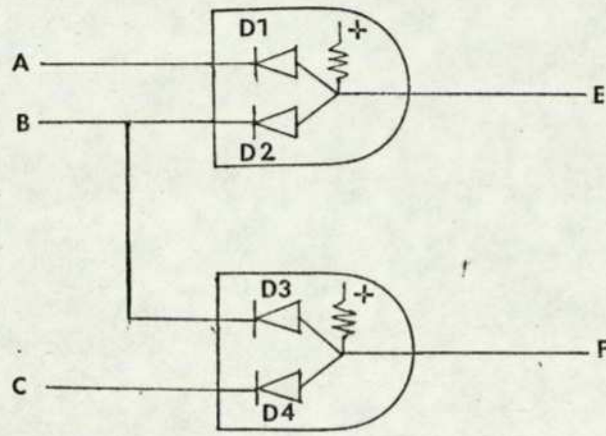
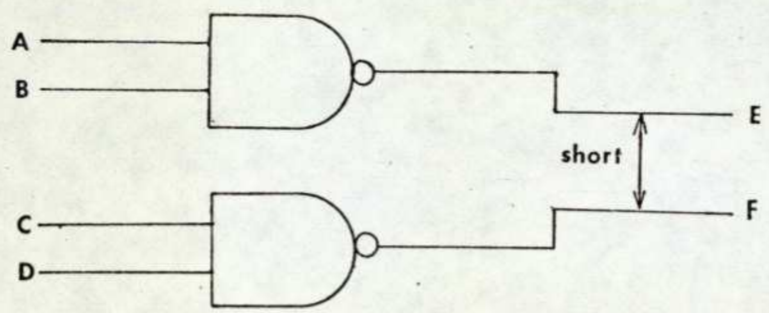
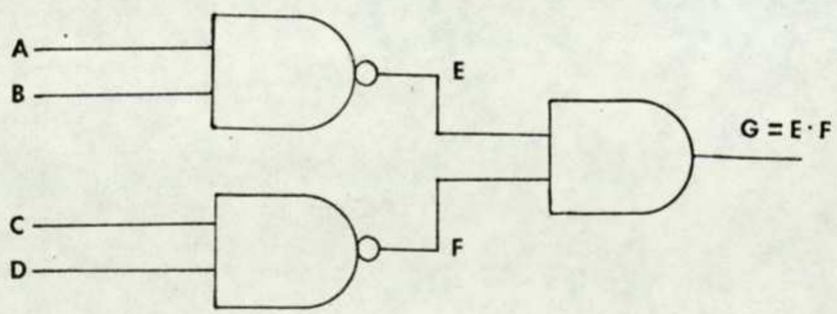


Fig 1.8 AND gate circuit



a. Bridging fault



b. Wired AND logic

Fig 1·9

be assumed to mean a 'stuck-at' fault; 'failures' and 'faults' are also used interchangeably.

1.4.3 Intermittent Faults

Considerable research efforts have been made in finding techniques for the detection and location of solid faults in digital systems (31). However, intermittent faults constitute a fair portion of failures that occur in digital systems. Ball and Hardie (32) have found that the field failures in aerospace digital systems tend to be intermittent in nature. After conducting a series of experiments on the effects and detection of intermittent failures in computer systems, they have concluded that

(i) intermittent faults represent 30% of pre-delivery failures, and almost 90% of operational failures.

(ii) conventional test procedures designed to screen solid faults, fail to detect intermittent faults.

(iii) it is costly to detect intermittent faults and the cost increases with time.

There are three possible ways to deal with intermittent faults in digital systems (32).

(a) Development of better test techniques for the detection and location of intermittent faults.

(b) Development of techniques for inducing the intermittent faults to appear as solid.

(c) Designing of network which will mask the effects of intermittent faults.

Method (a) still remains as an area in need of further research investigation. Recent efforts in this direction can be found in (33) (34) (35). Kamal et al (34) and Kamal (35) suggest a probabilistic model for intermittent faults and introduce a procedure for the diagnosis of a single intermittent fault in an irredundant combinational logic circuit.

Another model of fault intermittency, based on classical probability theory, has been presented by Parker et al (36). The modelling, detection and location of intermittent faults in logic circuits will be investigated later in the thesis.

In the second approach, the intermittent faults are forced to appear by the use of vibrational and thermal stimuli. Although many intermittent faults may be detected in this way, this technique has no scientific basis and needs further investigation.

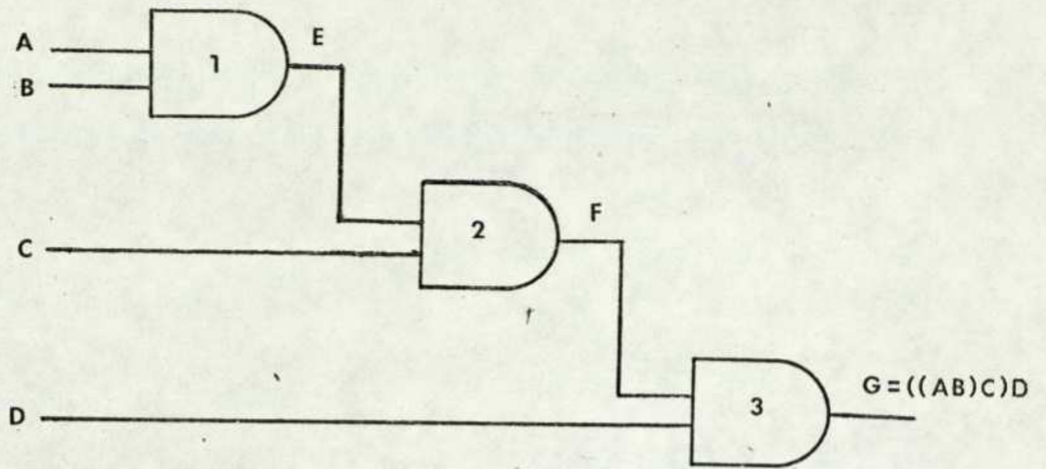
The third approach which uses redundancy techniques to make the system insensitive to the occurrence of intermittent failures, has been found to be very effective. The repetition of programme segments, as used in STAR computer, whenever an intermittent fault is detected seems to be a very promising solution as well. Perhaps the development of these techniques will eventually solve the problem of intermittent failures.

2. DIGITAL LOGIC SIMULATION

2.1 General Review of Logic Simulation

Digital simulation is defined as the process of exercising a software model of a digital system, with sets of input stimuli, to examine its behaviour and reactions. Simulation allows the designer to discover the errors and omissions in design before it is committed to hardware. The redesign cycle is materially reduced and the necessity for building a breadboard is eliminated, although breadboarding can sometimes be utilised for checking.

Simulation processes can be divided into two classes depending on how the circuit logic description is used, (a) compiler driven simulation and (b) table-driven simulation. In compiler driven simulation the circuit is treated as a computer programme in which the function of each circuit element is represented by a subroutine. Each subroutine consists of a series of instructions in machine language code. The subroutines must correspond with the order in which the elements are activitated by the signal in the circuit. Each element is assigned to a level corresponding to the propagation of signals from input to output. An element must be in a higher level than any of its input element and in a lower level than an element which it feeds. A compiler-coded model of a circuit given in Fig.2.1a is shown in Fig.2.1b, where 1 is in the level 1, 2 in level 2 and 3 in level 3. A limitation to this technique is that any change in the circuit involves updating of the assigned levels and hence re-compilation of the codes. Also all the elements in the circuit have to be evaluated, even if only a limited number of them are activated for a particular input stimulation. Examples of compiled-code



a. A Circuit example

```
LOAD  A
 AND  B
STORE E
 AND  C
STORE F
 AND  D
STORE G
```

b. Compiled code model of the circuit

Fig 2.1

simulators can be found in (37) and (38).

In table-driven simulation, the circuit is represented by a data structure with each circuit element stored in a table together with its characteristics such as function (i.e. AND, OR etc.), source of each of its inputs, destination of each of its outputs etc. A computer programme uses these tables to interpret the propagation of logic signals between the circuit inputs and outputs. When the output state of an element is to be evaluated, then a subroutine corresponding to the particular function of the element (stored in the table) is called in by the computer programme; each function is represented by a single subroutine. A circuit and its tabular form representation is shown in Fig.2.2 and Table 2.1 respectively. A change in states of A and B means element 1 must be evaluated, and a change in C means element 2 must be evaluated. The input states A and B form the subroutine "call variables" when the AND subroutine is accessed to evaluate D. Similarly, C and D are the "call variables" for the OR subroutine when the state of E is calculated. The output states of the elements are stored in the appropriate locations as indicated in Table 2.1.

One of the advantages of a table-driven simulator is the ability to limit simulation to only those parts of the circuit where there is logical activity. By keeping track of the output signal lines that change the logic value and the elements they drive, simulation of successor elements is limited to those elements which have a change on their input lines; this is known as 'selective trace simulation' (39). Since simulation is limited only to active logic in the circuit being simulated, the computation time may be materially reduced. Circuit modifications can be easily implemented by changing table entries appropriately. Examples of table-driven simulation can be found in (40) and (41).

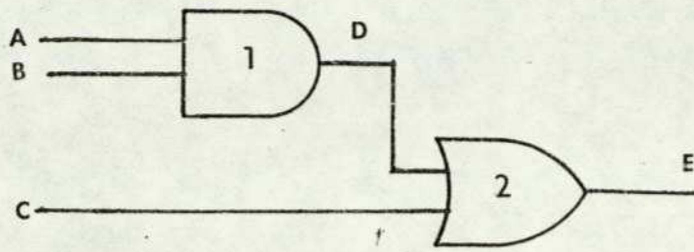


Fig 2.2 Circuit

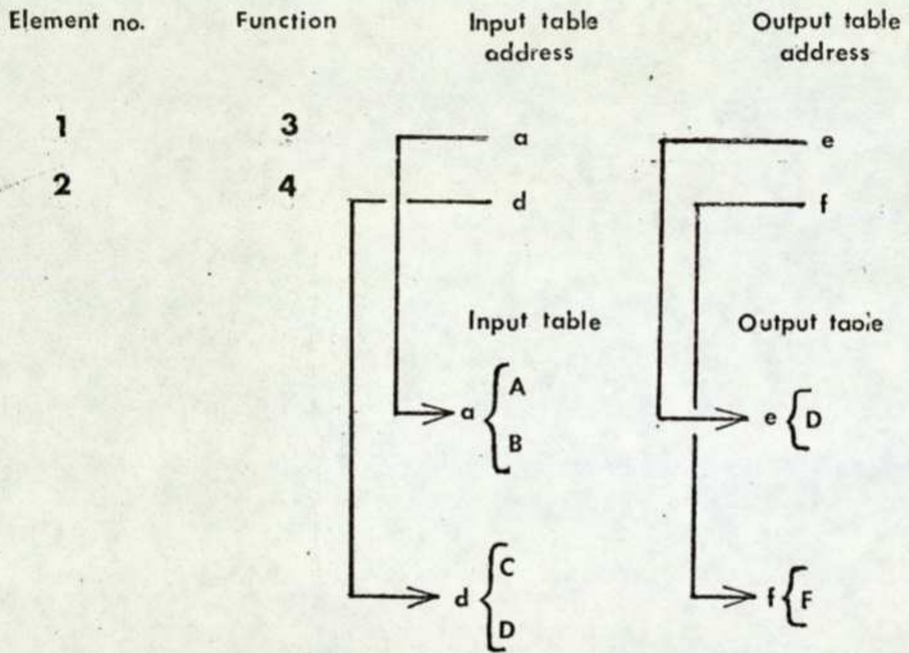


Table 2.1 Table-driven model of the circuit

There are various levels of simulation associated with digital systems; "Register transfer level" (RTL), "Functional level" and "Gate level". The RTL simulation is used to verify the overall system design and operation. Examples of RTL simulators are LOTIS (42), DDL (43) and CDL (44). Complex timing and fault behaviour are seldom considered in RTL. Functional level simulation is used to provide detailed system information such as timing, verification and fault signatures. However large number of states associated with a typical function create storage problem e.g. a 24-bit counter has over a million states. Besides, the function specification is seldom available in tabular form. Gate level simulation is used to model the system operation at the lowest logical level. Timing problems, such as races, spikes and static and dynamic hazards, presently can only be detected in a gate-level simulation environment. In addition, faults can be inserted internally only if the system is simulated at the gate level. Thus, by exercising the gate-level representation of the system, faults can be detected, and the response can be predicted; an example of gate-level simulation is TEGAS2 (45).

2.2 Timing Model in Logic Simulation

Two basic timing model strategies are employed in digital simulation. In the first technique, each digital device in the system is considered to have a fixed propagation delay which can be either zero or non-zero. This technique is known as the 'unit delay' or 'zero-delay' model, depending on the propagation time selected (39). In the other technique known as the 'assignable delay' model, each digital device is assigned a delay that is associated with its behaviour (46).

Zero delay simulators perform no timing analysis, the 'unit delay' model on the other hand permits race and hazard analysis to be performed

since there is a finite delay associated with each device. Assignable delay simulators provide a more accurate model of gate delays in integrated circuits, which vary between 4 and 8 ns for a circuit chip, depending on such factors as manufacturing batch and operating temperatures (3). With an assignable delay model it is possible to assign different propagation times to a device to reflect its specific parameters. For example a device might have different propagation times for the conditions of propagate "1", propagate "0", rise time and fall time; also nominal, minimum, maximum values for each conditions may vary. However an assignable delay simulator is more costly in terms of memory and execution time than the less realistic zero- and unit-delay simulators.

Since most of the simulation for design automation continues to be done using gates as the basic elements, it was decided to use their level of simulation in this project; circuit delays are not modelled because asynchronous sequential circuits, in which circuit timing must be accurately described for the detection of potential race and hazard conditions, are not simulated in this thesis. In combinational circuits consideration of timing of signals is not vitally important since any input vector will always propagate to a stable state value (46). However the following assumptions are necessary for simulating race- and hazard-free operation of a synchronous sequential circuit:-

(i) all clocked transitions occur simultaneously (47) and

(ii) Flip-flop inputs have stabilised prior to the occurrence of the clock pulse (48).

2.3 Coding of the Circuit Structure for Simulation

2.3.1 Combinational circuit model

The coding format used in this project for simulating combinational logic circuits was developed by Ludlow (49). A variation of this

technique will be described later for simulating synchronous sequential circuits. The language used for simulation is FORTRAN IV. Although not an ideal language for this type of simulation, it offers considerable programming flexibility. A description of how the coding format is utilised, is given with reference to the circuit of Fig.2.3. Each circuit element, including the primary inputs and outputs, is given a number. Primary input is a line which is not fed by any other line in the given circuit and primary output is a line whose signal is accessible to the exterior of the given circuit. In a circuit having p-inputs, r-outputs and q-gates, the numbering is done in the following manner:-

(i) Assign numbers 1 to p, starting from the top of the circuit diagram, to the p inputs (in order).

(ii) Assign numbers (p+1) to (p+r), again starting from the top of the circuit, to the r outputs (in order).

(iii) Assign numbers (p+r+1) to (p+r+q) to the q gates in the circuit in such a way that each gate has a higher number than any of its input gates and a lower number than the gates it feeds.

Each type of element in the circuit is given a function code:-

FUNCTION	CODE
INPUT	1
OUTPUT	2
AND	3
OR	4
NAND	5
NOR	6
INVERTER	7
EXCLUSIVE-OR	8

The data-table structure of the circuit in Fig.2.3 is shown in Table 2.2a.

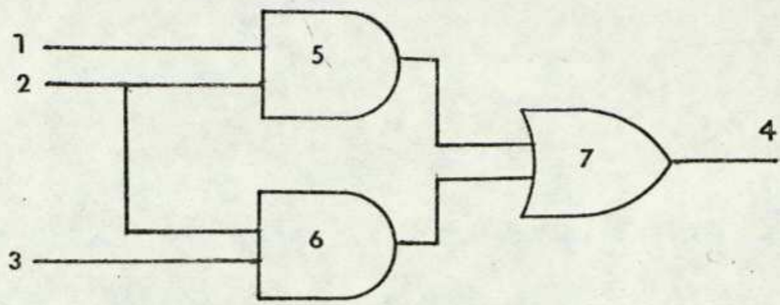


Fig 2.3 A circuit example

Element no.	Function	Connection data
1	1	0 5
2	1	0 5 6
3	1	0 6
4	2	7
5	3	1 2 0 7
6	3	2 3 0 7
7	4	5 6 0 4

a. Coded structure of the circuit

Circuit element no.	1	2	3	4	5	6	7							
NELEM	1	2	4	5	6	9	12							
NFUNC	1	1	1	2	3	3	4							
NCONC	5	5	6	6	-7	-1	-2	7	-2	-3	7	-5	-6	4
Array element no.	1	2	3	4	5	6	7	8	9	10	11	12	13	14

b. Circuit format in program arrays

Table 2.2

Each element in the circuit is described by

(a) the element number (E)

(b) the element function

(c) the element numbers which feed element 'E'

(d) the element numbers which are fed by 'E'.

The 'zero' in the data table indicates the termination of a list of input or output connections. The simulation programme reads in the circuit structure and stores the data in three one-dimensional arrays, NELEM, NFUNC and NCONC. The jth array element of NELEM and NFUNC contains the jth numbered circuit element and its function respectively. The jth element of NELEM contains the address of array position m of NCONC where the input and output connections of circuit element j is stored. If the jth element has i input connections and k output connections then the data is stored in the mth to (m+i+k)th array elements of NCONC. The input connections are identified by negative and the output connections by positive values. The coded circuit structure of Table 2.2a for example would be stored in the three arrays as shown in Table 2.2b after reading in by the computer. The logic states of the circuit elements during the simulation process are stored in a one-dimensional array NSTAT. The array NEXTS is utilised for temporary storage of the logic state of an element during the process of evaluation.

2.3.2 Development of sub-routines for function evaluation

The subroutines for functional evaluation of some commonly used logic circuit elements e.g. AND, OR, NAND and NOR have been described in (49). Functional subroutines for INVERTER, EXCLUSIVE-OR, D FLIP-FLOP, JK FLIP-FLOP and RS FLIP-FLOP have been developed by the author.

The subroutines for the gate elements are described here; those for the memory elements will be described in the next section. For

computer simulation of logic circuits, it was decided to use 1 and 2 for logic 0 and logic 1 respectively; the reasons for this assumption was that if due to programming error a zero appears in the program output, it might be confused with a logic 0 and the subsequent location of the error could be extremely time consuming. All the programme flow-diagrams enclosed in this thesis reflect this assumption. In many cases the programme inputs and outputs are directly reproduced, under a figure or table number; therefore this assumption should be borne in mind while referring to a figure or table.

(a) AND (Fig.2.4a) Find the jth element of array NELEM; this is the address in NCONC where the connection data of the jth element are stored. If the logical state of any input connection is 0 then NEXTS(j) = 0, but if all input connections are in the logical state 1 then NEXTS(j) = 1. Arrays are indicated by writing a superscript (or superscripts) after the name, in all the flow-diagrams enclosed in this thesis e.g. NEXTS(j) is shown as $NEXTS^j$.

(b) OR (Fig.2.4b) The logic states of the input connections are evaluated in the same way as AND but to simulate the OR function, NEXTS(j) = 1 if any input connection is at logic 1 and NEXTS(j) = 0 if only all input connections are at logic 0.

(c) NAND (Fig.2.4c) This function is simulated in an identical way as AND except that NEXTS(j) = 1 if any

input connection is at logic 0 and NEXTS(j) = 0 if all input connections are at logic 1.

(d) NOR (Fig.2.4d) — This function is simulated in an identical way to OR except that NEXTS(j) = 0 if any input connection is at logic 1 and NEXTS(j) = 1 if all input connections are at 0.

(e) INVERTER (Fig.2.4e) — This function is evaluated in the same way as NAND and NOR with only one input connection.

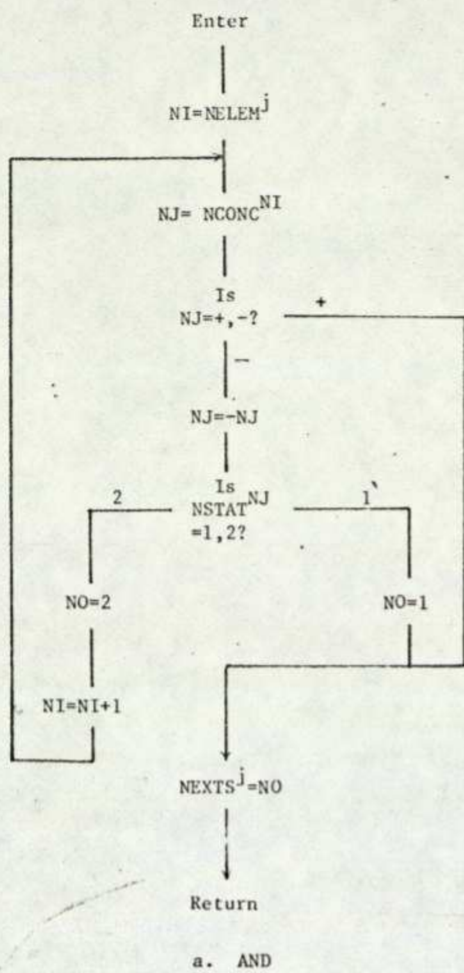
(f) EXCLUSIVE-OR (Fig.2.4f) — The logic states of the input connections are evaluated in the same way as OR but to simulate the EX-OR function, NEXTS(j) = 0 only when either NTR = 0 (all input connections are at logic 0) or MTR = 0 (all input connections are at logic 1); otherwise NEXTS(j) = 1.

(g) INPUT (Fig.2.4g) — The test input values at the primary inputs are stored in the corresponding positions for the inputs in NSTAT and NEXTS.

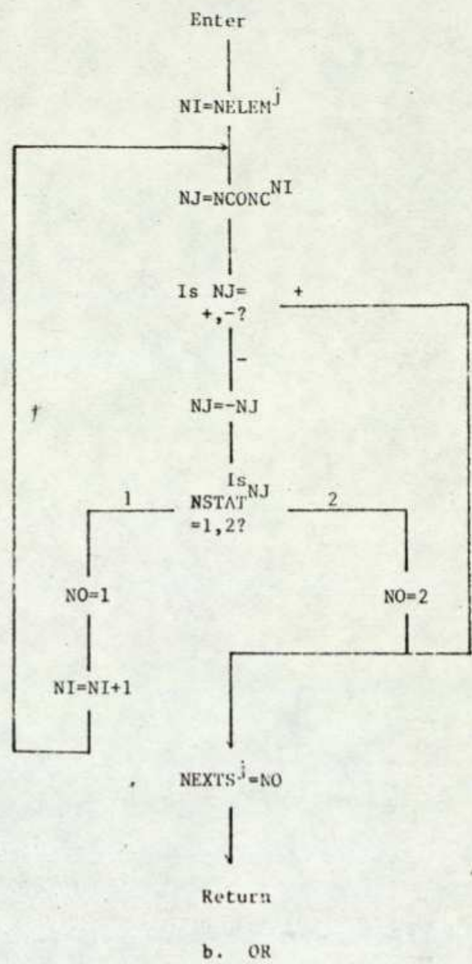
(h) OUTPUT (Fig.2.4h) — The logic state of the input connection is evaluated and stored in the appropriate position for the primary output elements in NSTAT.

2.3.3 Synchronous sequential circuit model

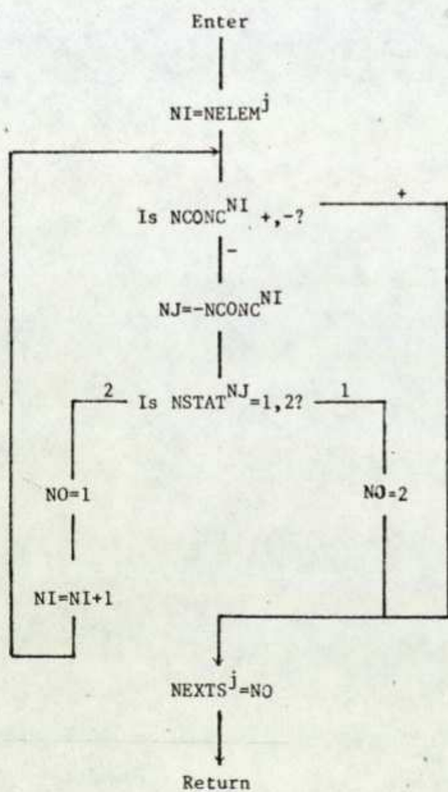
Consider the Huffman model of the synchronous sequential circuit as shown in Fig.2.5a; this time-sequential model can be represented by a



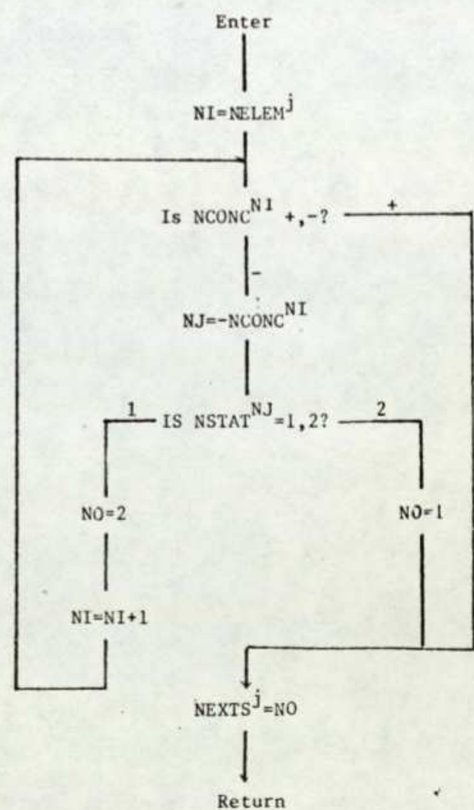
a. AND



b. OR



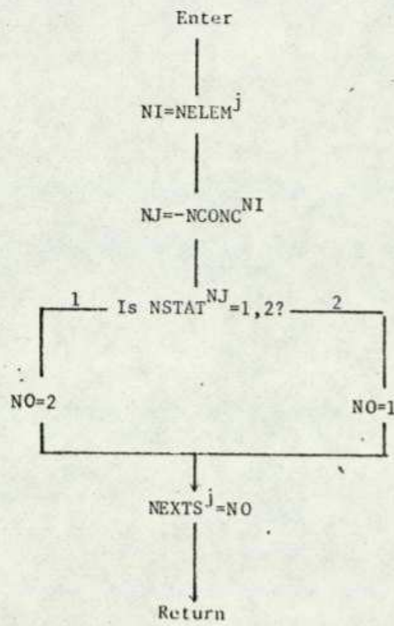
c. NAND



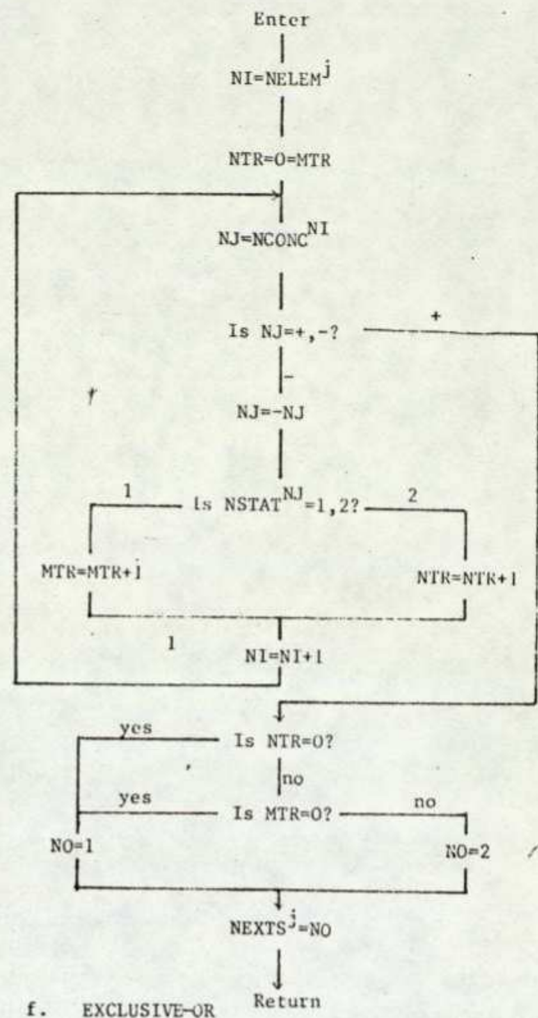
d. NOR

Fig.2.4 Flow-diagram of subroutines used for simulating gate functions

cont'd.

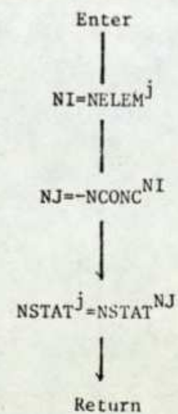


e. INVERTER



f. EXCLUSIVE-OR

g. INPUT



h. OUTPUT

space-sequential model shown in Fig.2.5b. $X(t)$ represents the primary input at time t, $Y(t)$ represents the primary output, $S(t)$ is the present state and $Z(t)$ the next-state of the circuit. Combinational logic is assumed to respond to signal changes instantaneously. Clocked memory elements tranform $Z(t)$ to $S(t+1)$ after the clock pulse is issued. Simulation procedure for test generation and verification in synchronous sequential circuits will be described later in the thesis. In a synchronous sequential circuit having X inputs, Y outputs, K number of flip-flops (S outputs, Z inputs) and q elements in the combinational logic, the numbering is done in the following way:-

(i) Assign numbers 1 to (X+S), starting from the primary input to the (X+S) number of inputs of the combinational logic. The numbers are allocated to the input of the combinational logic in such a way so that the number assigned to the $\bar{Q}$ output of a flip-flop is greater than the one assigned to the Q output by K.

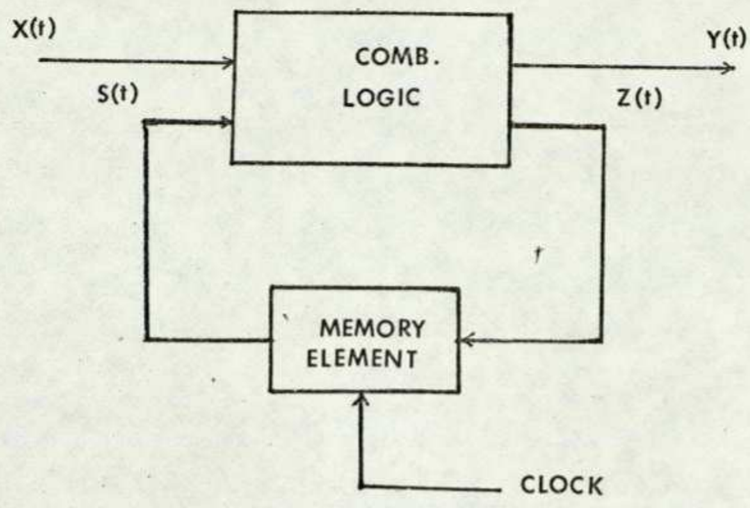
(ii) Assign numbers (X+S+1) to (X+S+Y+Z) starting from the primary output to the (Y+Z) number of outputs of the combinational logic.

(iii) Assign numbers (X+S+Y+Z+1) to (X+S+Y+Z+q) to the q gates in the combinational logic so that each gate has a higher number than any of its input gates and lower number than any gate it feeds.

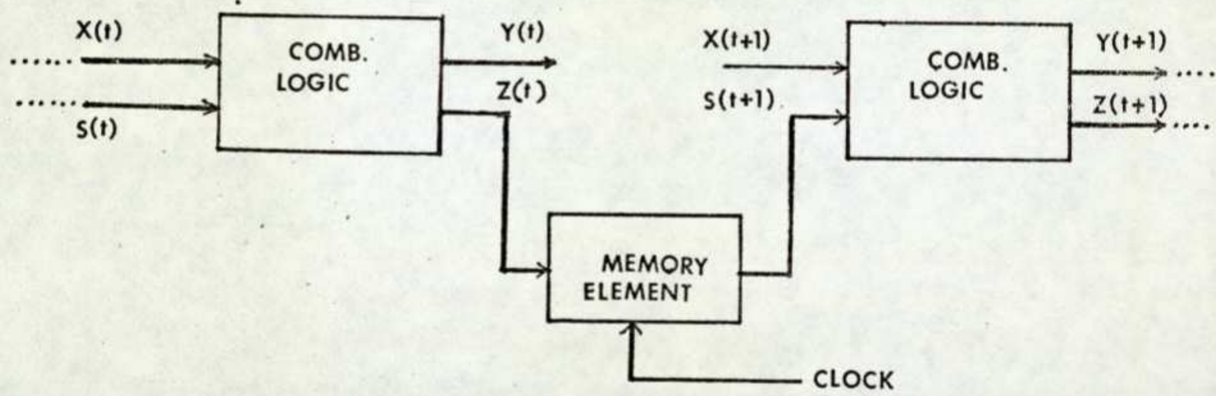
(iv) Assign numbers K to 1, starting from the top to the K flip-flops in the circuit, in order.

The circuit shown in Fig.2.6 has been numbered according to the above procedure.

It is assumed that only one type of flip-flop is used in the feedback circuit. A functional code is assigned to each type of flip-flop,



a. Time-sequential model



b. Space-sequential model

Fig 2.5

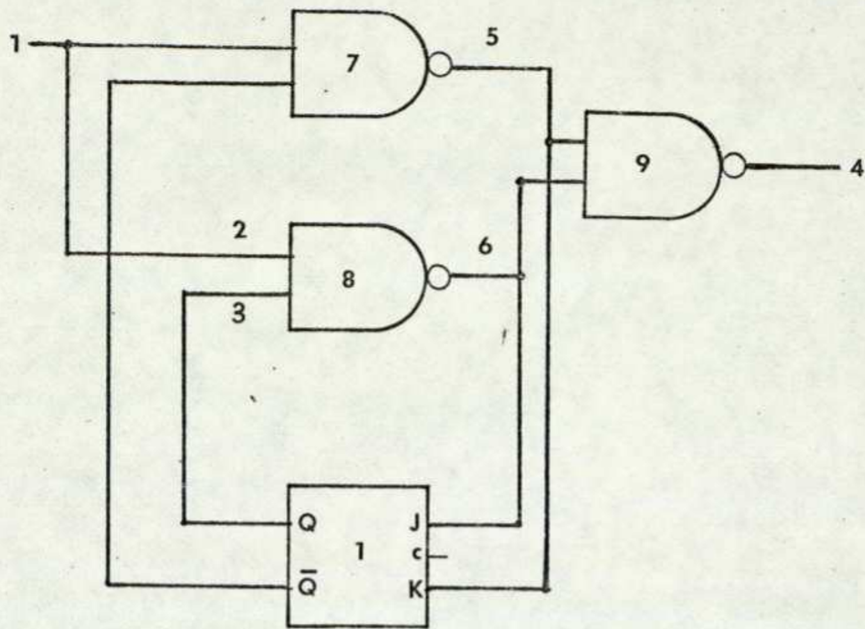


Fig 2.6 A Synchronous sequential circuit

Element no.	Function	Connection data			
1	1	0	7	8	
2	1	0	8		
3	1	0	7		
4	2	9			
5	2	7			
6	2	8			
7	5	1	3	0	5
8	5	1	2	0	6
9	5	7	8	0	4
FF no.					
1		6	5		

Table 2.3 Coded structructure of the circuit

TYPE	CODE
D	1
JK	2
SR	3

Each flip-flop in the circuit is described by

(a) the flip-flop number

(b) the input connections to the flip-flop.

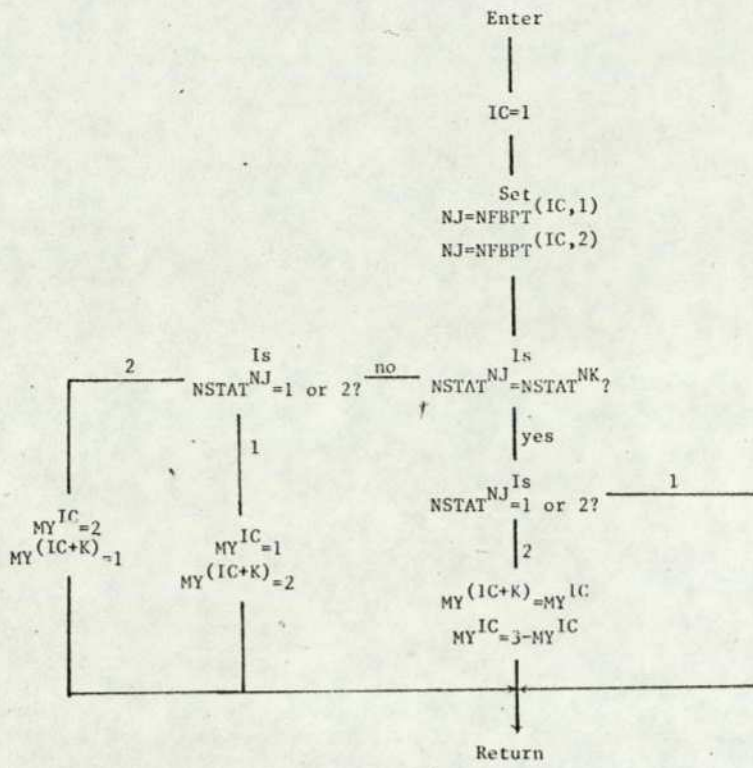
In the case of a JK(RS) flip-flop the J(S) input connection precedes the K(R) input connection; in the D-type flip-flop the data input connection is followed by a zero in the data table structure.

The data table structure of the circuit in Fig.2.6 is shown in Table 2.3.

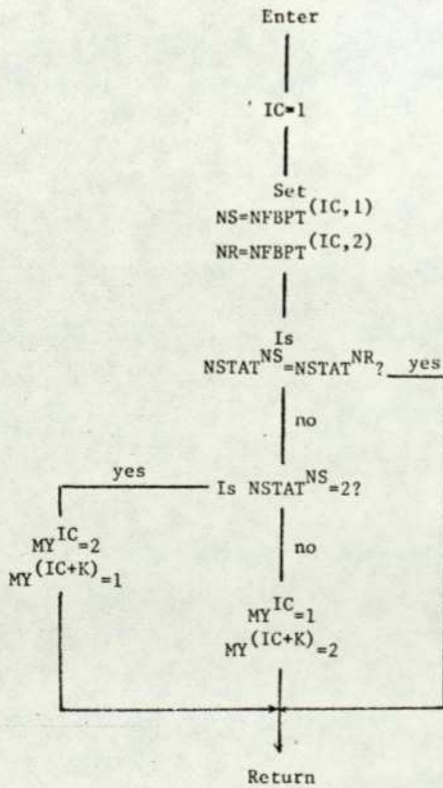
The input connections of the flip-flops are stored in a two-dimensional array NFBPT(IC,2) of the simulation programme; row IC corresponds to a particular flip-flop and the two columns store its input connections. A one-dimensional array MY stores the output states of a flip-flop. The functional routines for the clocked flip-flops are constructed in the following way:-

(a) JK-FLIP FLOP (Fig.2.7a) Corresponding to the flip-flop to be simulated, find the input connections of the flip-flop from array NFBPT; the output state Q and its complement $\bar{Q}$ are determined by the logical states of the input connections. When both the input connections are at logic 1, the output 'toggles'.

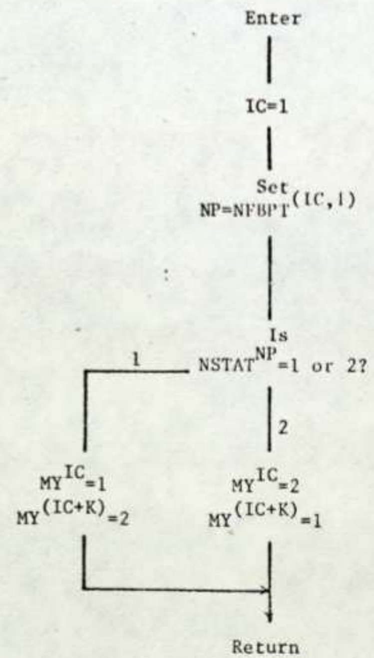
(b) RS FLIP FLOP (Fig.2.7b) The functional routine for a clocked RS flip-flop is evaluated in exactly the same



a. 'JK FLIP-FLOP' function



b. 'RS FLIP-FLOP' function



c. 'D FLIP-FLOP' function

Fig.2.7

way as a JK flip-flop but the possibility of both R and S input connections being at logic 1 at the same time (race condition) is ignored.

(c) D-FLIP FLOP (Fig.2.7c) The output state Q of the D flip-flop is determined by the logic state at its data input connection.

2.4 Fault Simulation in Logic Circuits

2.4.1 Techniques for fault simulation

Once the design of a digital system is completed, it is important to determine how faults will affect the operation of the system. This requires a scheme for inserting specific faults into the system and a prediction of its subsequent response to a test input. Three methods are normally used for fault simulation in logic circuits: manual, physical and digital (14). In manual simulation, an engineer traces the effect of each fault in the presence of each input test to the circuit output. Physical simulation involves the actual insertion of a fault into the working model of the system and then the response of the system to the relevant tests or testing sequences is observed. The first method may be very efficient for small circuits but is very much prone to human error. The second method is very expensive and time-consuming because not only a prototype system has to be built but its response to the test programme in the presence of each inserted fault has to be studied as well.

Digital simulation is the most popular method for fault-simulation and is used in this thesis. In this method the fault is injected into the circuit model and the simulation programme predicts the reactions of the circuit in the presence of various test inputs. Commonly one fault

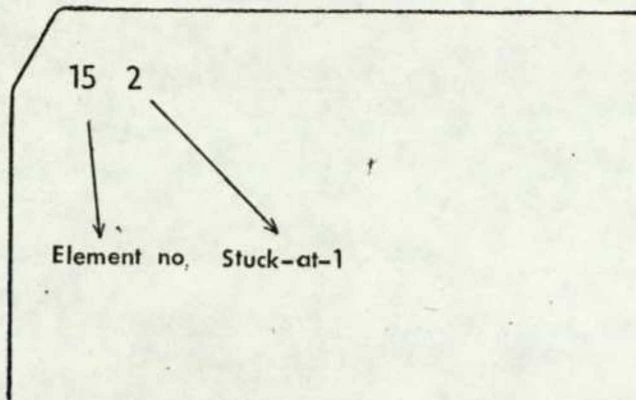
at a time is inserted for fault-simulation. This so-called "single-fault assumption" is statistically based on the mean time between faults of the system. The probability for detection of a fault is assumed to be high enough so that every fault is detected and corrected in a time period which is short compared to the mean time between faults. But the single-fault assumption may yield wrong results for redundant circuits (50), and may not be correct in relation to the acceptance testing after manufacture of circuits having very high component densities as in LSI. For these reasons it is more realistic to assume a multiple-fault model.

2.4.2 Fault insertion

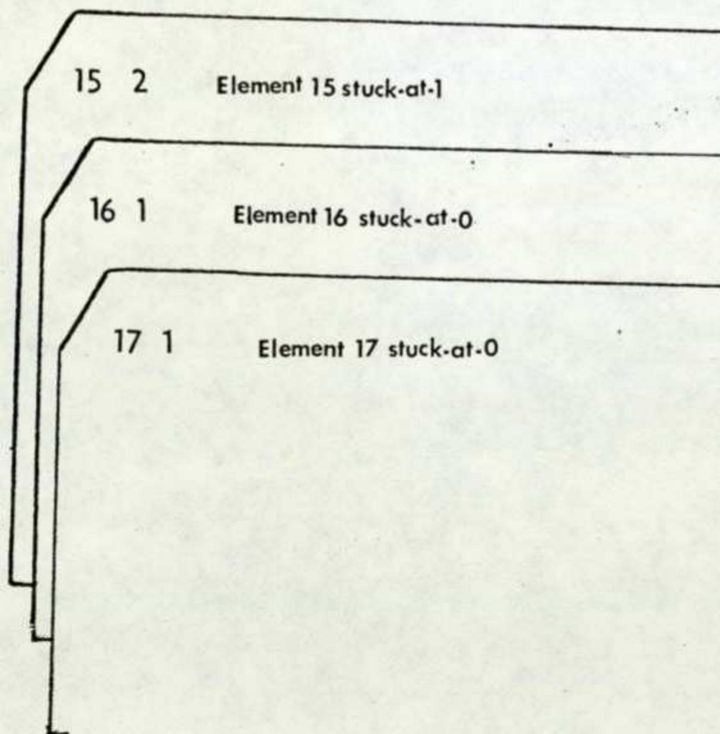
A technique for setting up a single fault in the circuit model during the process of simulation and resetting it has been described in (49). A modification of this technique will be used in this thesis for simulating faults having any multiplicity; thus when multiplicity is '1' a single fault is simulated.

It is assumed that faults occur only at the output of a gate. This is a reasonable assumption since if the output of gate C feeds in gate D and if the fault lies between C and D then it is not possible to say whether the fault is in the output terminal of gate C or the input terminal of gate D; since the reaction at the primary output will be identical for either of them (51). A fault is thus injected only at the output. The format for describing a single fault is illustrated in Fig.2.8a which indicates that the output of gate number 15 is stuck-at-1. Fig.2.8b illustrates the description of a multiple-fault which has a multiplicity (MLTFLT) of 3.

The data is stored in two one-dimensional arrays N1STR and IFALT. Consider the data of p faults (= 1,2,.....q.....MLTFLT) input



a. Single Fault



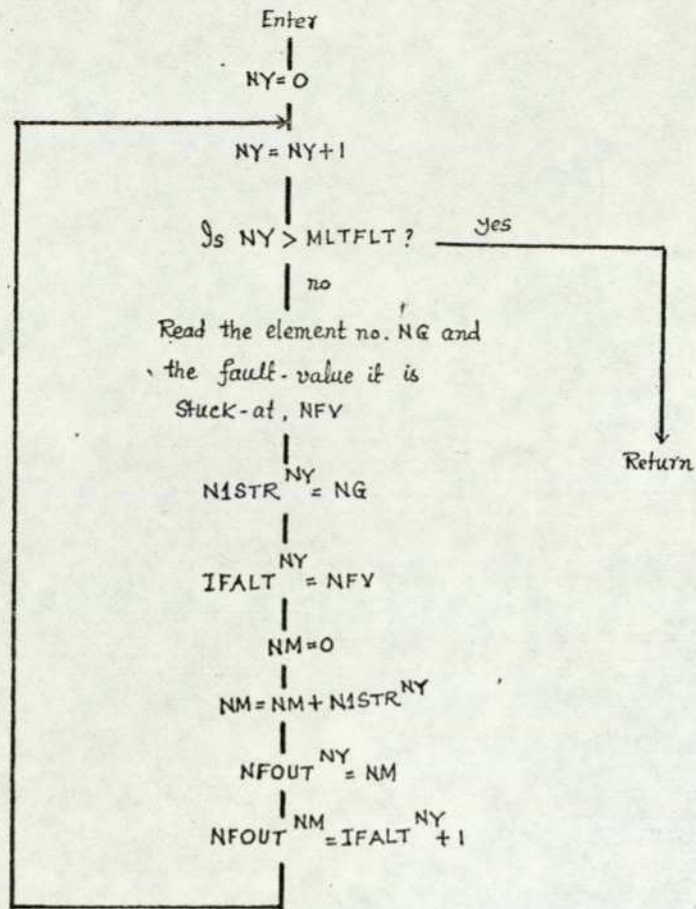
b. Multiple Fault

Fig 2.8

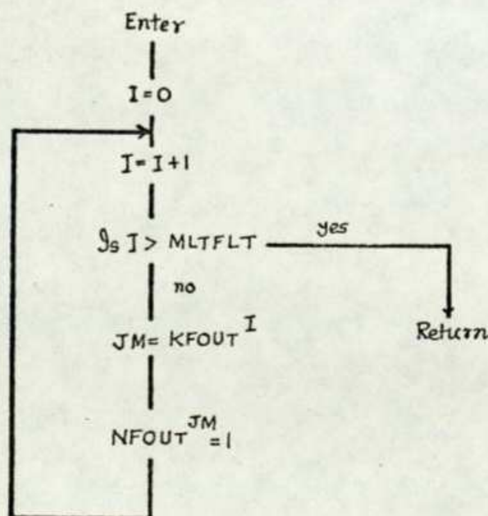
into the master routine. The faulty element numbers are stored simultaneously in N1STR(q) and KFOUT(q). IFALT(q) contains logic level of the fault (s-a-0 or s-a-1). Array NFOUT sets up the faults and removes them as desired. If the output of gate q is to be fixed at the (faulty) logic level 0 or 1, NFOUT (q) is set to 2 or 3 (where 3 corresponds to s-a-1, 2 to s-a-0 and 1 to no fault). The functional subroutines described previously in section 2.3.2 are modified in such a way that array NFOUT interacts each time a particular function is evaluated. The outputs of the circuit elements concerned are set up accordingly. Before setting up a new set of faults, the existing one is removed.

Flow-diagrams for fault set up and removal operations are shown in Fig.2.9(a) and (b). Modified versions of functional subroutines to take into account the effect of NFOUT are illustrated with the flow-diagrams of NAND,INPUT and OUTPUT elements in Fig.2.10.

In the case of flip-flops, it is assumed that a fault may occur at some excitation circuit to the flip-flop or at the output of the flip-flop; if the output of one side of the flip-flop is stuck-at some value, say 1, the output of another side of that flip-flop will take the value complementary to that stuck-at value, in this case 2. Since flip-flop outputs are connected to the inputs of the combinational logic, fault-simulation for the flip-flops is achieved by simulating the corresponding input faults to the combinational logic.

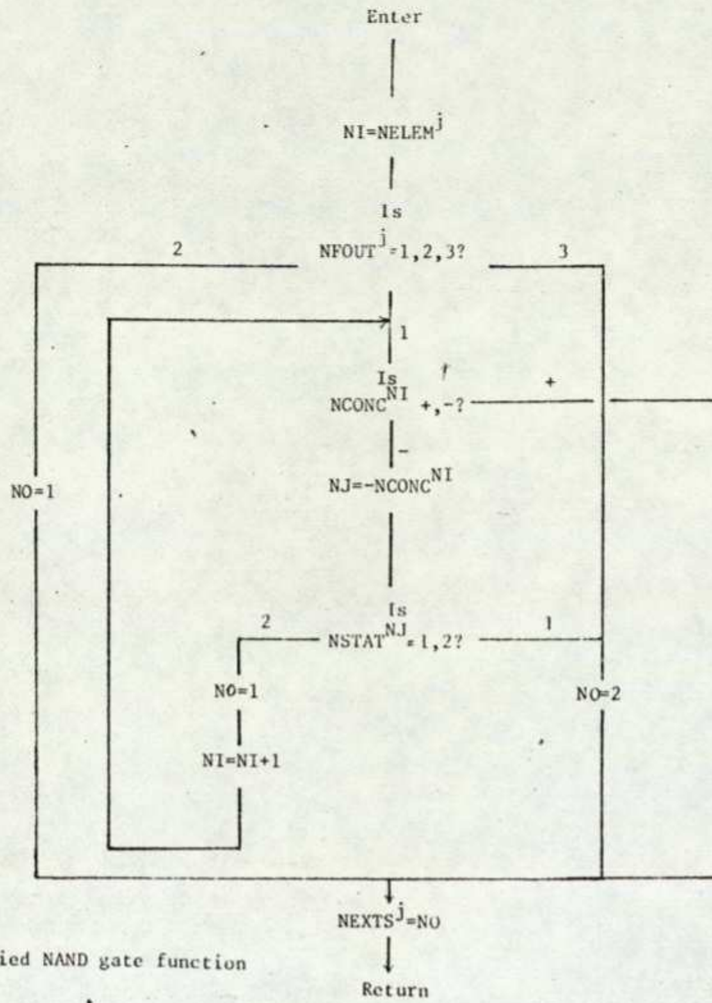


a. Fault set-up process

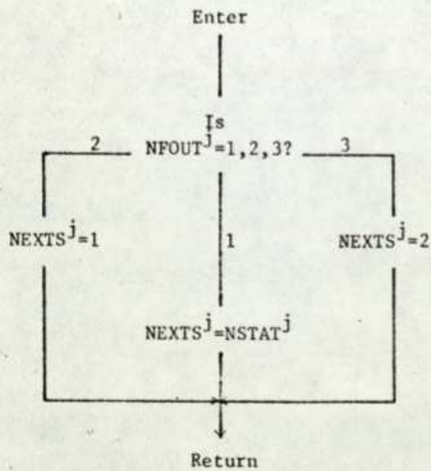


b. Fault removal process

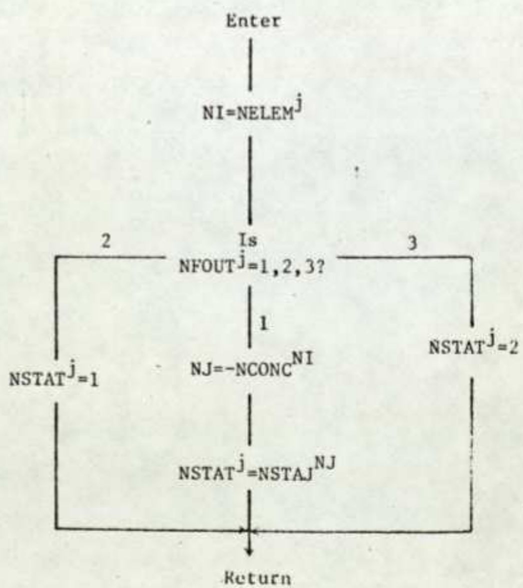
Fig 2.9



a. Modified NAND gate function



b. Modified INPUT function



c. Modified OUTPUT function

Fig.2.10

3. DIAGNOSIS OF SOLID FAULTS IN COMBINATIONAL LOGIC CIRCUITS

3.1 Introduction

This chapter commences by discussing various methods of test generation available for combinational logic circuits; then follows an explanation of the idea of 'Fault-folding'. Next a method for the diagnosis of multiple faults in combinational circuits, based on fault-folding, which was devised by the author, is described. This method will be illustrated by applying it to a simple circuit. The flow-diagrams of various programmes, used to computerise the method, will be described. The chapter closes by giving examples of the use of the computer algorithm in deriving minimal diagnostic test sets for various combinational circuits and a verification is made that the derived test set detects and locates multiple faults in a circuit. In all cases it is assumed that access is available only to the primary input and output of a network for the purpose of detection and location of a set of faults.

3.2 Test Generation for Diagnosing Solid Faults in Combinational Logic Circuits

Test generation means finding an input pattern which, when applied to the circuit under test, detects a specific fault. Much effort has been devoted to developing efficient algorithms for test generation. A survey of the principal theoretical and experimental approaches can be found in (31), (52), (14). In this section some of the methods developed over the years are described.

3.2.1 Truth table method

Let $F(X_n)[=F(x_1,x_2,\ldots..x_n)]$ represent the output of a fault-

free circuit and let s/a be a fault for which tests are to be derived. The method starts with the construction of truth tables of the normal and faulty circuits. A test for a given fault must produce a different response when the fault is present, $F(X_n^{s/a})$ than when the fault is not present, $F(X_n)$. Thus an input is a test for detecting the fault s/a in the circuit, if and only if

$$F(X_n) \oplus F(X_n^{s/a}) = 1.$$

where $\oplus$ represents the exclusive-OR operator.

Consider the circuit shown in Fig.3.1 and let the element 5 be s-a-1. The truth tables of the output functions of the fault-free and faulty circuit is shown in Table 3.1. From Table 3.1 it is seen that the following tests detect the fault:-

$$(x_1 x_2 x_3) = (000);(010);(100).$$

However this method is not practicable even for circuits of moderate size because a truth table has to be constructed for every fault, which results in excessive computation.

3.2.2 One-dimensional path sensitization

The approach taken in the path-sensitization method is to first select a path from the fault site to the network output. Inputs are then chosen so that the logic value of lines in the path are a function of the fault. A path has been 'sensitized' from the fault-site to the output when this condition has been established (53) . Consider the test determination for fault 5 s-a-1 in the circuit of Fig.3.1. The input at $x_1 x_2$ must be either 00, 01 or 10 to initialize the path since a s-a-1 fault is being considered. A path is sensitized to the output by establishing logic value 0 at x_3. Hence the fault 5 s-a-1 can be detected by either of the three tests 000, 010 or 100.

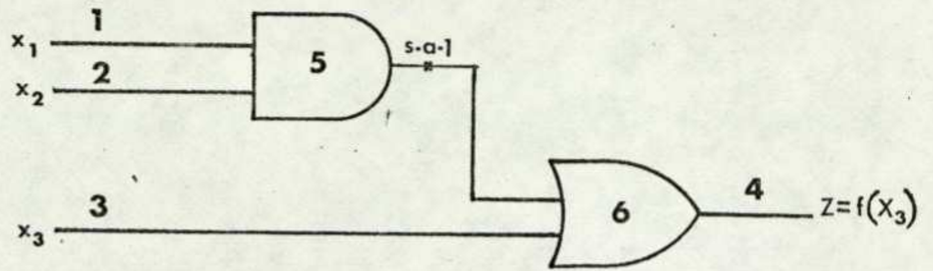


Fig 3.1 Circuit under test

x_1	x_2	x_3	f	$f^{5s\text{-}a\text{-}1}$	$f \oplus f^{5s\text{-}a\text{-}1}$
0	0	0	0	1	1
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0

Table 3.1 Exhaustive testing

The path-sensitizing method can be summarised as follows:-

(i) Select the fault for which tests are to be determined and select a path from the fault site to the network output.

(ii) Sensitize the path (forward trace).

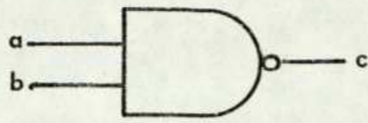
(iii) Establish network inputs as required by 2 above (backward trace).

A useful application of path sensitizing besides test generation is the determination of the set of faults tested for by a given input. However an example has been found which shows that this method may occasionally fail to find a test for a fault even though it exists (54). This is due to the presence of reconvergent fan-out paths in the circuit.

3.2.3 D-Algorithm

This is the first algorithmic method for test generation of combinational circuits. The essence of the method is to sensitize all possible paths from the site of failure to the circuit outputs simultaneously (55)(56).

In the D-algorithm two expressions are used to describe the function of a logical block. One is the 'singular cover' expression and the other is the 'D-cube' expression. Fig.3.2a shows a NAND block with its singular cover. Each row of the singular cover, termed as a singular cube, describes the relationship between the inputs and the output. The first row 0x1 for example specifies that when a=0, then the output c=1 regardless of the value of b. Fig.3.2b shows the D-cubes which also express the relationship between the inputs and the output. The letter D may assume only one value, 0 or 1; $\bar{D}$ takes on the value opposite to D; i.e. if D=1, then $\bar{D}$=0; when D=0, then $\bar{D}$=1. A D-cube of any logic block is called a 'primitive D-cube'. These cubes denote the ways in which one or more inputs can be made to control a gate's output. Cubes that only involve one gate input are termed 'single D-cubes' and those



a	b	c
0	x	1
x	0	1
1	1	0

a. Gate and singular cover

a	b	c
D	1	$\bar{D}$
1	D	$\bar{D}$
D	D	$\bar{D}$

a	b	c
$\bar{D}$	1	D
1	$\bar{D}$	D
$\bar{D}$	$\bar{D}$	D

b. Primitive D-cubes of the gate

Fig. 3.2

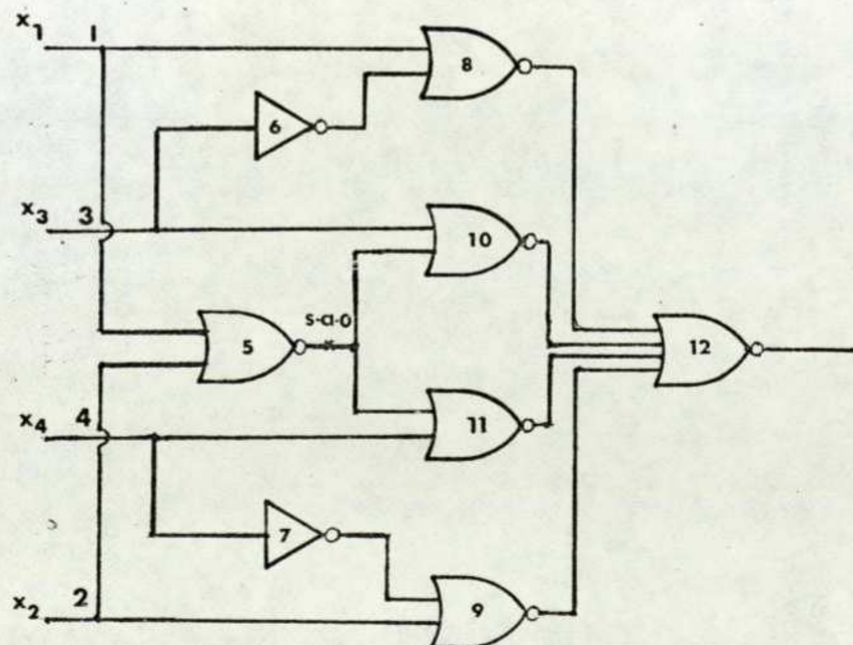


Fig 3.3 Circuit example

which denote two or more inputs jointly controlling a gate output are termed "double or multiple D-cubes".

The 'primitive D-cube of a failure' (pdcf) is used to specify how to exhibit the existence of a given fault. It consists of an input pattern which brings the influence of the failure to the output of the block. Each input of the block is assigned 1,0 or X and the output with influence is assigned D or $\bar{D}$. If the working circuit has 1(0) on a given line and the circuit with the given fault has 0(1), then a signal D($\bar{D}$) is assigned to the line. For example for the output of line C, in Fig.3.2a, stuck-at-1, the corresponding pdcf is 11$\bar{D}$.

The construction of a test uses a new kind of intersection termed 'D-intersection' of D cubes. This operation is used to make a D-cube of the whole circuit from the primitive D-cubes and singular covers, and is formally defined in (55).

The method of construction of a test starts with the selection of a pdcf of the given failure. Then D's are driven to primary outputs by successive D-intersections with the primitive D-cubes (d-operation). Finally the 'consistency operation' is used to construct a consistent primary input vector which realizes all of the conditions generated during the D-drive; this is similar to the backward-trace phase of the one-dimensional path-sensitizing method.

To illustrate the application of the method, consider finding a test for fault 5 s-a-0 in Fig.3.3. The pdcf of 5 s-a-0, termed as initial test cube tc^{o}, is

1	2	5
0	0	D

To propagate the fault to the primary output, drive D on 5 through either gate 10 or 11 or both; accordingly the D-cubes of gates 10 and 11

are

$$\begin{array}{ccc} 3 & 5 & 10 \\ 0 & D & \bar{D} \end{array} \qquad \begin{array}{ccc} 4 & 5 & 11 \\ 0 & D & \bar{D} \end{array}$$

To propagate the fault through gate 10 requires the D-intersection of tc^{o} with the cube of gate 10 which results in a new test cube tc^{1}

$$\begin{array}{ccccc} 1 & 2 & 3 & 5 & 10 \\ 0 & 0 & 0 & D & \bar{D} \end{array}$$

After specifying all signal values which can be determined from tc^{1} we obtain $tc^{1}*$,

$$\begin{array}{ccccccc} 1 & 2 & 3 & 5 & 6 & 8 & 10 \\ 0 & 0 & 0 & D & 1 & 0 & \bar{D} \end{array}$$

To propagate the fault on gate 10 to the primary output 12 it is necessary to D-intersect $tc^{1}*$ with the cube of gate 12.

The cube of gate 12 is

$$\begin{array}{ccccc} 8 & 9 & 10 & 11 & 12 \\ 0 & 0 & \bar{D} & 0 & D \end{array}$$

After D-intersection, the new test cube tc^{2} results

$$\begin{array}{cccccccccc} 1 & 2 & 3 & 5 & 6 & 8 & 9 & 10 & 11 & 12 \\ 0 & 0 & 0 & D & 1 & 0 & 0 & \bar{D} & 0 & D \end{array}$$

Next the consistency operation has to be applied to determine the value of the primary input lines corresponding to the value 0 on gates 8,9 and 11. The singular covers of blocks 8,9 and 11 are

$$\begin{array}{ccc} 1 & 6 & 8 \\ x & 1 & 0 \end{array} \qquad \begin{array}{ccc} 2 & 7 & 9 \\ x & 1 & 0 \end{array} \qquad \begin{array}{ccc} 4 & 5 & 11 \\ 1 & x & 0 \end{array}$$

After D-intersection of tc^{2} with the singular cube of block 8, a new test cube tc^{3} results,

$$\begin{array}{cccccccccc} 1 & 2 & 3 & 5 & 6 & 8 & 9 & 10 & 11 & 12 \\ 0 & 0 & 0 & D & 1 & 0 & 0 & \bar{D} & 0 & D \end{array}$$

D-intersection of tc^3 with the singular cube of block 9 results in test cube tc^4,

1	2	3	5	6	7	8	9	10	11	12
0	0	0	D	1	1	0	0	$\bar{D}$	0	D

D-intersection of tc^4 with the singular cube of block 11 results in test cube tc^5

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	1	D	1	1	0	0	$\bar{D}$	0	D

But the singular cube of block 7

4	7
0	1

when D-intersected with tc^5, results in the null-intersection, tc^6

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	ϕ	D	1	1	0	0	$\bar{D}$	0	D

which shows that fault 5 s-a-0 cannot be propagated via gate 10. It can also be shown that it is not possible to propagate it through gate 11 either. If only it is driven simultaneously through gates 10 and 11 then it is detectable as shown below

$tc^1 = tc^0 \cap$ (D-cube of 10) $\cap$ (D- cube of 11).

1	2	3	4	5	10	11
0	0	0	0	D	$\bar{D}$	$\bar{D}$

tc^{1*} results from tc^1 as described previously

1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	D	1	1	0	0	$\bar{D}$	$\bar{D}$

A double-cube of gate 12 is

8	9	10	11	12
0	0	D	D	$\bar{D}$

After D-intersecting tc^{1*} with the cube of gate 12, we obtain the final test cube tc^2

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	D	1	1	0	0	$\bar{D}$	$\bar{D}$	D

Thus $\overline{X1}\ \overline{X2}\ \overline{X3}\ \overline{X4}$ is a test for detecting 5 s-a-0. However this test also detects the faults 10 s-a-1, 11 s-a-1, 9 s-a-1 and 8 s-a-1, thus for a given test the D-Algorithm reveals all other faults that can also be detected by the same test. A shortcoming of the method is that considerable trial and error is involved in finding a test for a fault in networks with reconvergent fan-out and if a fault is not detectable all this effort is wasted.

3.2.4 Equivalent normal form method

This method was proposed by Armstrong (53) for the derivation of a nearly minimal fault-detection test set and uses the 'equivalent normal form' (enf) of a circuit. The enf of a circuit is obtained by tracing all paths from the circuit output to every circuit input while recording the gates through which the paths pass. The procedure is illustrated by deriving the enf of the hypothetical circuit of Fig.3.4. Starting with gate 4,

$$H = (f+g)_4$$

where the subscript 4 identifies the gate through which inputs f and g pass. Proceeding to the second level of gates

$$H = \left|(Ae)_2 + (eD)_3\right|_4 .$$

In a similar manner H can be expanded in terms of inputs to remaining gate level,

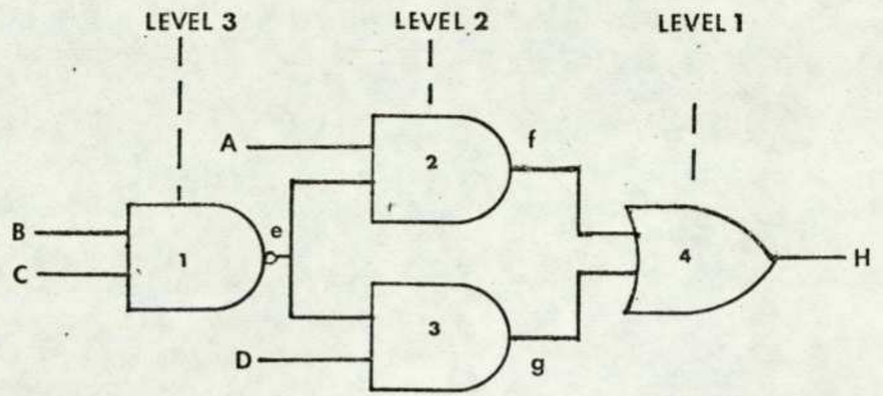


Fig.3.4 Example for enf method

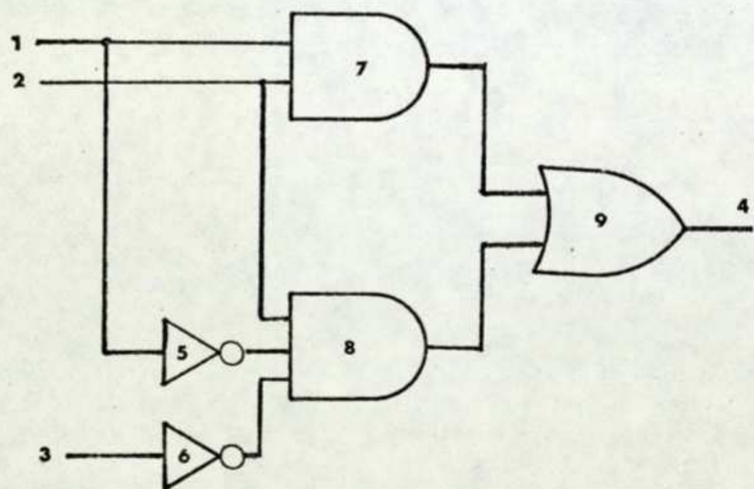


Fig. 3.5 Circuit example

$$H = \left|\left(A(\overline{BC})_1\right)_2 + \left((\overline{BC})_1 D\right)_3\right|_4$$

$$= \left|\left(A(\bar{B}_1+\bar{C}_1)\right)_2 + \left((\bar{B}_1+\bar{C}_1)D_3\right|_4$$

$$= \left|A_2(\bar{B}_{12}+\bar{C}_{12}) + (\bar{B}_{13}+\bar{C}_{13})D_3\right|_4$$

$$= \left|A_2\bar{B}_{12}+A_2\bar{C}_{12}+\bar{B}_{13}D_3+\bar{C}_{13}D_3\right|_4$$

$$= A_{24}\bar{B}_{124}+A_{24}\bar{C}_{124}+\bar{B}_{134}D_{34}+\bar{C}_{134}D_{34}$$

The output H is now expressed as a function of only the external inputs and as a result the enf of the circuit is obtained. Each subscripted input variable in the expression, referred to as a 'literal', specifies uniquely a path from the corresponding circuit input to the output.

Fault-detection tests for the enfs are constructed by selecting a set of literals whose paths contain every connection in the corresponding circuit and finding tests which check a single appearance of each literal for stuck-at-1 and stuck-at-0 faults. A literal is tested for stuck-at-1 by assigning 0 to it, 1's to all other literals in the term and suitable values to the other variables so that all other terms in the enf become 0. A literal is tested for stuck-at-0 by assigning 1 to all literals in the term and choosing suitable values which will make other terms in the enf equal to 0 . For example the literal $\bar{B}_{134}$ in the third term of H can be tested for s-a-0 by assigning 1 to $\bar{B}_{134}$ and D_{34} and to make other terms in the enf equal to 0, it is necessary to assign 0 to A and 1 to C. Hence $\bar{A}\bar{B}CD$ is the desired test; similarly $\bar{B}_{134}$ can be tested for s-a-1 by BCD. This method can be computer-programmed, but large number of literal appearances make it impracticable for larger circuits.

The work of Poage (57) in deriving a minimal test set for single and multiple faults is worth-mentioning here. He developed a special

calculus to describe faults in circuits but again his method is not practical because of the excessive computation time and storage.

3.2.5 Boolean difference

This method is unique in the sense that a test can be generated by computing Boolean functions. Boolean difference is defined as an exclusive-OR of two Boolean functions - one represents the fault-free circuit and the other the faulty circuit (58). For example, given a logic function $F(x_n)$ of variables $x_1,x_2,...x_i,...x_n$, the existence of a test for a fault affecting variable x_i is determined by the condition

$$F(x_1,x_2,...1...x_n) \oplus F(x_1,x_2,...0...x_n) = 1$$

The Boolean difference operator is represented by

$$\frac{d}{dx_i} .$$

If $\frac{dF(x_n)}{dx_i} = 1$, then a test for a fault affecting x_i exists but if $\frac{dF(x_n)}{dx_i} = 0$ then $F(x_n)$ is independent of x_i. The set of tests for a fault on x_i can be represented by the following expressions:

$$x_i \frac{dF(x_n)}{dx_i} \text{ for } x_i \text{ stuck-at-0}$$

and

$$\bar{x}_i \frac{dF(x_n)}{dx_i} \text{ for } x_i \text{ stuck-at-1}$$

Consider the circuit in Fig.3.1, and let the element 5(=h) be stuck-at-1. The output of the circuit is,

$$F(x_n) = (x_1.x_2)+x_3 = h+x_3.$$

For the fault h s-a-1 we have

$$\frac{dF(x_n)}{dh} = (1+x_3) \oplus x_3.$$

After some algebra (59)

$$\frac{dF(x_n)}{dh} = \bar{x}_3.$$

Tests for h s-a-1 are given by

$$\bar{h}.\frac{dF(x_n)}{dh} = \bar{h}\bar{x}_3 = \overline{x_1 x_2}\bar{x}_3.$$

Hence the fault 5 s-a-1 is detected by the test set

(000,010,100).

3.3. Fault Indistinguishability in Combinational Circuits

A fault in a digital network may said to be 'detectable' if the response of the network in the presence of the fault is different from that of the fault-free network to at least one set of input combinations. If the network has the same response for all possible input combinations in the presence of two distinct faults, then the faults are said to be 'indistinguishable' or functionally equivalent (60). For example in the circuit of Fig.3.5 faults 5 s-a-0 and 8 s-a-0 are detectable but indistinguishable because the output of the fault-free circuit for input combination 010 is different from that of the circuit in the presence of either of the faults. It is not possible to say which one of these actually contributed to the failed response. The indistinguishable faults for various gates are as follows (61):

Gate	Indistinguishable failures
AND	Any input s-a-0, output s-a-0
OR	Any input s-a-1, output s-a-1
NAND	Any input s-a-0, output s-a-1
NOR	Any input s-a-1, output s-a-0
INVERTER	Any input s-a-1(0), output s-a-0(1)

For networks having many cascaded gates, an indistinguishable fault may propagate through several levels of gates forming an equivalent class of faults. For example, in the circuit of Fig.3.5 if 3 is s-a-1, gate 6 will be s-a-0 which in turn will make gate 8 s-a-0. Faults 3 s-a-1, 6 s-a-0 or 8 s-a-0 are all indistinguishable and belong to the same fault equivalent class. The advantage of having a functionally equivalent class of faults is that if during the testing, access is available only to the primary input and output of the network then it is sufficient to test for a single representative of each class of faults (60). Since only one element of each equivalent class need to be considered for fault-simulation to verify the relevant test set, considerable reduction in simulation time may be obtained.

3.3.1 Use of fault-folding in deriving an indistinguishable fault set

A systematic approach for deriving the fault equivalent classes of a combinational circuit, termed as 'Fault - folding', has been described by Klin To (62). The central idea behind the process of 'Fault-folding' is the application of certain relations called, test equivalent, test implied, test cover between faults in order to reduce the number of faults that have to be considered in the actual test generation procedure. If T_i is a test-set for a set of faults F_i then two faults f_1 and f_2, where $f_1 \varepsilon F_1$ and $f_2 \varepsilon F_2$, are said to be 'test-equivalent', if a test for f_1 is also a test for f_2. Thus if t_1 and t_2 are the tests for f_1 and f_2 respectively then $t_1 \varepsilon T_1$ implies $t_1 \varepsilon T_2$ and vice-versa. They are 'test-implied' if $t_1 \varepsilon T_1$ implies $t_1 \varepsilon T_2$ but there is a test for f_2, $t'_2 \notin T_1$. Fig.3.6a shows that for a two input NAND gate, A s-a-0, B s-a-0, C s-a-1 are test equivalent and C s-a-0 is test implied by A s-a-1 or B s-a-1. Test relations for other gates are shown in Fig.3.6 -b, -c, and -d respectively.

A B C	As-a-0	As-a-1	Bs-a-0	Bs-a-1	Cs-a-0	Cs-a-1
0 0 1	no	no	no	no	TEST	no
0 1 1	no	TEST	no	no	TEST	no
1 0 1	no	no	no	TEST	TEST	no
1 1 0	TEST	no	TEST	no	no	TEST

a. NAND

A B C	As-a-0	As-a-1	Bs-a-0	Bs-a-1	Cs-a-0	Cs-a-1
0 0 1	no	TEST	no	TEST	TEST	no
0 1 0	no	no	TEST	no	no	TEST
1 0 0	TEST	no	no	no	no	TEST
1 1 0	no	no	no	no	no	TEST

b. NOR

A B C	As-a-0	As-a-1	Bs-a-0	Bs-a-1	Cs-a-0	Cs-a-1
0 0 0	no	no	no	no	no	TEST
0 1 0	no	TEST	no	no	no	TEST
1 0 0	no	no	no	TEST	no	TEST
1 1 1	TEST	no	TEST	no	TEST	no

c. AND

A B C	As-a-0	As-a-1	Bs-a-0	Bs-a-1	Cs-a-0	Cs-a-1
0 0 0	no	TEST	no	TEST	no	TEST
0 1 1	no	no	TEST	no	TEST	no
1 0 1	TEST	no	no	no	TEST	no
1 1 1	no	no	no	no	TEST	no

d. OR

Fig.3.6 Test relations for gates

The fault f_1 is said to 'test cover' f_2 if and only if f_1 is test-equivalent to f_2, or f_2 is test implied by f_1. If f_1 test covers f_2 then a test for f_1 is also a test for f_2 and it is sufficient to find a test for f_1 only. Starting from the network primary output, which is assumed to have an incorrect value, a set of test-covered related faults may be obtained by moving towards the primary inputs; in other words faults are 'folded' back from the primary output to the primary inputs. The reason for choosing the primary output as the initial failure site is that any detectable fault within the network will affect its value. During the folding operation, graphs that represent the testability relations of faults within the network can be formed, these graphs are termed as 'Fault-folded graphs'. The fault-folded graphs for the circuit of Fig.3.5 are shown in Fig.3.7. Test equivalent relations are identified by curly brackets and test-implied relations by one-directional arrows; inverters are identified by bi-directional arrows. In the case of non-redundant fan-out free networks, the fault-folded graphs are isomorphic to the network logic diagram i.e. one graph can be always be produced from the other and the logic diagram (62). In the following sections it will be shown how fault-folded graphs can be used to generate diagnostic test set for multiple-fault detection in combinational logic networks.

3.4 Multiple Faults in Combinational Logic Circuits

One of the assumptions normally made in test-generation schemes is that onlya single fault can occur. However as was explained before (section 2.4) this assumption is valid only if the network is frequently tested and if there is no redundancy in the network. A network may be classified as redundant if it contains undetectable faults (63). It is assumed here that more than one fault can occur in

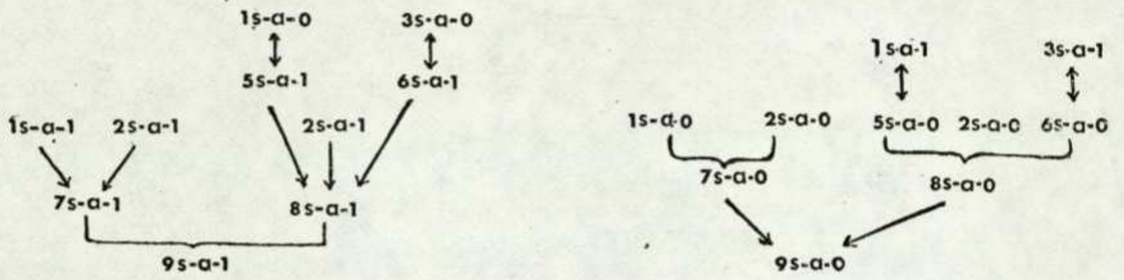


Fig 3.7 Fault-folded graphs for the circuit of fig. 3.5

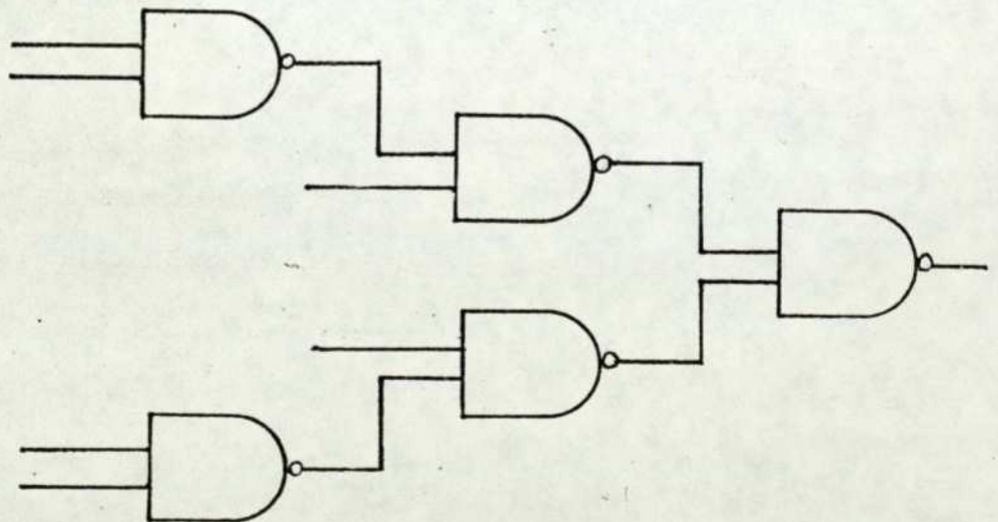


Fig 3.8 Multilevel fan-out free network

the network at the same time.

Before applying the fault folding method for multiple fault detection it is necessary to discuss some published results. Designing multiple fault detection tests for a logic network is difficult because of the extremely large number of faults that have to be considered. In a circuit having K-lines there are 2K possible single faults but a total of 3^k-1 multiple faults. Some authors (64) (65) (66) have considered the case of a test set derived with single fault assumption which can be used for multiple detection also. Schertz and Metze (67) have shown that for a "restricted fan-out free" network every single fault-detection test set is also a multiple fault detection test set. Any network which does not contain the network of Fig.3.8 (or its equivalent) as a subnetwork is called, following Schertz, 'a restricted fan-out free network'. Hayes (63) has proved that the network of Fig.3.8 is the smallest fan-out free network that can contain multiple faults which are not detected by every single fault detection test set and the multiplicy of the fault in that case is at least 4.

A single-output network is defined to be a fan-out free network (or tree network) if each line in the network is connected to only one gate input (68). When a network is not fan-out free it is possible to find a fan-out free equivalent of it by the following sequence of steps:

(a) Replace each primary input, which is connected to p gates of the network, by p primary inputs of the same name, so that each of these is connected to a single gate only.

(b) Replace any gate, which has s fan-outs, by s gates of the same type together with the subnetwork associated with the gate.

Kohavi and Kohavi (69) have shown that the test set designed

to detect all multiple faults in the fan-out free equivalent of a network, will also detect all detectable multiple faults in the original network.

Application of the fault-folding technique in conjunction with the results mentioned above produces a set of tests which detects whether or not there is any fault (single or multiple) in the network and locates the fault within an indistinguishable class (or classes if multiple fault).

3.5 An Algorithm for Deriving A Fault Detection Test Set For Combinational Circuits

It is helpful to commence by listing the sequences of steps:-

(1) Transform the given combinational network into equivalent single output networks (assuming a multi-output network) and make each of the networks fan-out free as described in the previous section.

(2) Draw up 'Fault-folded' graphs for each of the single-output networks.

(3) Trace a path from a primary input, in a fault-folded graph, towards the primary output and form a set of faults by including in the set all the faulty gates which are encountered in the process; in the case of a test equivalent relation, all the input faults to it belong to the same fault-set. Detailed explanations of this step will be given in the next section. This step may be repeated for every primary input in the corresponding fault-folded graph, until each fault is included in one set or other.

(4) Take the complements of the fault values at the primary inputs in a set. If all the primary inputs of the original

network are in the set, then this step gives a fault detection test which is said to be 'completely defined' and will detect all detectable faults in the set. If a test is not completely defined i.e. it contains x's at some of the primary inputs then it is necessary to fix the x values at 0 or 1 systematically in order to find a test for each element in the set. In other words the "Don't know" values in the incompletely defined test are not "Don't care" values and cannot arbitrarily be assigned with 0's and 1's. The test input values which can be derived from a fault-set will be termed as 'static' input and the combination of x values which together with 'static' inputs form a test for the fault set will be defined as 'control' input (70).

(5) Repeat step 4 for all the sets obtained in step 3.

(6) Apply steps 3 to 5 to the fault-folded graphs of each sub-network obtained in step 2.

(7) Apply steps 2 to 6 to each of the single output sub-network obtained by step 1. If all the faults in any shared portion of the original network, contained in any of the sub-networks have been tested at the corresponding sub-network outputs, then the shared portion need not be considered while testing other sub-networks.

(8) Select a test for each fault-set which detects the maximum number of faults in the set. A test which detects all faults in a set is defined to be a 'complete' test.

(9) If T is the set of tests obtained after applying step 8 and if F1 is a set of faults detected by T1ϵT and F2 is a set of faults detected by T2ϵT, then T1 can be removed from

the test set T if $F1 \subseteq F2$.

(10) The test set for the original network is obtained by uniting the sub-sets obtained in step 8 i.e. $T = \{T_i\}$ where i=1,2, 3....

In this way, a minimal fault detection test set for a combinational circuit can be derived. Since each test detects a set of indistinguishable faults, the detection set is also the fault location test set for the given network. Fig.3.9 presents the flowchart of the complete algorithm.

3.5.1 Application of the algorithm to generate a fault-detection test set for a combinational logic network

(1) The network of Fig.3.10a is transformed into two single fan-out free subnetworks as shown in Figs.3.10b and c.

(2) The fault-folded graphs for 14 s-a-0 and 14 s-a-1 are shown in Figs.3.11a and b respectively. Similar graphs for 15 s-a-0 and 15 s-a-1 are shown in Figs.3.12a and b respectively.

(3) Start from 2 s-a-0 on the left hand side of Fig.3.11b and proceed towards 14 s-a-1. The first element in the fault-set is 2 s-a-0, the next element encountered is 9 s-a-1. 9 s-a-1 is test equivalent to 2 s-a-0 and 3 s-a-0, hence 3 s-a-0 is included as the third element in the set. The next element encountered while moving from 9 s-a-1 to 14 s-a-1 is 12 s-a-0 which will be the fourth element of the set. The fifth element of the set 14 s-a-1 is test equivalent to 12 s-a-0 and 11 s-a-0. But 12 s-a-0 has already been included in the set, so 11 s-a-0 becomes the sixth element of the set. 11 s-a-0 is test implied by 9 s-a-1 or 10 s-a-1. If the forward tracing was done via

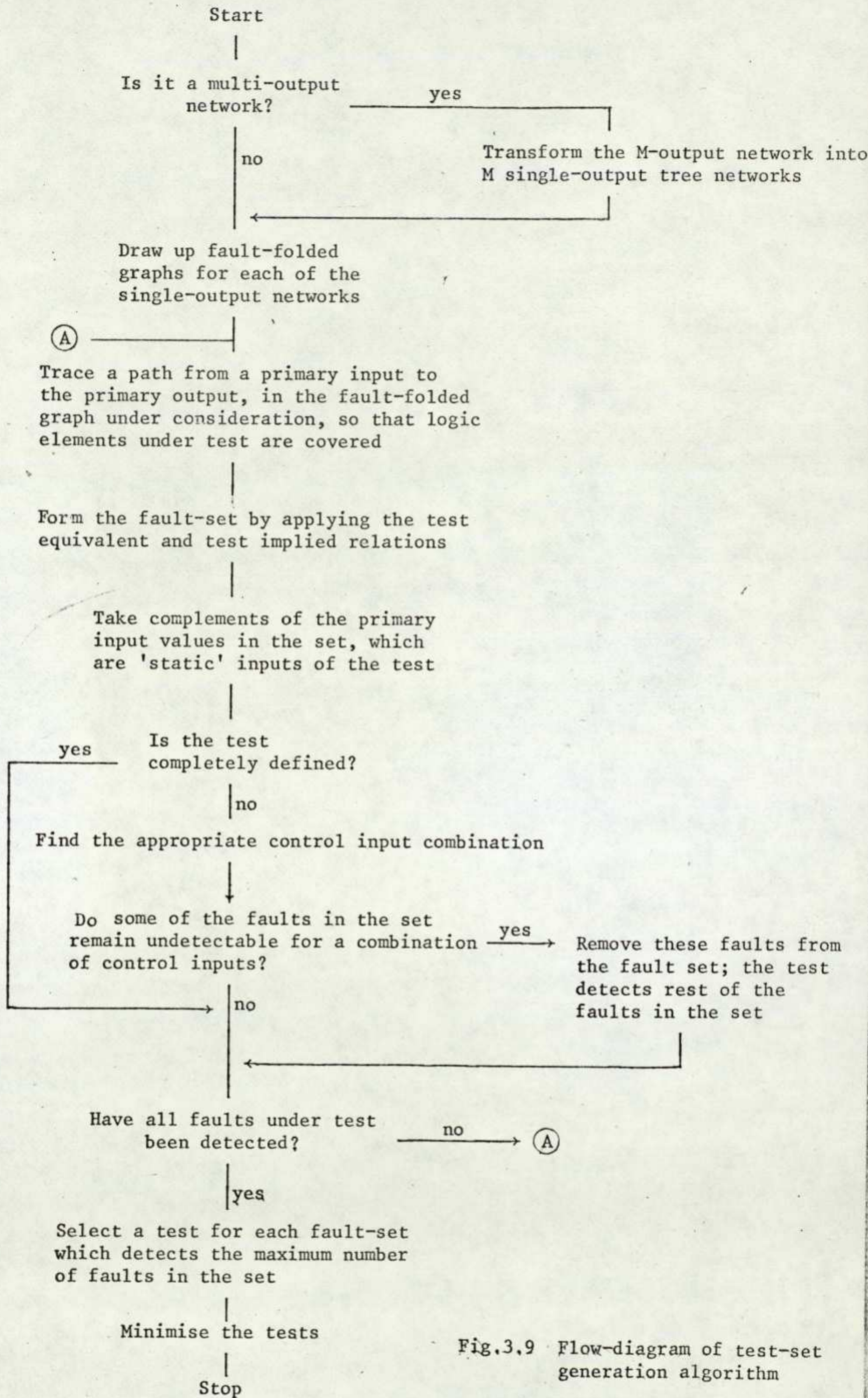
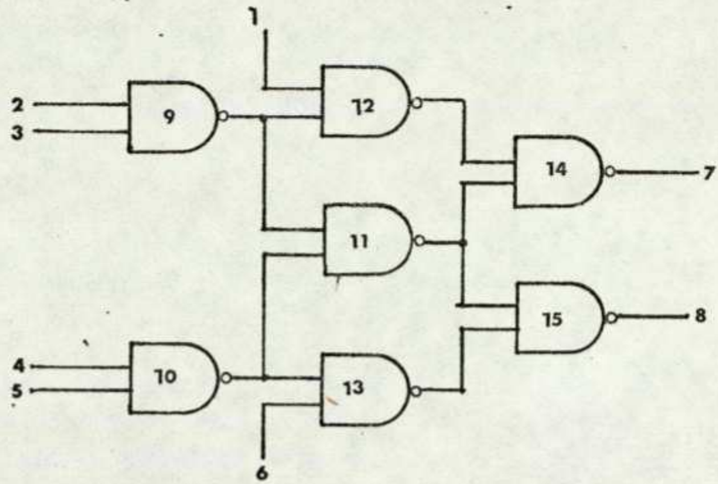
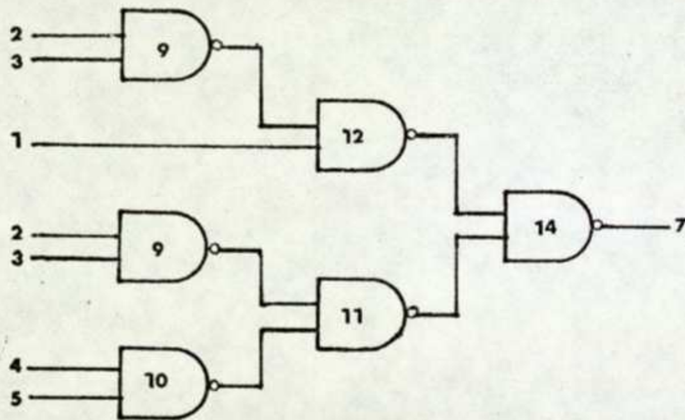


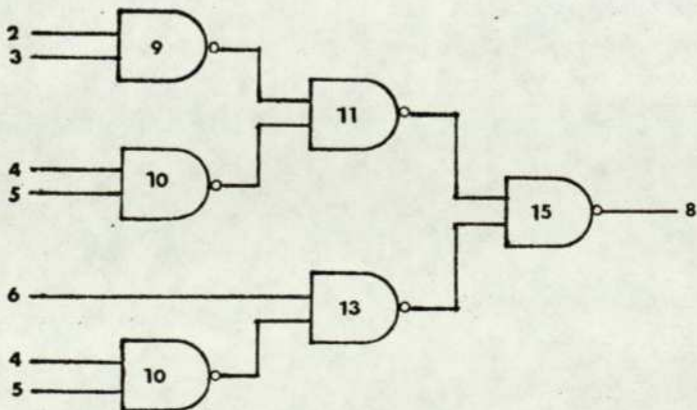
Fig.3.9 Flow-diagram of test-set generation algorithm



a. Circuit under test

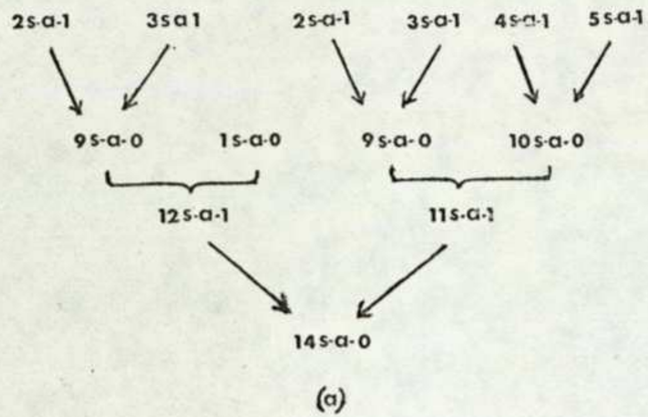


b. Sub-network (fan-out free) associated with output 7



c. Sub-network (fan-out free) associated with output 8

Fig 3.10



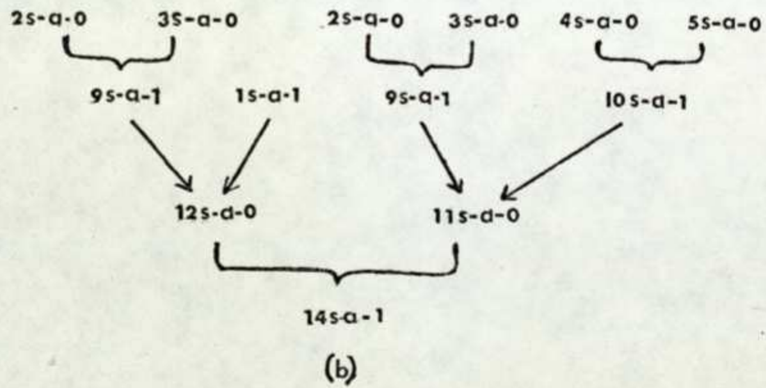
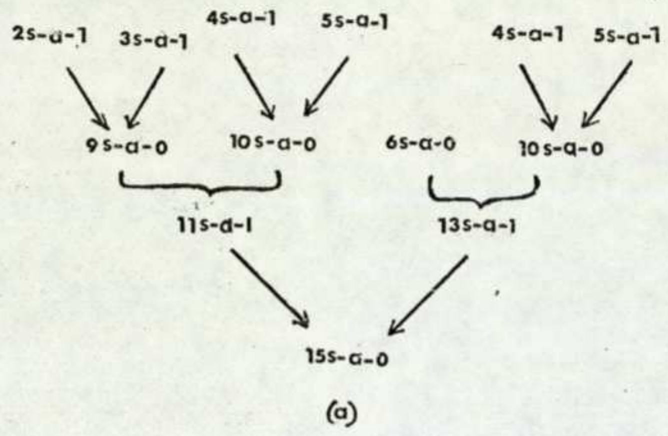
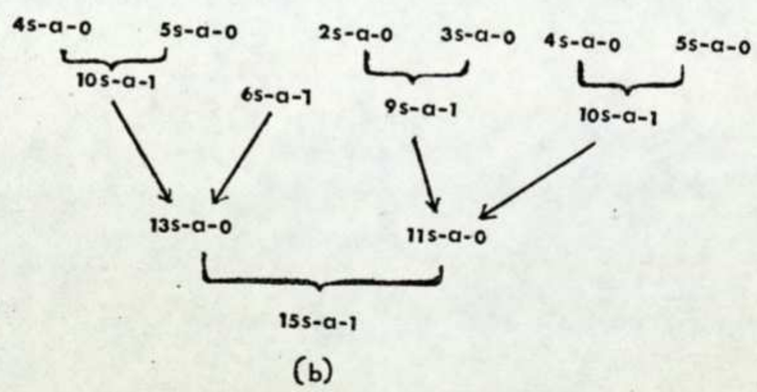


Fig. 3.11 Fault-folded graphs for the sub-network of Fig 3.10b.



(a)



(b)

Fig 3.12 Fault-folded graphs for the sub-network of Fig 3.10c.

9 s-a-1 then the fault set is complete because 9 s-a-1 has already been included in the fault-set and it is

{2 s-a-0, 9 s-a-1, 3 s-a-0, 12 s-a-0, 14 s-a-1, 11 s-a-0}

..... set A

However if it was decided to proceed via 10 s-a-1 instead of 9 s-a-1, the final set becomes

{2 s-a-0, 9 s-a-1, 3 s-a-0, 12 s-a-0, 14 s-a-1, 11 s-a-0
10 s-a-1, 4 s-a-0, 5 s-a-0} ..... set B

Incidentally this is the set which will be obtained if 5 s-a-0 is chosen as the starting primary input. When an element is stuck at a value which makes it test equivalent with the inputs which are feeding it then the sensitization procedure becomes : Forward-trace from a primary input to the primary output until an element which is test-equivalent to its inputs is encountered. In that case back-trace via each of the remaining inputs of the element towards the primary inputs.

For example, in the case of 14 s-a-1, which has two input elements 12 s-a-0 and 11 s-a-0, forward-trace from 2 s-a-0 via 12 s-a-0 but back-trace towards primary inputs via 11 s-a-0. If forward-tracing is started from 5 s-a-0 via 11 s-a-0 then back-tracing to the primary inputs has to be done by 12 s-a-0. As will be apparent from the fault-folded graph, this situation does not arise when the element encountered is stuck at a value which has test implied relations with its input elements. In this case forward tracing is continued via the next element.

If during a fault-set formation, an element is encountered,

while forward-tracing or back-tracing, which is already included in the set but has opposite stuck-at value to one in the set, then the process is terminated and step 3 is repeated with a different primary input. The reason for the termination is that two mutually exclusive elements (in this case an element with opposite stuck-at values) cannot belong to the same set, hence the fault-set is unrealizable. An example of this situation can be found in the fault-folded graph of the circuit of Fig.3.13a, shown in Fig.3.13b. If 4 s-a-1 is the first element of the set, then the construction proceeds as follows:

{4 s-a-1, 8 s-a-1, 9 s-a-1, 6 s-a-1, 1 s-a-0, 12 s-a-1
14 s-a-1, 13 s-a-1 .....}

It is now possible to proceed either via 10 s-a-1 or via 11 s-a-1. In the former case the rest of the elements are

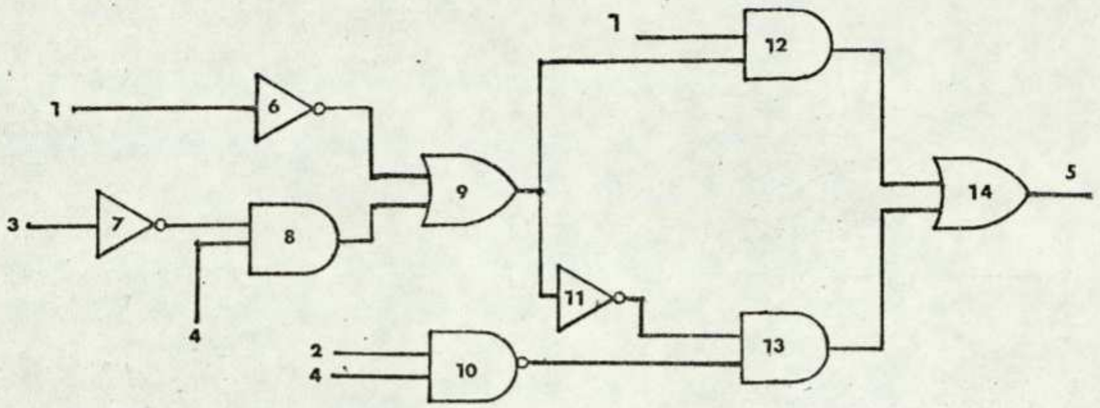
{..............., 10 s-a-1, 2 s-a-0, 4 s-a-0};

the set is not realizable because 4 s-a-1 and 4 s-a-0 cannot be included in the same set.
In the later case the set takes the form

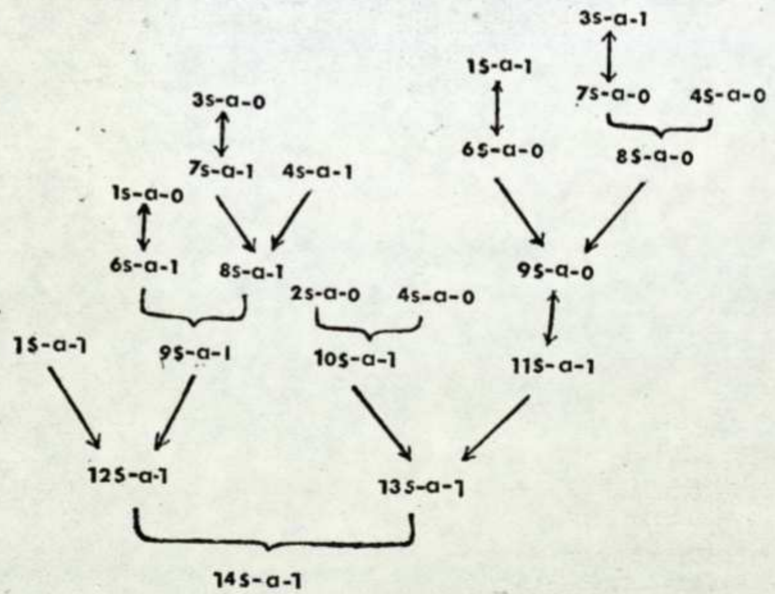
{................, 11 s-a-1, 9 s-a-0, ...}

Since 9 s-a-0 and 9 s-a-1 are mutually exclusive, this set also is not realizable.
In Fig.3.11a starting from 3 s-a-1 on the right-hand side the following fault-set is obtained

{3 s-a-1, 9 s-a-0, 11 s-a-1, 10 s-a-0, 4 s-a-1, 14 s-a-0}
..... set C



a. Circuit



b. Fault-folded graph of the circuit

Fig. 3.13

(4) and (5) In set A by complementing the stuck-at values of primary inputs the 'static inputs' of a possible test which will detect a fault within the set are found.

Primary inputs	1	2	3	4	5
Test value	x	1	1	x	x

where x is 'unknown'.

Since the test is not completely defined it is necessary to find a combination of 'control inputs' which together with the 'static inputs' will detect all detectable faults in the set. Four primary inputs (1,4,5 and 6) have no test values assigned to them, hence there can be 16 ($=2^4$) combinations of control inputs,

Primary inputs	1	2	3	4	5	6
Test value	0	1	1	0	0	0
	0	1	1	0	0	1
	0	1	1	0	1	0
	.	.	.	.	.	.
	.	.	.	.	.	.
	.	.	.	.	.	.
	1	1	1	1	1	0
	1	1	1	1	1	1

It so happens that 011000 is a test which detects all the faults in set A i.e. all x's are 0's. Similarly in set B, 111111 or 111110 detects all faults except 10 s-a-1 which incidentally is not detected at primary output 14 and in set C, 000000 is a test which detects all faults in the set.

(7) and (8) After the application of the steps the following result is obtained:

Primary output	Test	Faults detected
14	011000	9 s-a-0, 12 s-a-0, 14 s-a-1, 11 s-a-0
	000000	9 s-a-0, 10 s-a-0, 11 s-a-1, 14 s-a-0
	100110	9 s-a-0, 12 s-a-1, 14 s-a-0
15	111111	11 s-a-0, 15 s-a-1, 13 s-a-0, 10 s-a-1
	011000	13 s-a-0, 15 s-a-1, 11 s-a-0, 9 s-a-1
	000000	9 s-a-0, 11 s-a-1, 10 s-a-0, 15 s-a-0
	011001	13 s-a-1, 10 s-a-0, 15 s-a-1

(9) Since none of the fault-sets above is a sub-set of any other fault-set in the group, this step is not applicable.

(10) After the union of the test sets for output 14 and 15, the final test set for the network of Fig.3.10a is

{011000, 000000, 100110, 111111, 011001}.

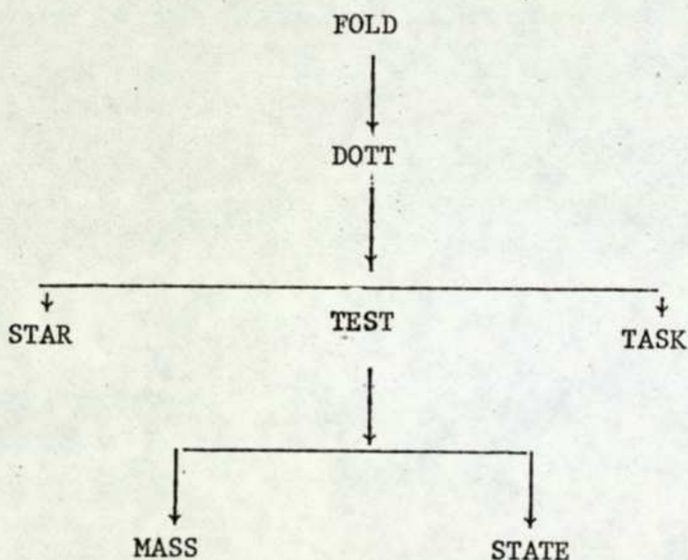
Thus for the six-input combinational circuit only five tests are needed to detect all single and multiple faults. Since each test also locates faults within an equivalent class, the test set can be used for the diagnostic testing of the combinational circuits.

3.5.2 Computer simulation of the algorithm

The algorithm described in Fig.3.9 has been programmed on an ICL 1905E computer. It was realised while applying the algorithm for test generation that for larger circuits it would be time-consuming and extremely tedious to make the circuit fan-out free for each output and to draw the fault-folded graphs for each fan-out free network. The programme has been designed in such a way so that these two tasks would be automatically taken care of. The programme has been divided into two subprogrammes; the first subprogramme reads in the relevant

data and then automatically generates tests which at the same time are verified as well. Once a test set for the network is obtained, the second subprogramme checks for any redundant test in the set and removes it, thus generating the minimal test set for the network. If the absolute minimum test set is not desired then subprogramme 2 can be ignored.

SUBPROGRAMME 1. This consists of a main routine FOLD which calls in the subroutine DOTT. Subroutine DOTT, in turn, has three subroutines TEST, STAR and TASK. Subroutine TEST has two subroutines of its own, they are MASS and STATE. The structure of sub-programme 1 is shown below:



FOLD : This programme reads in the data input and traces back the desired path from the output to an input, at the same time recording the logical state and function code of the elements. In other words the assumed fault at the output is folded back to the input via the intervening elements. The flow-diagram is shown in Fig.3.14, the arrays and variables used have the following meaning and uses

MDATA — a two-dimensional array which stores the code number of m gates encountered while back-tracing, for each of the n paths, from a primary output to a primary input.

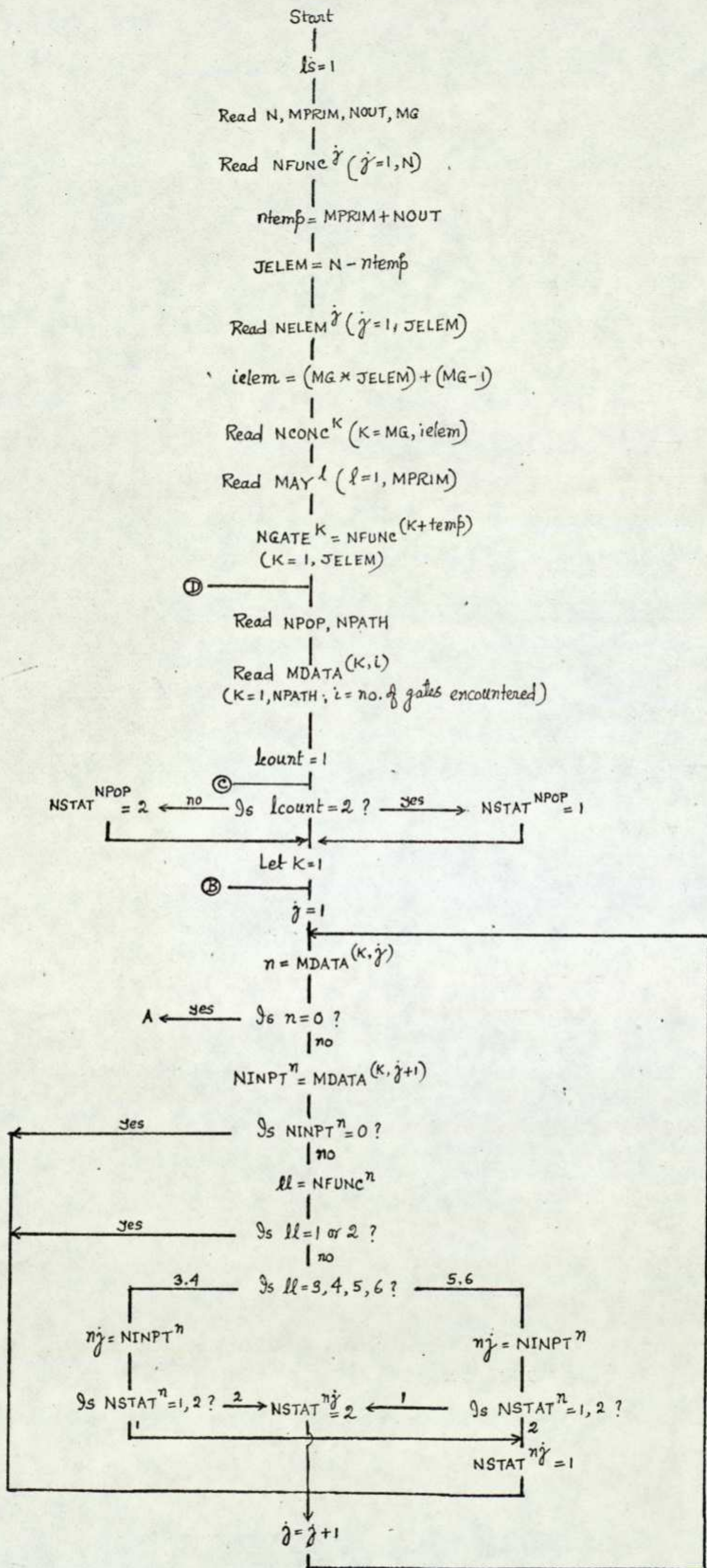
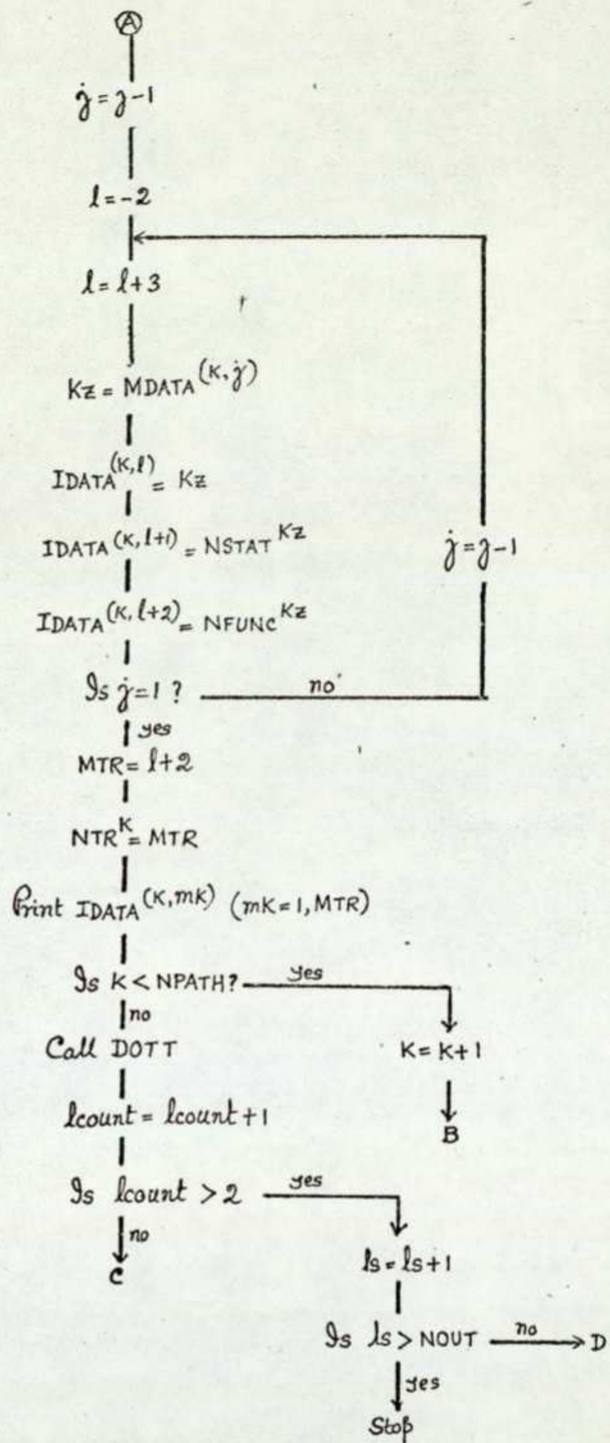


Fig 3.14 Flow-diagram of FOLD

cont'd



A
j = j-1
l = -2
l = l+3
Kz = MDATA(K, j)
IDATA(K,l) = Kz
IDATA(K, l+1) = NSTAT Kz
IDATA(K, l+2) = NFUNC Kz
Is j=1 ?
no
j=j-1
yes
MTR = l+2
NTR K = MTR
Print IDATA(K, mK) (mK = 1, MTR)
Is K < NPATH?
yes
K = K+1
B
no
Call DOTT
lcount = lcount+1
Is lcount > 2
yes
no
C
ls = ls+1
Is ls > NOUT
no
D
yes
Stop

NPOP	primary output under test.
N	total number of elements in the circuit.
MPRIM	number of primary inputs in the circuit.
NOUT	number of primary outputs in the circuit.
JELEM	number of gates in circuit.
MG	maximum number of inputs any gate in the circuit has.
NPATH	number of paths, from a primary output to the primary inputs, to be tested.
NFUNC	a one-dimensional array used to store function codes of the N elements in the circuit.
NELEM	a one-dimensional array of NELEM(I) variables which stores the code numbers of the gate elements in the circuit.
NCONC	a one-dimensional array of NCONC(I) arrays which stores the input connections of the elements in NELEM (Note : input connections to jth element of NELEM are stored in k^{th}(=jxMG) and $(k+MG-1)^{th}$ positions of NCONC. Non-existing input connections for an element are substituted by zero's).
MAY	a one-dimensional array MAY(I) variables used for storing the primary input numbers of the circuit.
NSTAT	a one-dimensional array of NSTAT(I) variables which stores the logical states of the circuit elements.
NGATE	a one-dimensional array of NGATE(I) variables which stores function codes of the gate elements in the circuit.
NINPT	a one-dimensional array of NINPT(I) variables used for storing the input connections of a gate element

during the folding operation.

IDATA a two-dimensional array of IDATA(K,M) variables which is used to store the code number of an element, its logical state and function code in that order for the kth path, in the m positions.

MTR an integer variable used to hold the number of elements in each of the k paths of array IDATA.

NTR a one-dimensional array of NTR(I) variables which is used for storing MTR for each of the K paths of array IDATA.

DOTT : This subroutine uses the array IDATA of programme FOLD to form an indistinguishable fault-set and derives the static input value of the test used to detect the faults. In other words it carries out step 3 and part of step 4 of the algorithm with the help of subroutine TEST. The flow-diagram is shown in Fig.3.15.

TEST : This subroutine decides whether an element encountered (while proceeding from a primary input to the primary output) has test equivalent or test implied relations with its input elements and calls subroutines MASS and STATE respectively. It also permits sensitization of paths from the inputs of the test-equivalent elements to the primary inputs. If during the fault-set formation an inconsistency develops e.g. same element with different logic states, then it returns to subroutine DOTT which prints a diagnostic error message 'ILLEGAL COMBINATION IN SET' and proceeds with the next path under test. Flow-diagrams of sub-routines TEST, MASS and STATE are shown in Figures 3.16, 3.17 and 3.18 respectively. The arrays and variables used in DOTT, TEST, MASS and STATE have following uses:

NFLAG an integer variable which takes on the value of 1

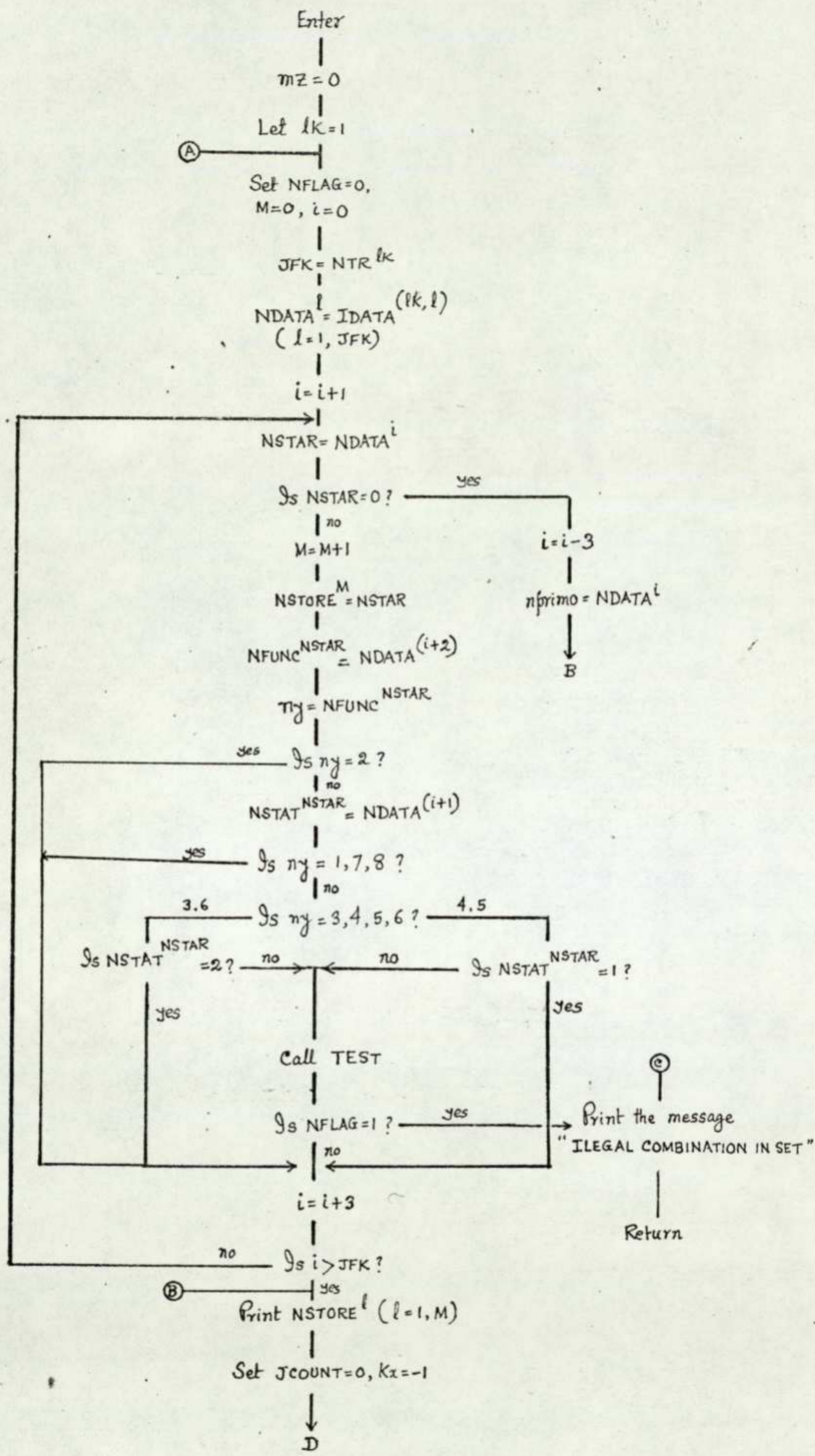
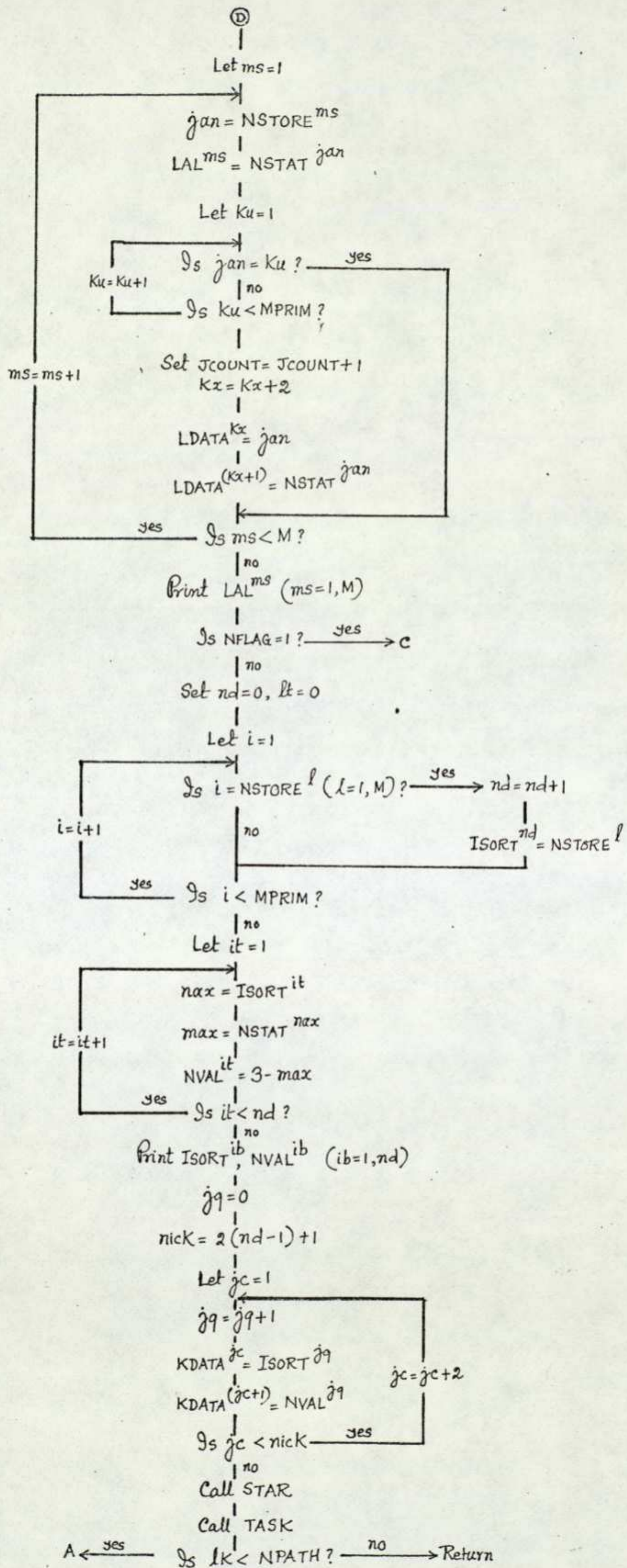


Fig 3.15 Flow-diagram of DOTT

cont'd


D
Let ms=1
jan = NSTORE^ms
LAL^ms = NSTAT^jan
Let Ku=1
Is jan = Ku ?
yes
no
Ku=Ku+1
Is Ku < MPRIM ?
Set JCOUNT= JCOUNT+1
Kx = Kx+2
ms=ms+1
LDATA^Kx = jan
LDATA^(Kx+1) = NSTAT^jan
yes
Is ms < M ?
no
Print LAL^ms (ms=1,M)
Is NFLAG=1 ?
yes
C
no
Set nd=0, lt=0
Let i=1
Is i = NSTORE^l (l=1,M) ?
yes
nd=nd+1
i=i+1
no
ISORT^nd = NSTORE^l
yes
Is i < MPRIM ?
no
Let it=1
nax = ISORT^it
max = NSTAT^nax
it=it+1
NVAL^it = 3- max
yes
Is it < nd ?
no
Print ISORT^ib, NVAL^ib (ib=1,nd)
jq=0
nick= 2(nd-1)+1
Let jc=1
jq=jq+1
KDATA^jc = ISORT^jq
jc=jc+2
KDATA^(jc+1) = NVAL^jq
Is jc < nick
yes
no
Call STAR
Call TASK
A
yes
Is lk < NPATH ?
no
Return

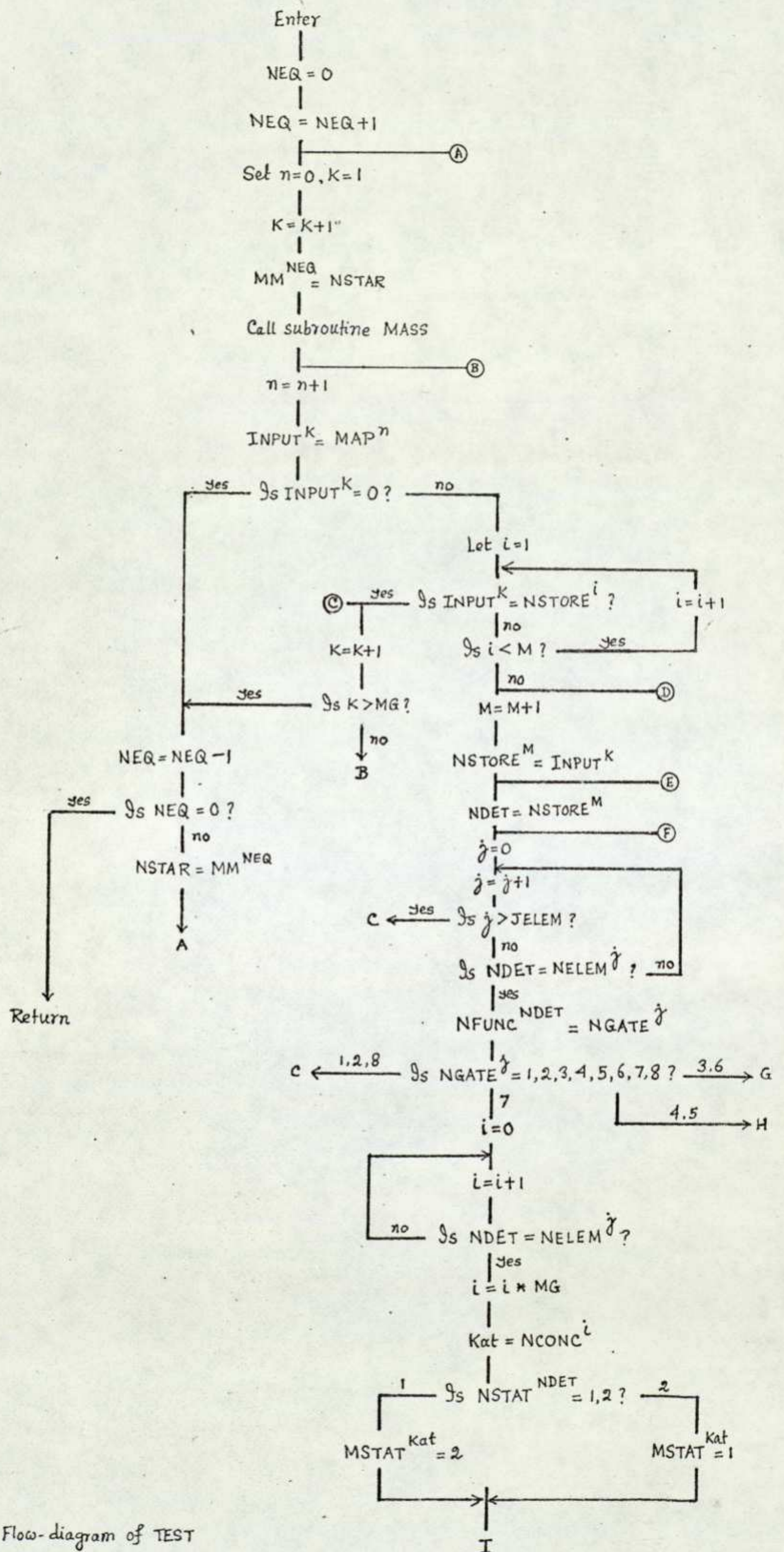
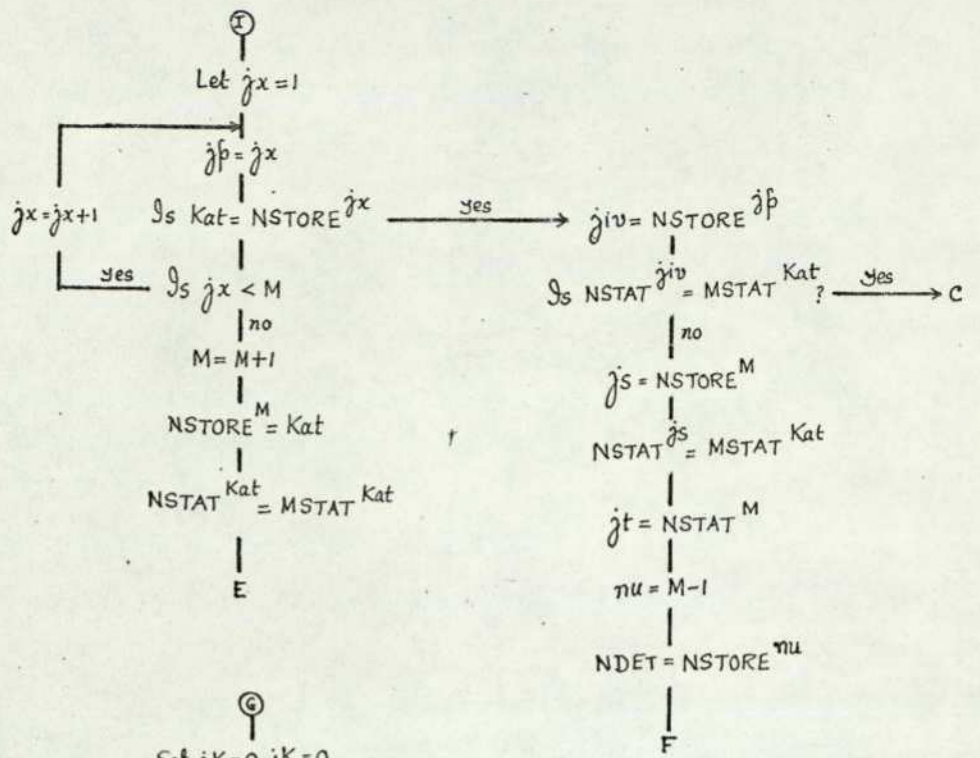
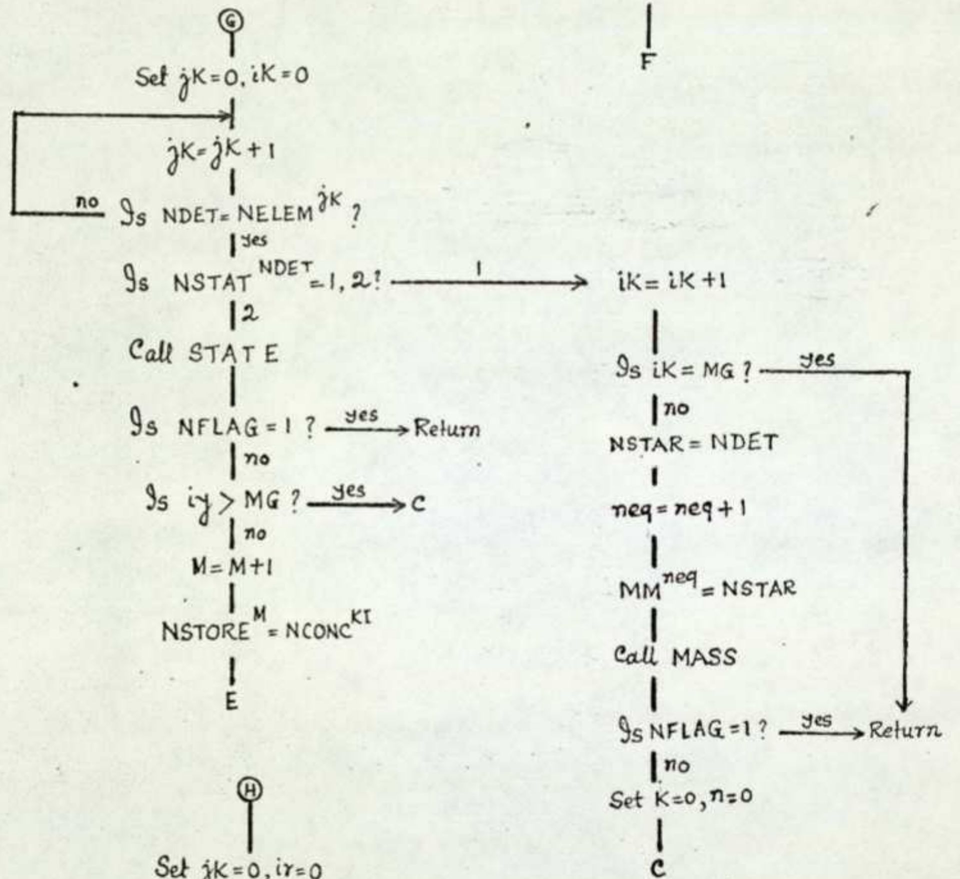


Fig 3.16 Flow-diagram of TEST

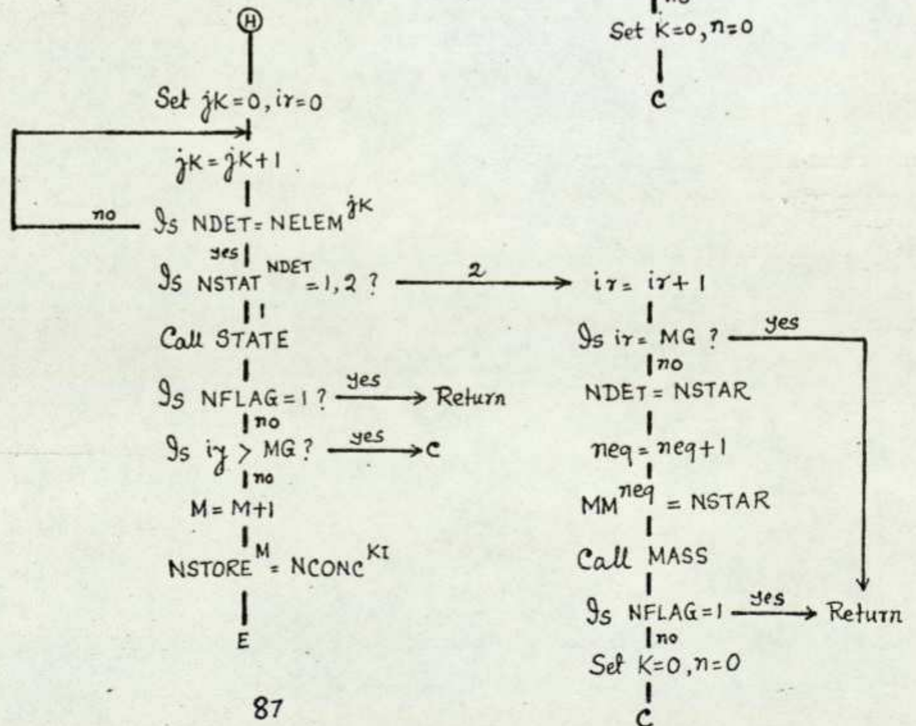
cont'd



I
Let jx = 1
jp = jx
jx = jx+1
Is Kat = NSTORE^jx
yes
jiv = NSTORE^jp
yes
Is jx < M
Is NSTAT^jiv = MSTAT^Kat ?
yes
C
no
M = M+1
no
js = NSTORE^M
NSTORE^M = Kat
NSTAT^js = MSTAT^Kat
NSTAT^Kat = MSTAT^Kat
jt = NSTAT^M
E
nu = M-1
NDET = NSTORE^nu
F



G
Set jK = 0, iK = 0
jK = jK+1
no
Is NDET = NELEM^jK ?
yes
Is NSTAT^NDET = 1, 2?
1
iK = iK+1
2
Call STATE
Is iK = MG ?
yes
Is NFLAG = 1 ?
yes
Return
no
no
NSTAR = NDET
Is iy > MG ?
yes
C
neq = neq+1
no
M = M+1
MM^neq = NSTAR
NSTORE^M = NCONC^KI
Call MASS
E
Is NFLAG = 1 ?
yes
Return
no
Set K=0, n=0
C



H
Set jK = 0, ir = 0
jK = jK+1
no
Is NDET = NELEM^jK
yes
Is NSTAT^NDET = 1, 2 ?
2
ir = ir+1
1
Call STATE
Is ir = MG ?
yes
no
Is NFLAG = 1 ?
yes
Return
NDET = NSTAR
no
Is iy > MG ?
yes
C
neq = neq+1
no
M = M+1
MM^neq = NSTAR
NSTORE^M = NCONC^KI
Call MASS
E
Is NFLAG = 1
yes
Return
no
Set K=0, n=0
C

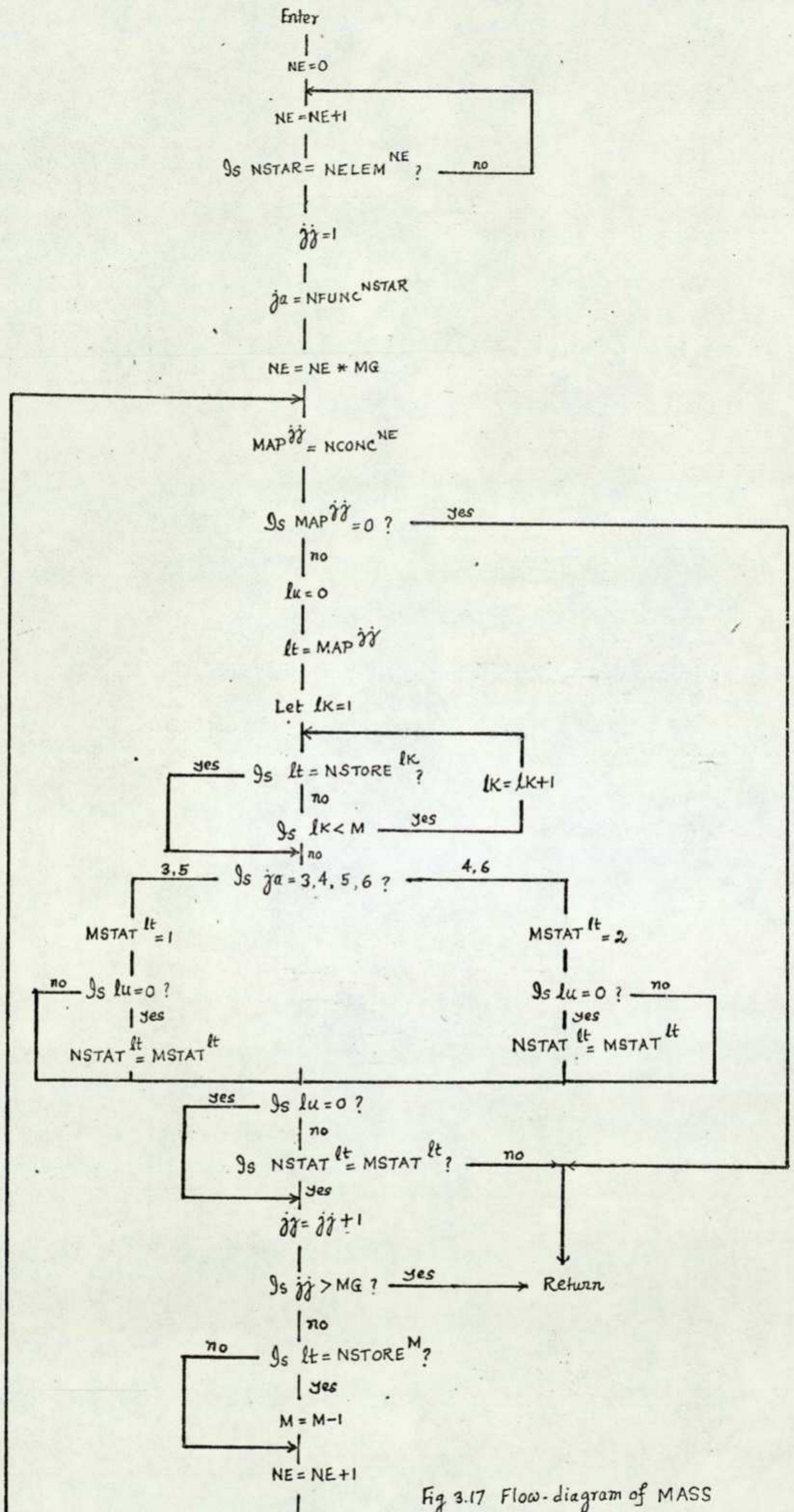


Fig 3.17 Flow-diagram of MASS

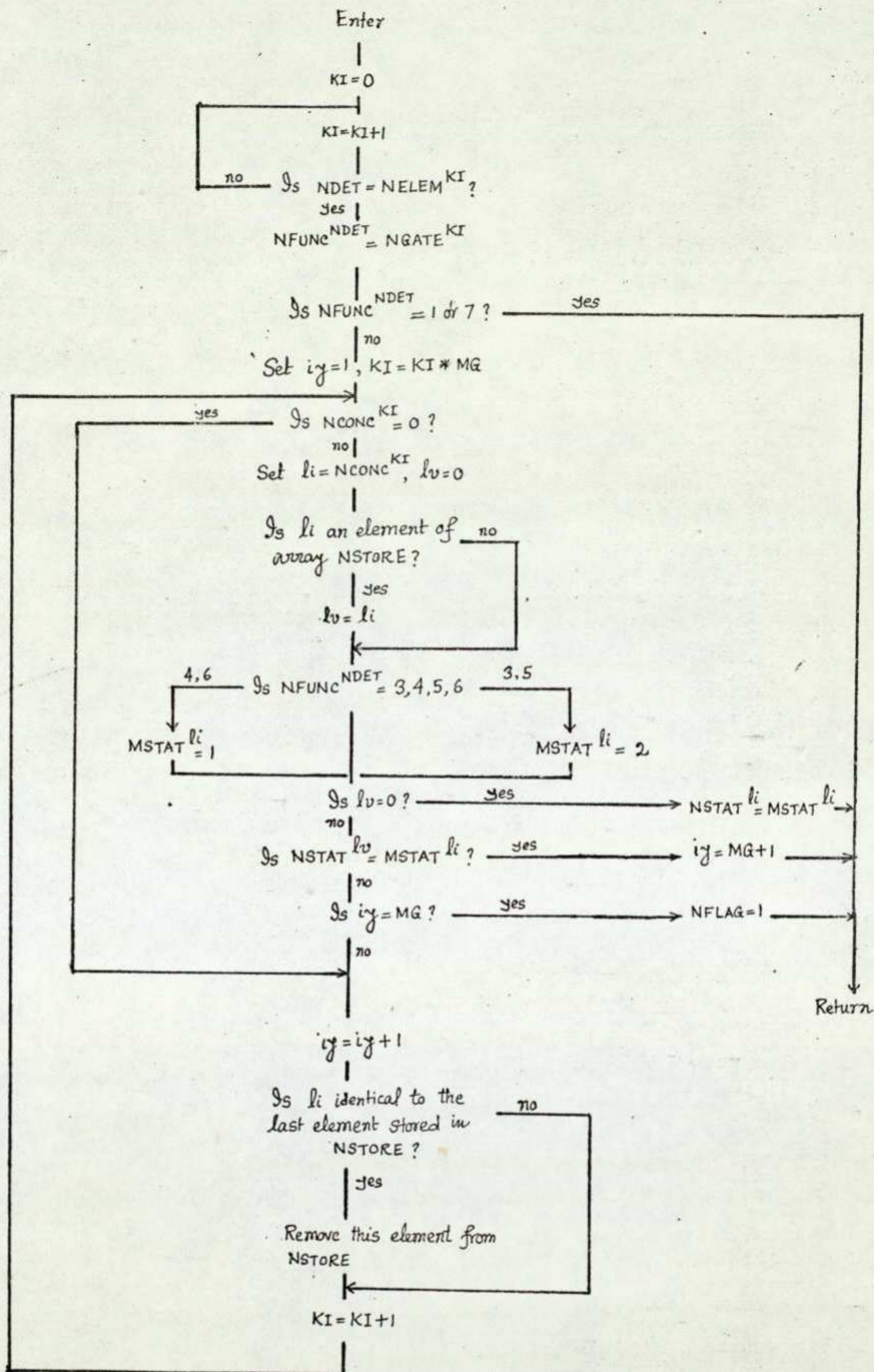


Fig 3.18 Flow-diagram of STATE

when an inconsistency is found, otherwise it is equal to '0'.

NDATA — a one-dimensional array of variables NDATA(I) which is used to store the elements of array IDATA.

JFK — an integer variable which holds the number of elements in NDATA.

NSTAR — an integer variable which holds the value of an element while forward-tracing.

M — an integer variable which counts up the number of elements in the fault-set.

NSTORE — a one-dimensional array of variables NSTORE(I) which is used to store the elements of a fault-set.

LAL — a one-dimensional array of variables LAL(I) which stores the logical states of the elements in array NSTORE.

LDATA — a one-dimensional array of variables LDATA(I); the ith(=1,3,5,...) and the (i+1)th positions are used to store a gate element of the fault-set and its logical state respectively.

JCOUNT — an integer variable which holds the number of gate elements in LDATA.

MM — a one-dimensional array of variables MM(I) which stores the gates whose test equivalent inputs are to be determined.

MAP — a one-dimensional array of variables MAP(I) which stores input connections of test-equivalent elements.

NE,KI — two integer variables which point to the input

connections of an element stored in array NCONC during the execution of subroutine MASS and STATE respectively.

MSTAT a one-dimensional array of variables MSTAT(I) which stores the logical states of the input connections of an element.

NEQ an integer counter used to point to the locations of array MM.

NDET an integer variable which holds on the value of an element during the back-tracing phase.

ISORT a one-dimensional array of variables ISORT(I) which stores the primary input elements of a fault-set.

NVAL a one-dimensional array of variables ISORT(I) which stores the static test input values of a fault set.

KDATA a one-dimensional array of variables KDATA(I); the ith(=1,3,5...) and the (i+1)th positions are used to store a static primary input and its logical state respectively.

STAR : This subroutine uses the array KDATA of subroutine DOTT to calculate all possible control input combinations and stores them in array NTV. The flow-diagram is shown in Fig.3.19. The arrays and variables used in the subroutine are

LAY a one-dimensional array of variables LAY(I) which stores the ith(=1,3,5...) element of array KDATA.

NUM an integer variable holding the number of primary inputs having unknown logic values.

ILA an integer variable holding the total number of control input combinations.

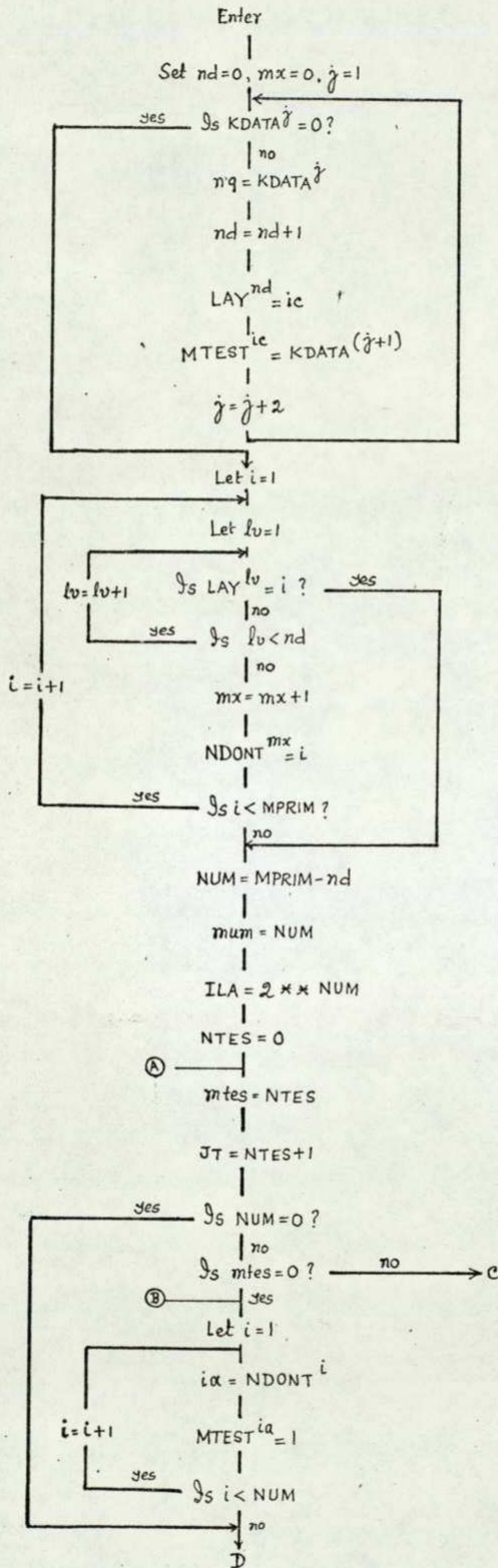
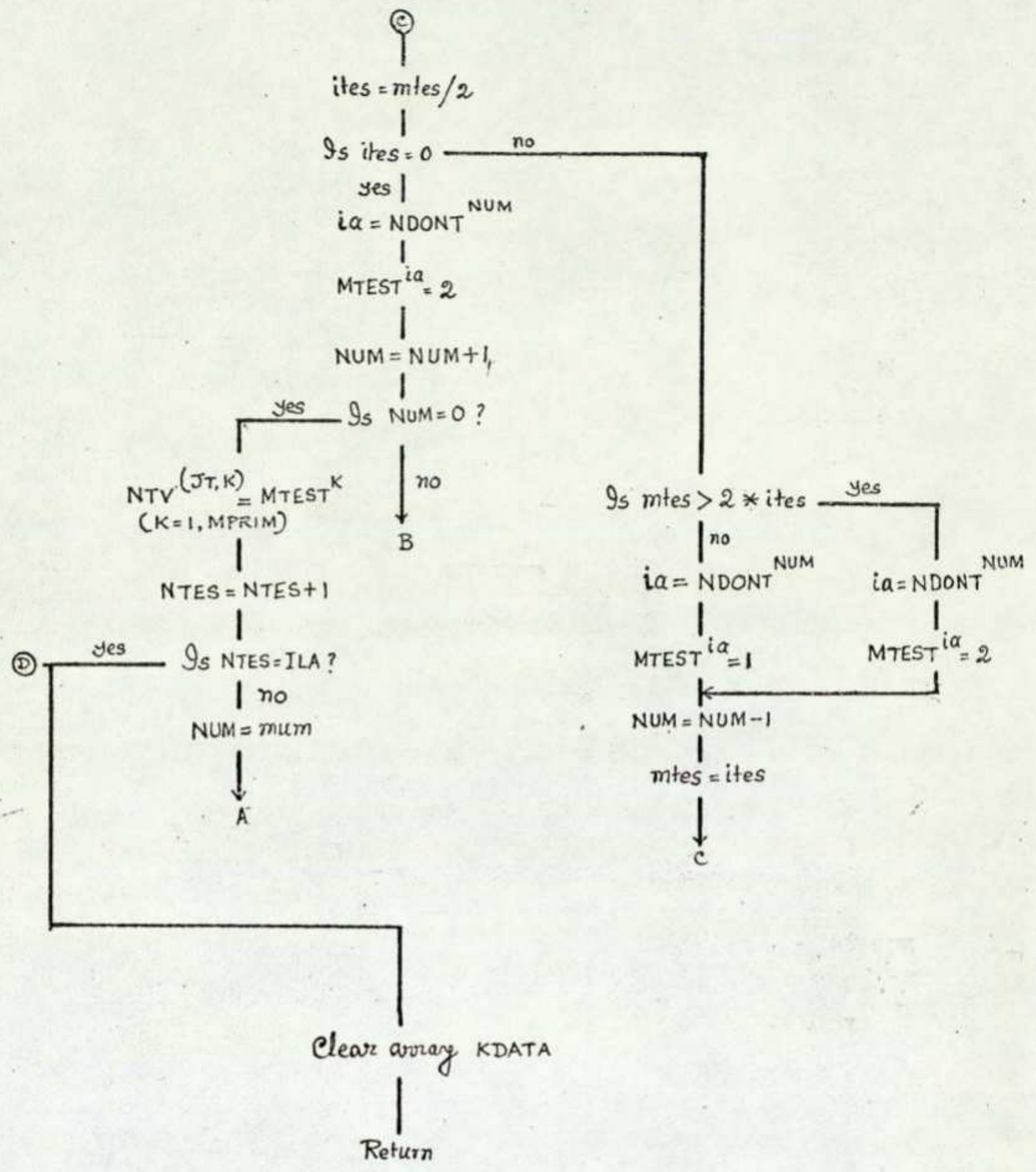


Fig 3.19 Flow-diagram of STAR

cont'd


C
ites = mtes/2
Is ites = 0
no
yes
ia = NDONT NUM
MTEST ia = 2
NUM = NUM+1
yes
Is NUM = 0 ?
NTV (JT,K) = MTEST K
(K=1, MPRIM)
no
B
Is mtes > 2 * ites
yes
no
ia = NDONT NUM
ia = NDONT NUM
NTES = NTES+1
D
yes
Is NTES = ILA ?
no
NUM = mum
A
MTEST ia = 1
MTEST ia = 2
NUM = NUM-1
mtes = ites
C
Clear array KDATA
Return

NDONT — a one-dimensional array of variables NDONT(I) which stores the primary input elements that are not stored in array KDATA.

MTEST — a one-dimensional array of variables MTEST(I) which are used to store the test input values.

NTES,JT — integer variables which take on control input combinations during the execution of the subroutine.

NTV — a two-dimensional array of variables, NTV(M,N) which store mth test input, in n positions.

<u>TASK</u> : This subroutine simulates each gate fault in a fault-set stored in array LDATA of subroutine DOTT in the presence of each test stored in array NTV of subroutine STAR. If a test detects all faults in the set, control returns to subroutine DOTT, otherwise the process is continued until all the tests in array NTV have been exhausted. If a particular fault or faults cannot be detected by any test in NTV, then the conclusion is that the fault or faults do not belong to the fault set under test. The evaluation of function routines of circuit elements and the fault-simulation procedure have been described in sections 2.3.2 and 2.4.2 respectively. The function routines have been slightly modified for deriving the input connections of a gate element in this subroutine; the modified version of function evaluation routine is illustrated with the flow-diagram of an AND function in Fig.3.20. The flow-diagram of subroutine TASK is shown in Fig.3.21.

The arrays and variables used have the following uses:

NFG — a one-dimensional array of variables NFG(I) which stores the gate elements of the fault-set.

NSV — a one-dimensional array of variables NSV(I) which is used to store the logical states of the elements

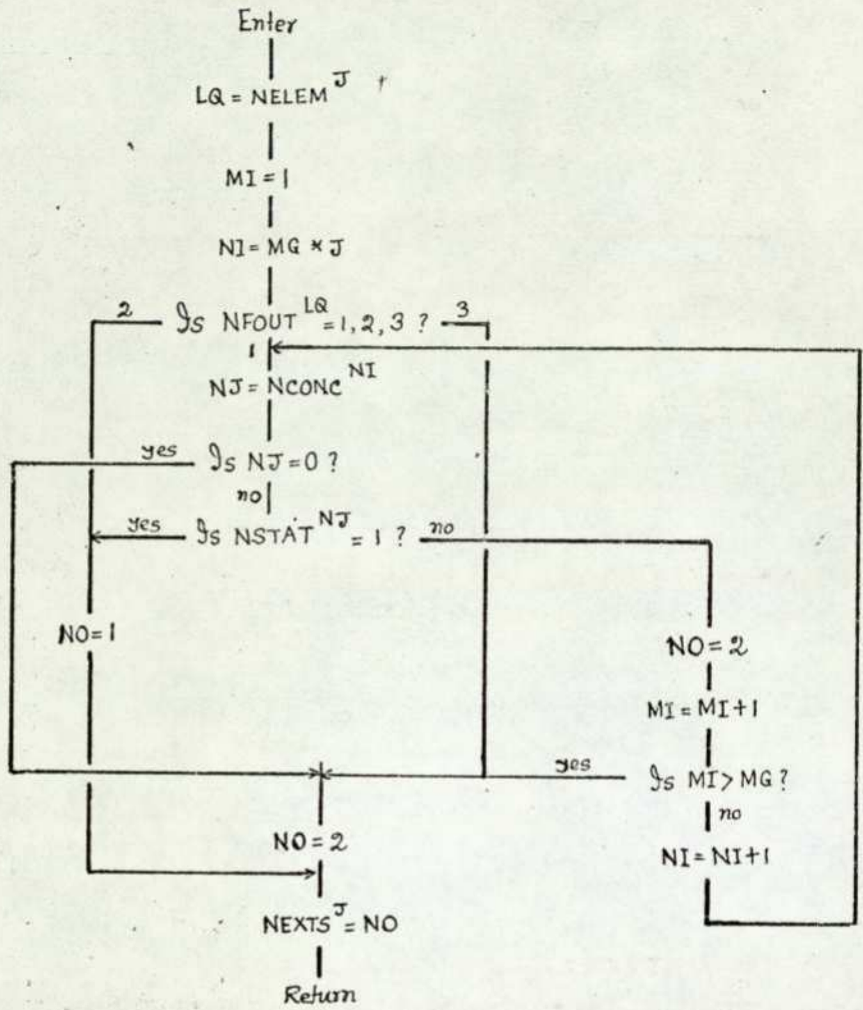


Fig 3.20 Flow-diagram of modified AND function.

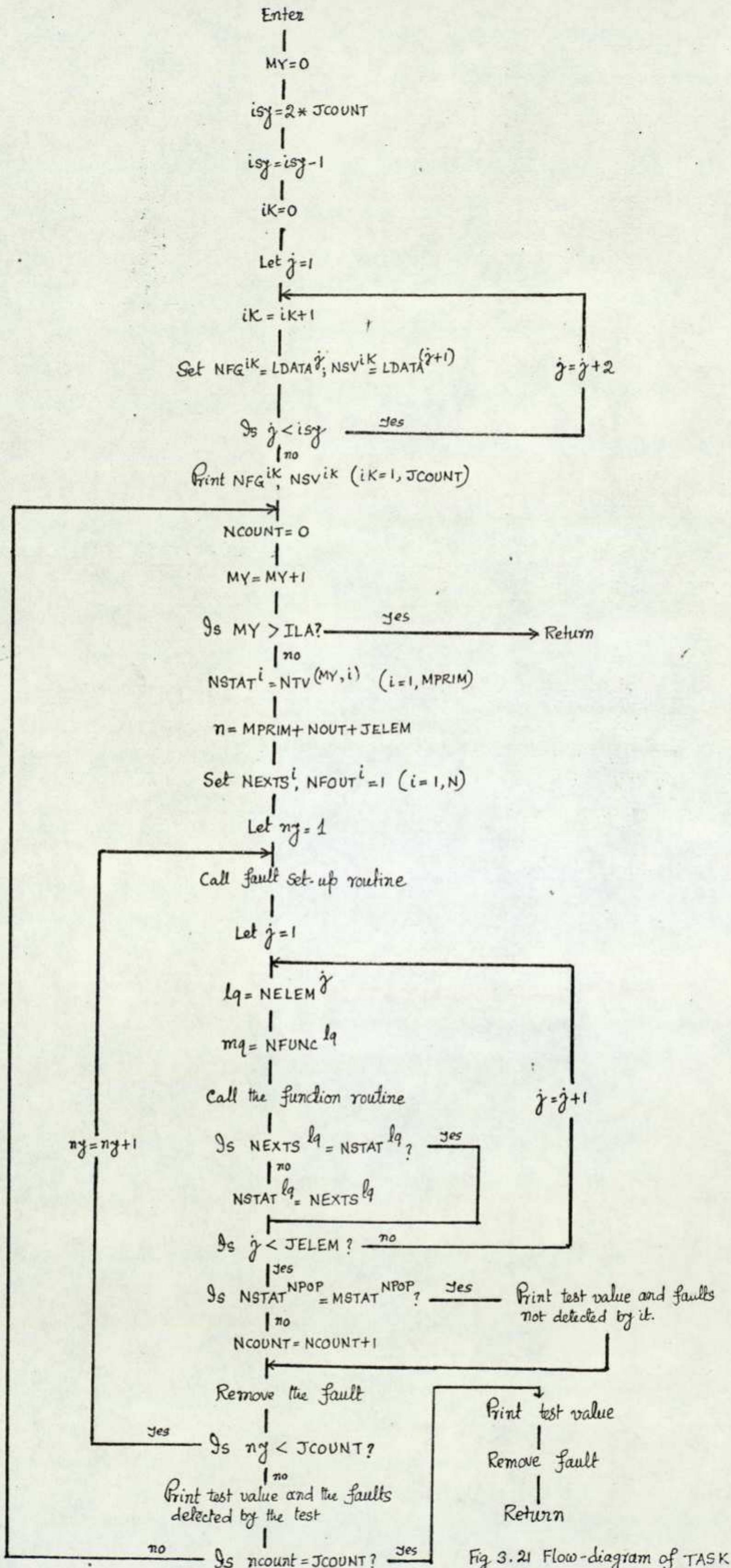


Fig 3.21 Flow-diagram of TASK

stored in NFG.

NCOUNT	an integer counter used to count up the number of faults in the set.
MY	an integer counter used to count up the number of tests being applied.
NG,NFV	integer variables used to hold the faulty gate element and its logic value respectively.
IFALT,N1STR, NFOUT	one-dimensional arrays whose functions and uses have been explained in section 2.4.2.
LQ	an integer variable used to hold the gate element number.
NI	an integer variable used to point to the input connections of a gate element, stored in array NCONC.
NEXTS	a one-dimensional array whose function has been explained in section 2.3.1.

The data input required by programme FOLD for the automatic generation of a test set to detect each fault in the circuit of Fig.3.25 is shown in Fig.3.29.

SUBPROGRAMME 2. This consists of a main routine, named MINI, which carries out step 9 of the algorithm. The flow-diagram is shown in Fig.3.31. The arrays and variables used in the programme have the following definition and uses:

NX	total number of tests/fault sets.
MPRIM	number of primary inputs in the circuit.
NPRIM	a one-dimensional array of variables NPRIM(I) which stores the primary inputs of the circuit.
NCOUNT	a one-dimensional array of variables NCOUNT(I)

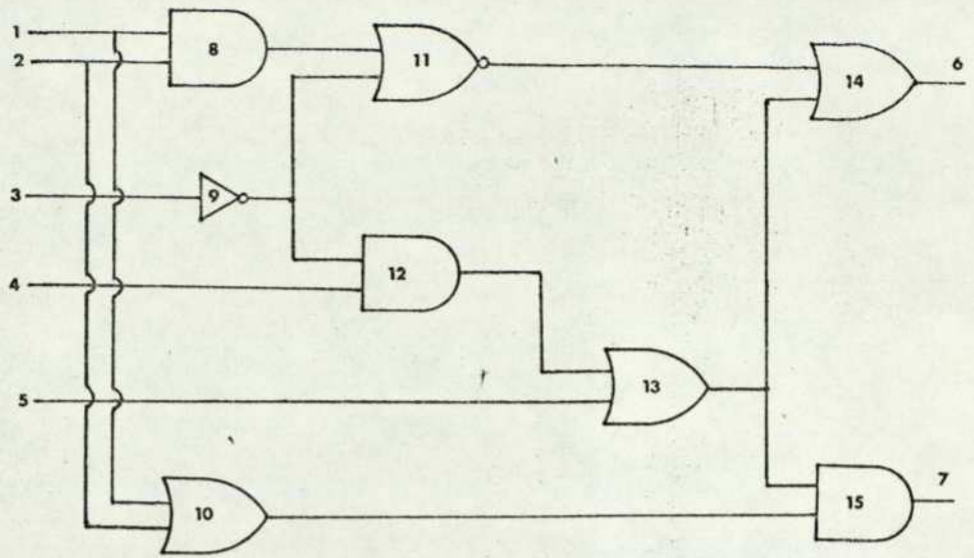


Fig 3.22 Circuit example

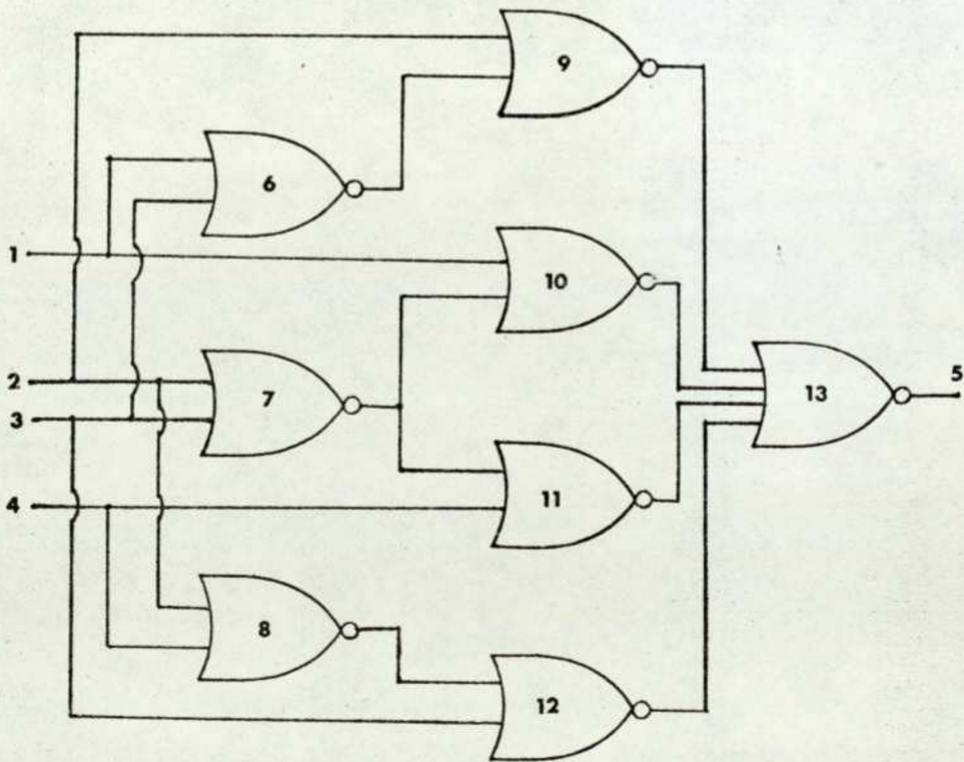


Fig 3.23 Schneider's circuit

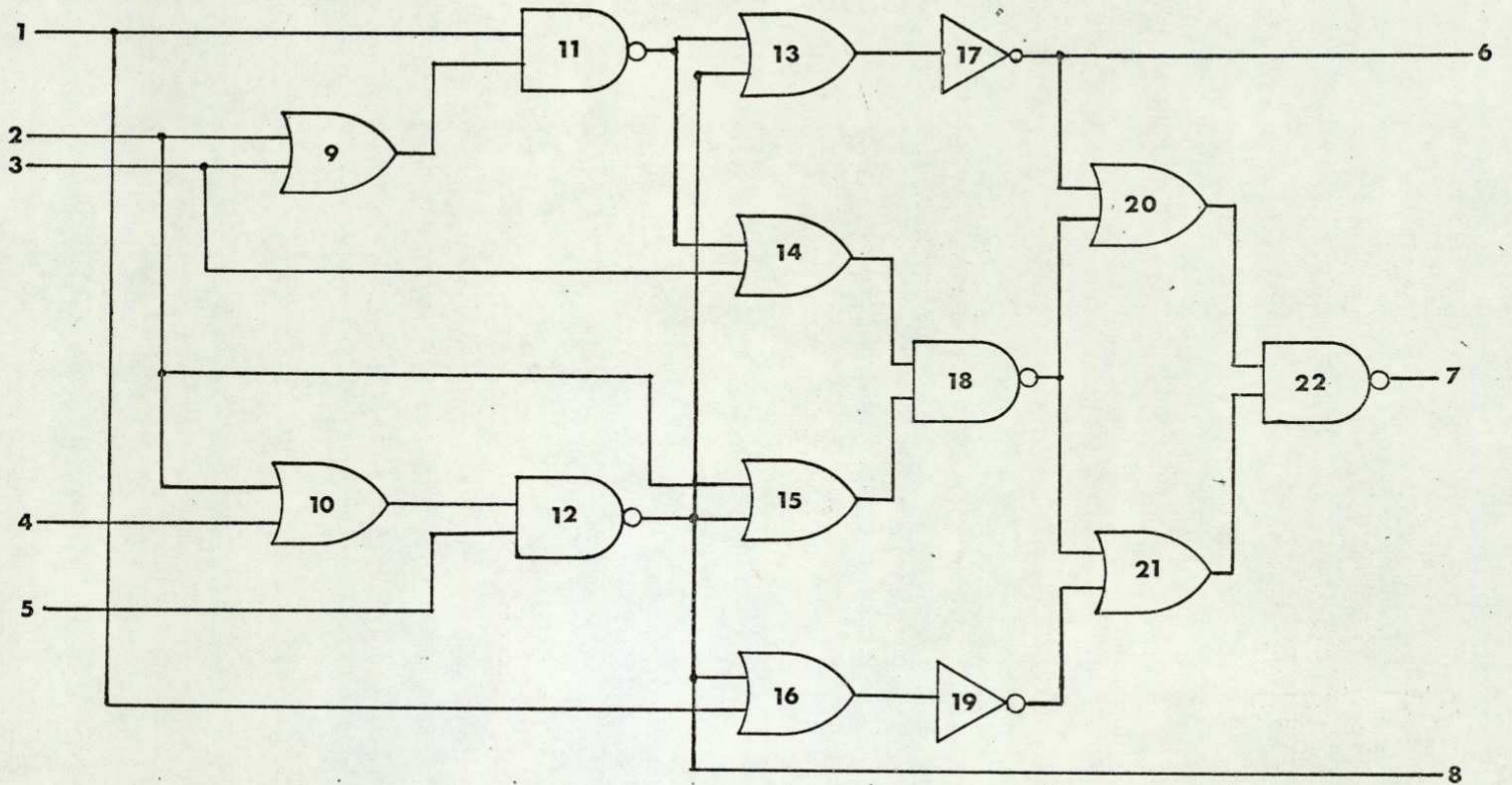


Fig 3.24 Circuit example

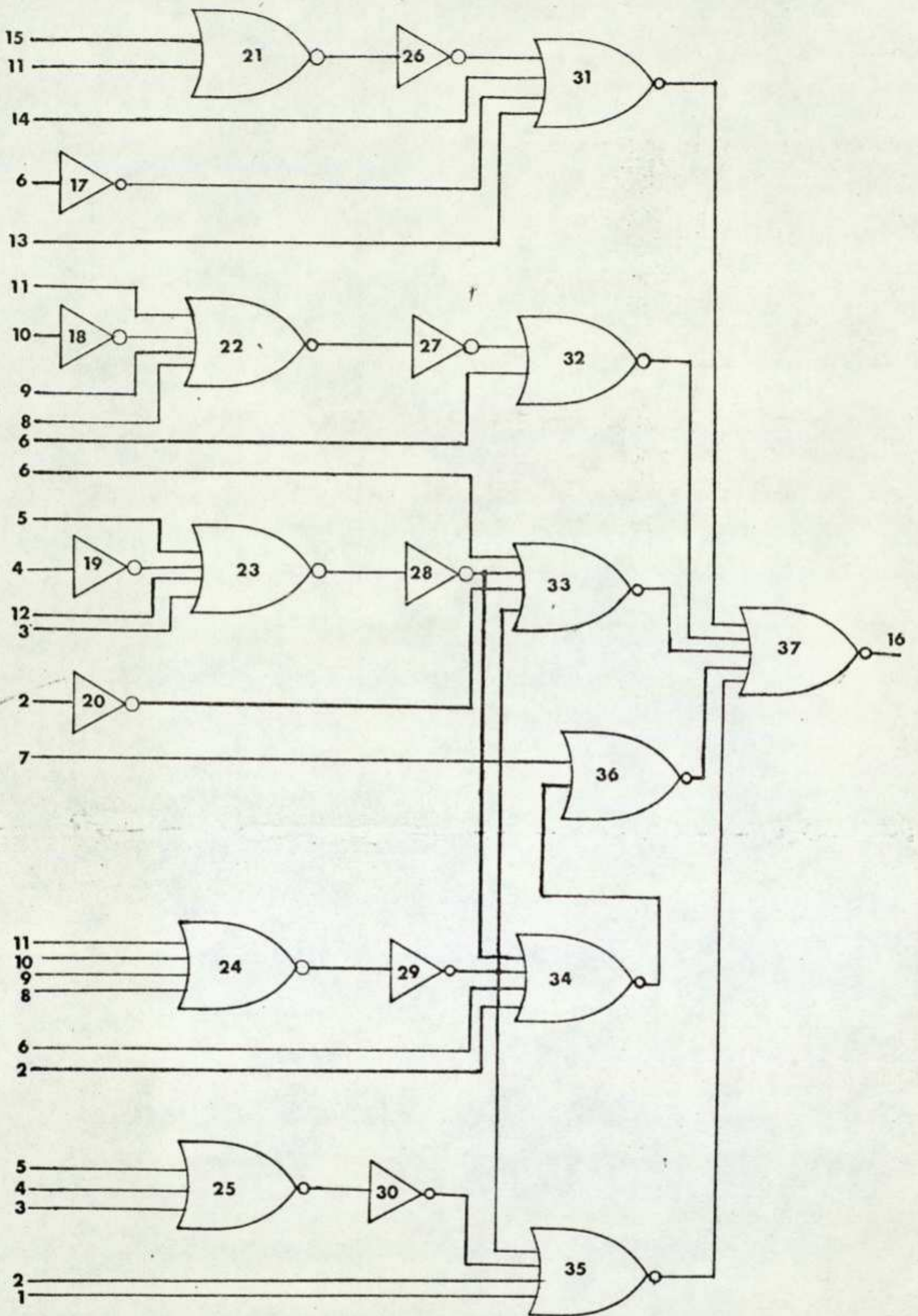


Fig 3.25 Sample Network (Inverters 17, 18, 19, 20 are assumed to be fault-free.)

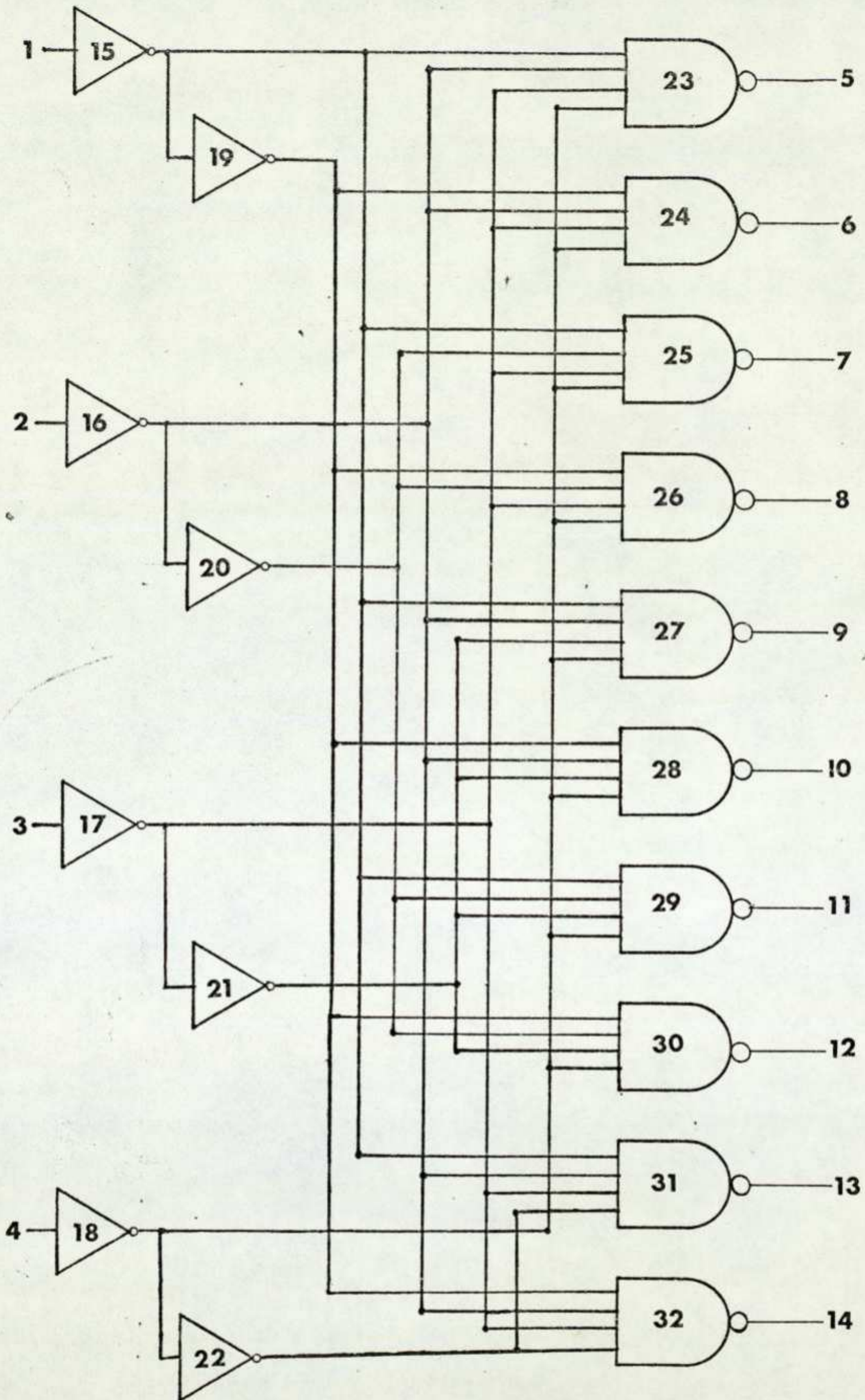


Fig 3.26 BCD- to -Decimal decoder

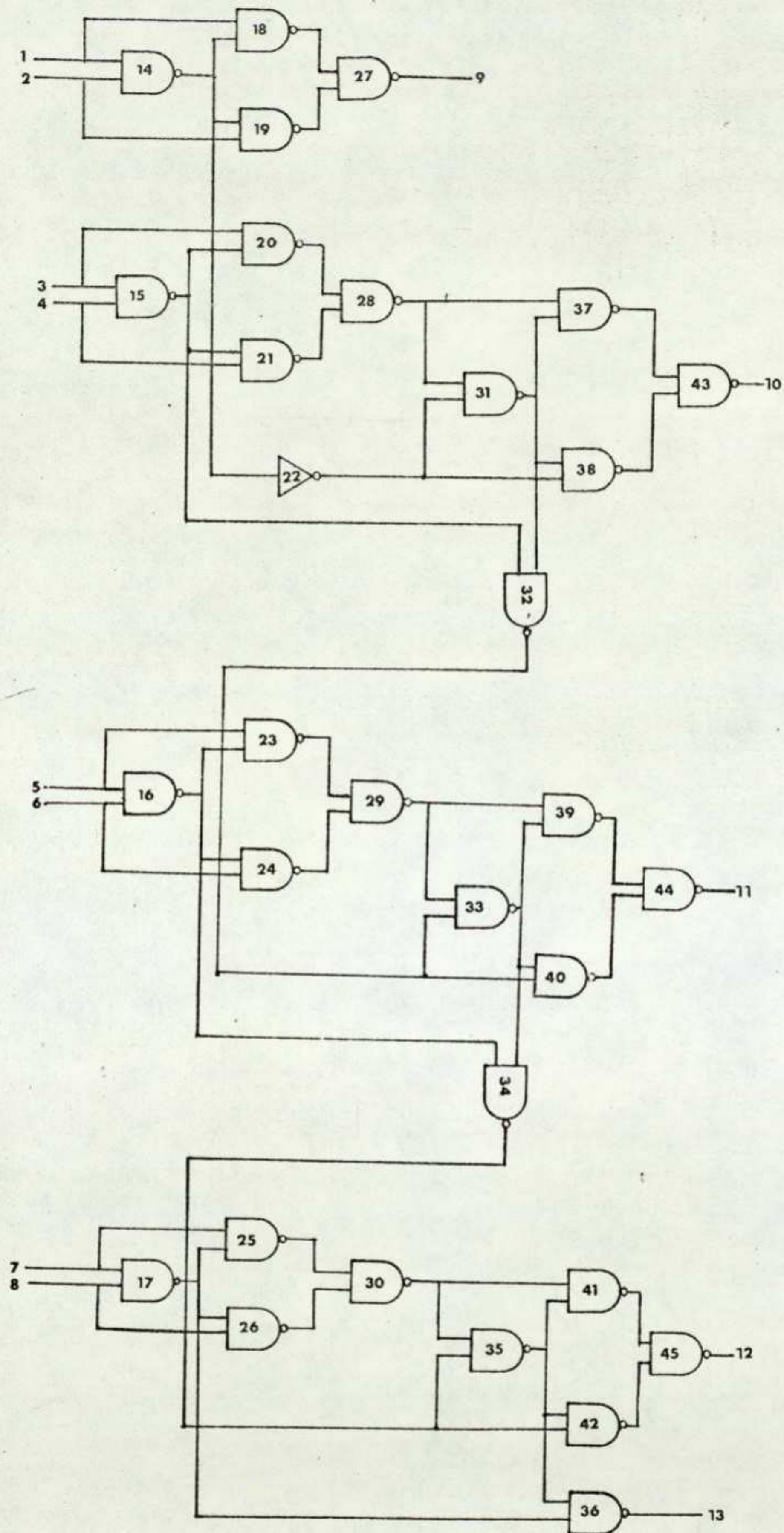
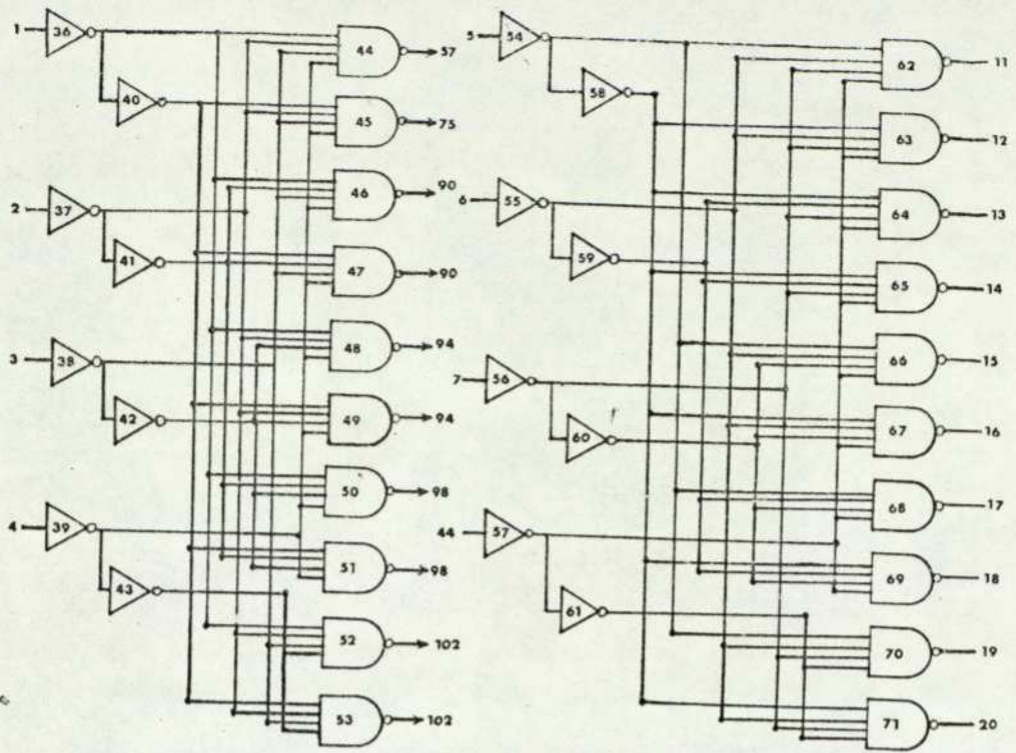
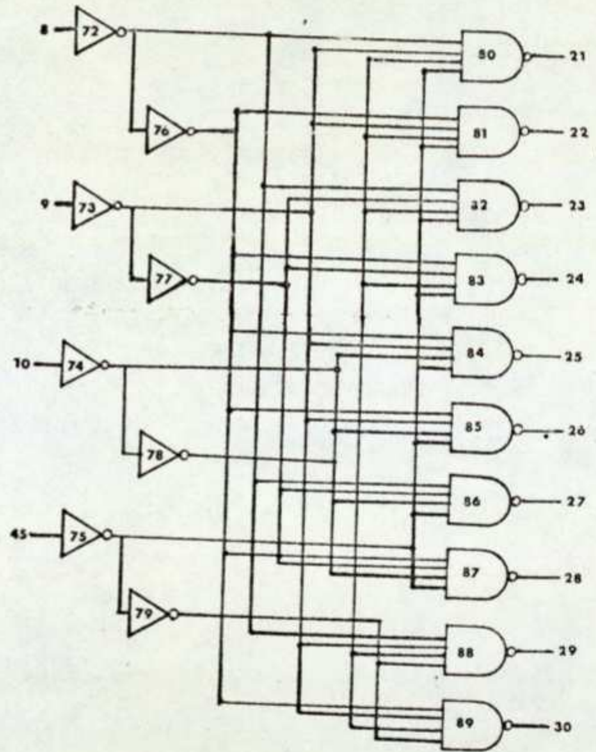
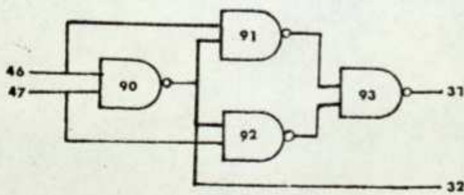
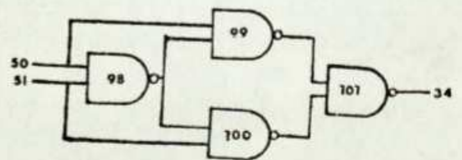


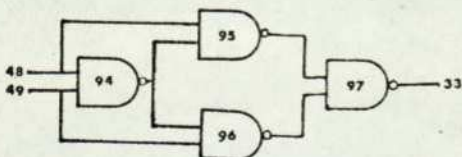
Fig 3.27 4-bit parallel adder











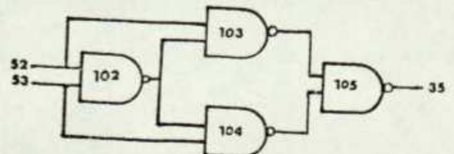


Fig 3.28 Circuit example

```
37 15 1 5
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2
7 7 7 7 6 6 6 6 6 7 7 7 7 7 6 6 6 6 6 6
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
6 0 0 0 0 10 0 0 0 0 4 0 0 0 0 2 0 0 0 0 11 15 0 0 0 8 9 11 18 0
3 5 12 19 0 8 9 10 11 0 3 4 5 0 0 21 0 0 0 0 22 0 0 0 0 23 0 0 0 0
24 0 0 0 0 25 0 0 0 0 14 17 13 26 0 6 27 0 0 0 6 20 28 29 0 2 6 28 29 0
1 2 29 30 0 7 34 0 0 0 31 32 33 36 35
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
37 10
37 36 34 28 23 5
37 31 26 21 11
37 33 28 23 19 4
37 32 27 22 18 10
37 33 29 24 11
37 35 30 25 3
37 31 17 6
37 35 29 24 11
37 32 6
37 35 30 25 3
```

Fig 3.29 Data input for program 'FOLD'

```
15 14
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 8 8 8 8 6 6 6 6 6 5 5 4 4
2 1 1 2 1 1 1 1 1 1 1 1 1 2 1
1 1 2 1 1 2 2 1 1 1 1 1 1 2 1
2 1 1 1 1 1 2 2 1 1 1 1 1 1 1
1 1 1 1 1 2 2 1 1 1 2 1 1 2 1
2 1 1 1 1 2 2 1 1 1 2 1 1 1 1
2 1 1 1 1 2 2 1 1 1 1 1 1 2 1
2 1 1 1 1 2 2 1 1 1 1 1 1 2 1
2 1 1 1 1 2 2 1 1 1 2 1 1 2 1
1 2 1 2 1 1 2 1 1 1 1 1 1 1 1
1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
1 1 1 2 1 1 1 2 1 1 1 1 1 1 1
1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 2 1 1 2 1 1 1 1 1
1 1 1 1 1 2 2 1 1 2 1 1 1 1 1
23 1 28 2 24 1 29 2 34 1 36 2 37 1 31 2 32 2 27 1 22 2 33 2 35 2
25 2 30 1 35 2 37 1 31 2 32 2 33 2 36 2
31 2 37 1 32 2 27 1 22 2 33 2 36 2 35 2
24 2 29 1 35 2 37 1 31 2 32 2 36 2 33 2
21 2 26 1 31 2 37 1 32 2 33 2 36 2 35 2
33 2 37 1 31 2 32 2 36 2 35 2
37 1 32 2 31 2 33 2 36 2 35 2
33 2 37 1 31 2 32 2 36 2 35 2
23 1 28 2 33 1 29 2 24 1 37 2
25 1 30 2 35 1 29 2 24 1 37 2
24 2 29 1 34 2 36 1 37 2
23 2 28 1 34 2 36 1 37 2
32 1 27 2 22 1 37 2
21 1 26 2 31 1 37 2
```

Fig 3.30 Data input for program 'MINI'

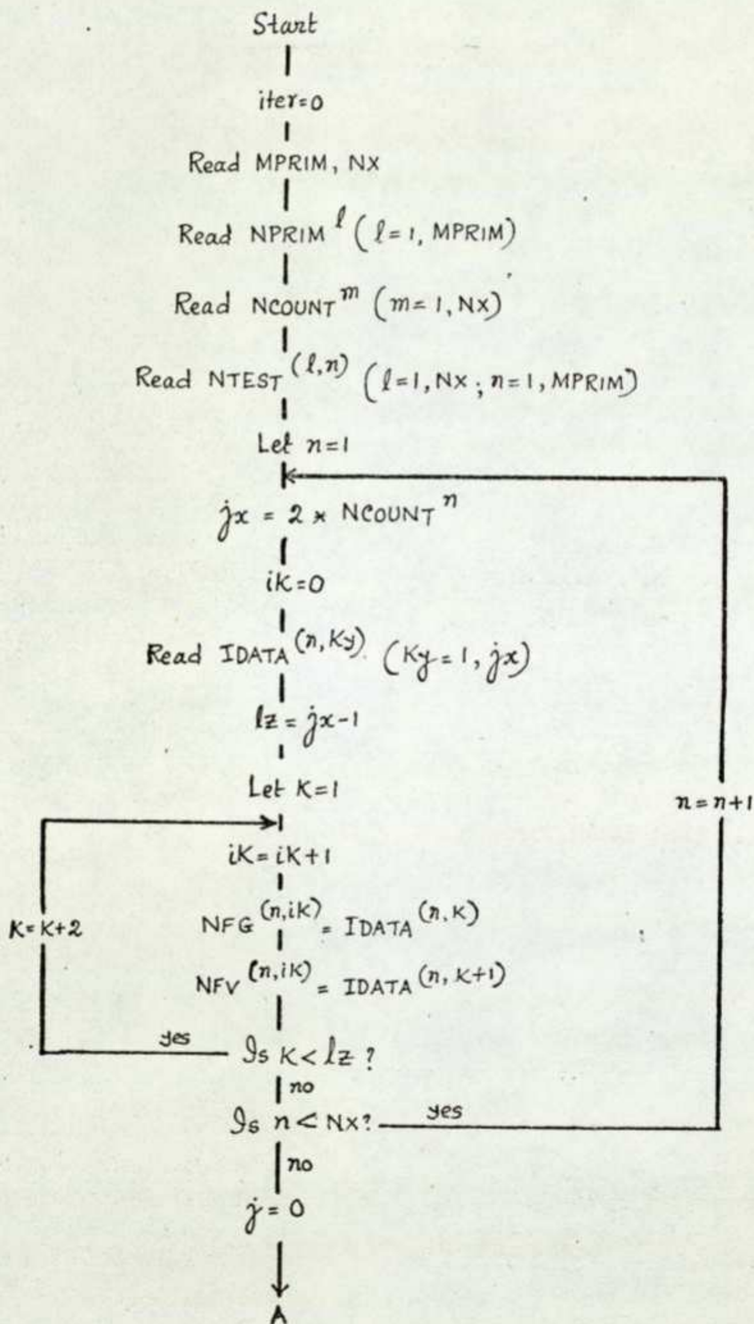
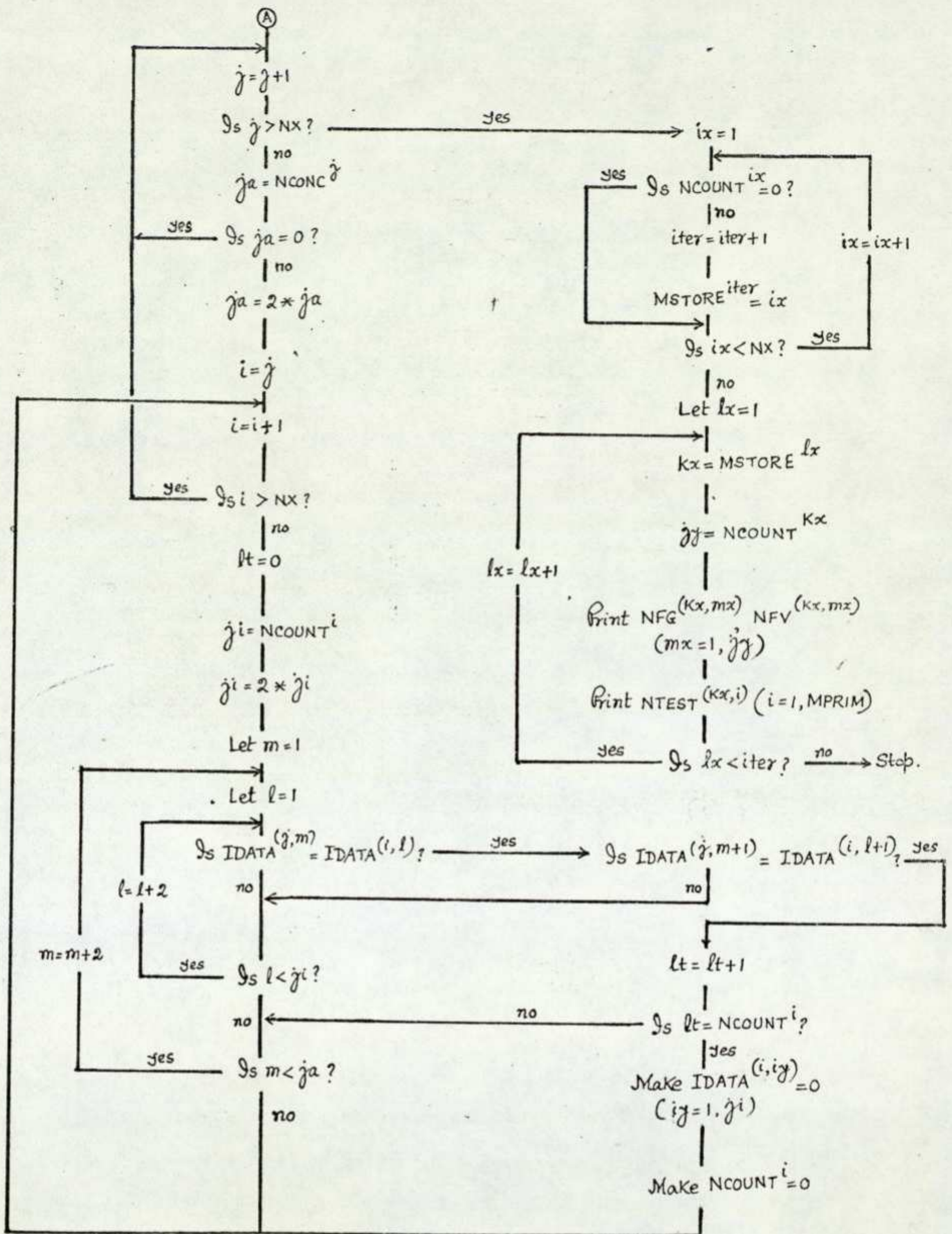


Fig 3.31 Flow-diagram of MINI

cont'd.



A
j = j+1
Is j > NX ?
yes
no
ja = NCONC j
ix = 1
Is NCOUNT ix = 0 ?
iter = iter+1
ix = ix+1
MSTORE iter = ix
Is ix < NX ?
Is ja = 0 ?
ja = 2 * ja
i = j
Let lx = 1
i = i+1
Kx = MSTORE lx
Is i > NX ?
jy = NCOUNT Kx
lt = 0
lx = lx+1
Print NFG (Kx, mx) NFV (Kx, mx)
(mx = 1, jy)
ji = NCOUNT i
ji = 2 * ji
Print NTEST (Kx, i) (i = 1, MPRIM)
Let m = 1
Is lx < iter ?
Stop.
Let l = 1
Is IDATA (j, m) = IDATA (i, l) ?
Is IDATA (j, m+1) = IDATA (i, l+1) ?
l = l+2
m = m+2
lt = lt+1
Is l < ji ?
Is lt = NCOUNT i ?
Is m < ja ?
Make IDATA (i, iy) = 0
(iy = 1, ji)
Make NCOUNT i = 0

which stores the number of elements in the fault set (defined as the 'weight' of the fault-set). Elements are stored in NCOUNT in decreasing order of 'weight'.

NTEST — a two-dimensional array of variables NTEST(M,N) which stores the mth test input in n positions.

IDATA — a two-dimensional array of variables which stores the faults detected by the ℓth test in k positions.

The data input for MINI i.e. the test set generated for the circuit of Fig.3.25 and the indistinguishable fault set associated with each test, is shown in Fig.3.30.

Fig.3.32 shows the minimized test set derived and the fault-set associated with each test; the stuck-at value of each element is shown underneath the element number in the figure e.g. $\binom{23}{1}$ means element 23 is s-a-0. The programme used a core size of 9K and took 10 secs of "mill time". The programme developed in this section for automatic test set generation should be capable of handling circuits having up to 15 primary inputs but the total elements in the circuit should not exceed 120. This limitation is not restrictive because by increasing the arrays sizes, the programme may be used to handle larger circuits. The mill time used by the programme depends to a large extent on the number of control input combinations to be applied. In its present state the programme may generate up to 300 control input combinations for an incompletely defined test; the core store used by the programme is 14K.

A number of circuits of different types of complexity have been handled by the programme and the results are recorded in Fig.3.33. The mill time recorded were obtained by running the programme on remote terminals under ICL-MAXIMOP system; significantly less mill times were obtained by running the same programme under ICL-GEORGE system.

MINIMAL TEST SET AND FAULTS DETECTED

23	28	24	29	34	36	37	31	32	27	22	33	35
1	2	1	2	1	2	1	2	2	1	2	2	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	2	1	1	2	1	1	1	1	1	1	1	1	1	2	1

25	30	35	37	31	32	33	36
2	1	2	1	2	2	2	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	2	1	1	2	2	1	1	1	1	1	1	2	1

24	29	35	37	31	32	36	33
2	1	2	1	2	2	2	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	1	1	2	2	1	1	1	2	1	1	2	1

21	26	31	37	32	33	36	35
2	1	2	1	2	2	2	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	2	1	1	1	1	2	2	1	1	1	2	1	1	1	1

23	28	33	29	24	37
1	2	1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	2	1	2	1	1	2	1	1	1	1	1	1	1	1

25	30	35	29	24	37
1	2	1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	1	1	1	2	1	1	1	1	1	1	1	1

24	29	34	36	37
2	1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	2	1	1	1	2	1	1	1	1	1	1	1

23	28	34	36	37
2	1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1

32	27	22	37
1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	1	1	1	2	1	1	2	1	1	1	1	1

21	26	31	37
1	2	1	2

PRIM. INPUTS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
TEST VALUE	1	1	1	1	1	2	2	1	1	2	1	1	1	1	1

Fig 3.32 Diagnostic test set for the circuit of Fig. 3.25

Example number	1	2	3	4	5	6	7
Number of inputs/outputs	5/2	4/1	5/3	4/10	8/5	15/1	10/25
Number of basic gates	8	8	14	18	32	22	70
Primary input fan-out	Yes	Yes	Yes	Yes	No	Yes	No
Mill time (in secs.) to generate static inputs only	8	16	38	11	55	40	55
Maximum number of control input combinations applied to generate any completely defined test	All	All	All	All	All	32	300
Mill time (in secs.) used to generate completely defined test set	18	20	50	22	100	66	1824
Number of tests generated	9	7	10^{+}	20	30	14	102
Time to minimize the test set	6	3	5	*	24	10	*
Number of tests after minimization and merging	6	5	6	11	24	10	45

+ untestable faults present (due to redundancy)   * minimization not required

Example 1. Circuit of Fig.3.22 (14)
2. Circuit of Fig.3.23 (54)
3. Circuit of Fig.3.24 (72)
4. Circuit of Fig.3.26
5. Circuit of Fig.3.27
6. Circuit of Fig.3.25 (71)
7. Circuit of Fig.3.28

Fig.3.33 Results of the test-set generation programme

3.6 Application of the Fault-detection Test Set in Locating Multiple Faults in Combinational Circuits

The procedure to locate a set of faults in a given circuit is as follows:

(a) Apply the test set, derived by the previously explained algorithm, to the circuit under test.

(b) Record the tests for which the primary output value is different from the functional circuit. If for tests, which are common elements of different test sub-sets in a multi-output circuit, one primary output is correct, then the corresponding sub-network can be assumed to fault-free.

(c) Remove all the possible faults located by the test set (assuming the hardware permits replacement) and go back to (a).

In this way, by a systematic application of fault detection, fault-location and fault-removal, one can completely diagnose and hence repair the network.

Consider the circuit of Fig.3.10a, having the multiple fault [9 s-a-1, 10 s-a-1, 12 s-a-1] which is to be detected and located. The test-set is applied to the circuit and the responses are recorded, under the heading 'FIRST RUN', as shown in Fig.3.34. Since it was assumed that there were no line failures, gate 14 and primary output 7 are in the same logical state and so are gate 15 and primary output 8. The dash in a column indicates that the corresponding output response to the test is not of interest i.e. the test does not belong to the test sub-set designed to detect faults in the sub-network associated with that primary output. The network output at 8 is incorrect for tests [111111, 011000] and for test 011000, the output at 7 is also incorrect.

FIRST RUN

Test	Expected Response at primary output		Actual Response at primary output	
	7	8	7	8
1 1 1 1 1 1	-	0	-	1
0 1 1 0 0 0	0	0	1	1
0 0 0 0 0 0	1	1	1	1
0 1 1 0 0 1	-	1	-	1
1 0 0 1 1 0	1	-	1	-

SECOND RUN

Test	Expected Response at primary output		Actual Response at primary output	
	7	8	7	8
1 1 1 1 1 1	-	0	-	0
0 1 1 0 0 0	0	0	0	0
0 0 0 0 0 0	1	1	1	1
0 1 1 0 0 1	-	1	-	1
1 0 0 1 1 0	1	-	0	-

Fig.3.34 Output response to the test set

Since test 011000 is common for both the sub-networks, it is reasonable to assume that the fault is in the shared portion of the network. It can be found that the common faults for the test are 9 s-a-1, 11 s-a-0, 10 s-a-1. The faults are removed and the test set is applied again, the result is recorded under the heading 'SECOND RUN'. Only test 100110 gives an incorrect response, hence the fault is in the indistinguishable fault category associated with the test. A further application of the test-set will prove that all faults have been removed.

3.6.1 Programme development for the faulted and fault-free simulation of combinational logic circuits

The complexity of modern logic circuits makes the task of determining their outputs for a given set of inputs extremely tedious and hence error-prone. The task becomes more difficult for an engineer who has to consider the effects of faults and find a test routine to detect and locate them. To alleviate the problem, a number of programs have been developed over the years which can determine, corresponding to the input patterns applied, different output conditions for a good model and a faulted model of the given circuit. A new program (PERM) devised by the author is described in this section which can be used to predict the output of a combinational circuit when it is fault-free or has a single or multiple fault. The flow-diagram of the programme is shown in Fig.3.35. Details of the circuit structure coding and fault-simulation for the program can be found in sections 2.3.1 and 2.4.2 respectively.

The rest of the arrays and variables used in the flow-diagram have the following uses:

NDATA — a one-dimensional array of variables NDATA(I) which is used to store circuit element number,

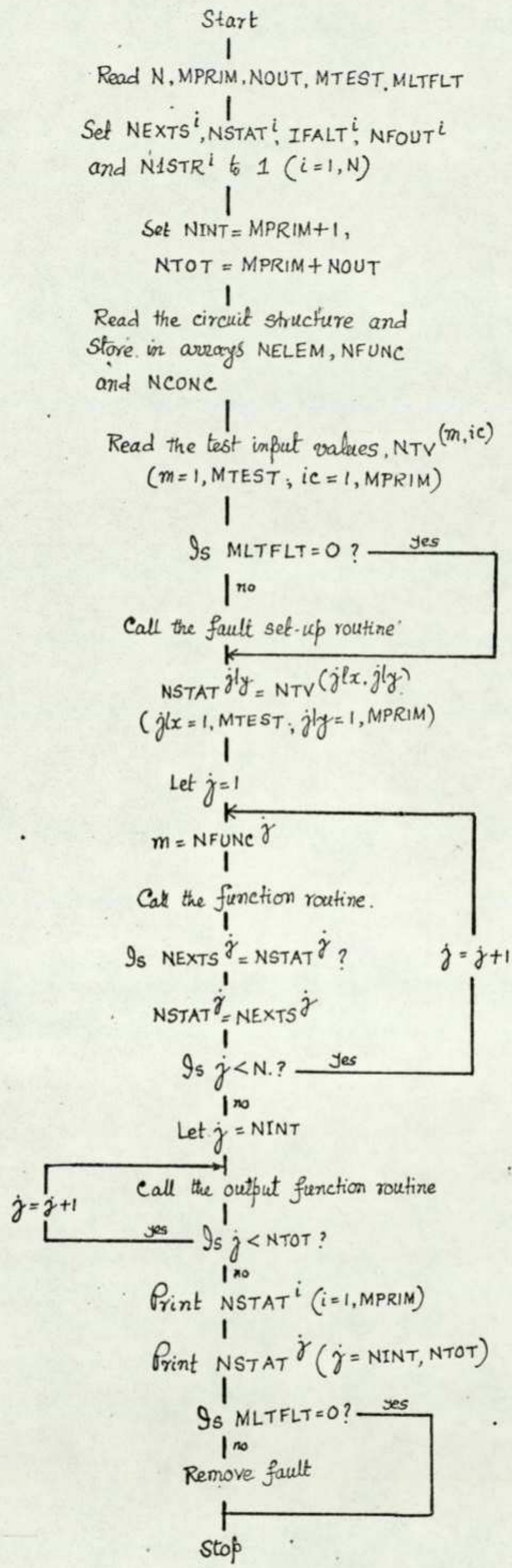


Fig 3.35 Flow-diagram of PERM

	its function codes, and input, output connections
N	total number of elements in the circuit.
MPRIM	number of primary inputs in the circuit.
NOUT	number of primary outputs in the circuit.
MTEST	number of input patterns to be applied to the circuit.
NTV	a two-dimensional array of variables NTV(M,N) which stores mth test input in n positions.
MLTFLT	multiplicity of a fault and is normally $\geqslant 1$. When it is set to 0, fault-free simulation of the circuit results.
NINT	an integer variable which holds the first primary output element number.
NTOT	an integer variable which holds the last primary output element number.

3.6.2 A circuit example

Consider the circuit of Fig.3.25. The coded circuit structure is shown in Fig.3.36. The fault-free operation of the circuit (MLTFLT=0) is simulated in the presence of the test-set generated by the previously described algorithm; the result is shown in Fig.3.37. This occupied about 7 secs of mill time and used a core size of about 6.5K. The behaviour of the circuit was next simulated successively for four different sets of multiple faults which were arbitrarily chosen. The results are shown in Fig.3,38 (the failed outputs are underlined). The mill time required in this case was about 22 secs. The fault-set associated with each test for the circuit is shown in Fig.3.32. The procedure described in section 3.6 may now be applied to repair the circuit

```
1 1 0 35
2 1 0 35 34 20
3 1 0 25 23
4 1 0 19 25
5 1 0 23 25
6 1 0 17 32 33 34
7 1 0 36
8 1 0 22 24
9 1 0 22 24
10 1 0 18 24
11 1 0 21 22 24
12 1 0 23
13 1 0 31
14 1 0 31
15 1 0 21
16 2 37
17 7 6 0 31
18 7 10 0 22
19 7 4 0 23
20 7 2 0 33
21 6 11 15 0 26
22 6 8 9 11 18 0 27
23 6 3 5 12 19 0 28
24 6 8 9 10 11 0 29
25 6 3 4 5 0 30
26 7 21 0 31
27 7 22 0 32
28 7 23 0 33 34
29 7 24 0 33 34 35
30 7 25 0 35
31 6 26 14 17 13 0 37
32 6 6 27 0 37
33 6 6 28 20 29 0 37
34 6 28 6 29 2 0 36
35 6 30 2 1 29 0 37
36 6 7 34 0 37
37 6 31 32 33 35 36 0 16
```

Fig 3.36 Coded structure of the circuit of Fig 3-25

```
FAULT-FREE SIMULATION

TEST VALUE= 2 1 1 2 1 1 1 1 1 1 1 1 1 2 1
    OUTPUT= 2

TEST VALUE= 1 1 2 1 1 2 2 1 1 1 1 1 1 2 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 1 2 1 1 2 1
    OUTPUT= 2

TEST VALUE= 2 1 1 1 1 2 2 1 1 1 2 1 1 1 1
    OUTPUT= 2

TEST VALUE= 1 2 1 2 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1
    OUTPUT= 1
```

Fig 3.37 Output response to the test-set

```
ELEMENT 33 STUCK-AT 1
ELEMENT 26 STUCK-AT 1
ELEMENT 32 STUCK-AT 1

TEST VALUE= 2 1 1 2 1 1 1 1 1 1 1 1 1 2 1
    OUTPUT= 2

TEST VALUE= 1 1 2 1 1 2 2 1 1 1 1 1 1 2 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 1 2 1 1 2 1
    OUTPUT= 2

TEST VALUE= 2 1 1 1 1 2 2 1 1 1 2 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 2 1 2 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1
    OUTPUT= 1
```

a

```
ELEMENT 21 STUCK-AT 2
ELEMENT 23 STUCK-AT 1
ELEMENT 32 STUCK-AT 1

TEST VALUE= 2 1 1 2 1 1 1 1 1 1 1 1 1 2 1
    OUTPUT= 1

TEST VALUE= 1 1 2 1 1 2 2 1 1 1 1 1 1 2 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 1 2 1 1 2 1
    OUTPUT= 2

TEST VALUE= 2 1 1 1 1 2 2 1 1 1 2 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 2 1 2 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
    OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1
    OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1
    OUTPUT= 1
```

b

Fig 3.38 Output response

cont'd

ELEMENT 22 STUCK-AT 2
ELEMENT 29 STUCK-AT 1
ELEMENT 25 STUCK-AT 1

TEST VALUE= 2 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1
OUTPUT= 1

TEST VALUE= 1 1 2 1 1 2 2 1 1 1 1 1 1 1 2 1
OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 1 2 1 1 1 2 1
OUTPUT= 2

TEST VALUE= 2 1 1 1 1 2 2 1 1 1 2 1 1 1 1 1
OUTPUT= 2

TEST VALUE= 1 2 1 2 1 1 2 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1
OUTPUT= 1

c

ELEMENT 28 STUCK-AT 2
ELEMENT 27 STUCK-AT 1
ELEMENT 30 STUCK-AT 2

TEST VALUE= 2 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1
OUTPUT= 1

TEST VALUE= 1 1 2 1 1 2 2 1 1 1 1 1 1 1 2 1
OUTPUT= 2

TEST VALUE= 1 1 1 1 1 2 2 1 1 1 2 1 1 1 2 1
OUTPUT= 2

TEST VALUE= 2 1 1 1 1 2 2 1 1 1 2 1 1 1 1 1
OUTPUT= 2

TEST VALUE= 1 2 1 2 1 1 2 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 2 1 1 1 2 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 1 2 1 1 2 1 1 1 1 1 1
OUTPUT= 1

TEST VALUE= 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1
OUTPUT= 1

d

4. DIAGNOSIS OF SOLID FAULTS IN SEQUENTIAL LOGIC CIRCUITS

4.1 Introduction

Techniques for generating test sequences for fault diagnosis in sequential circuits have been reviewed by several authors (14) (31) (52) and (72). These techniques are apparently based on two divergent approaches. In the first approach, known as the circuit testing approach, the test sequence is derived from the given circuit structure while the other approach, the machine identification approach, is based on verifying that the circuit under test does operate according to its state table.

4.1.1 Circuit testing approach

The circuit testing approach is mainly based on the works of Seshu and Freeman (73) and Poage and McCluskey (74). The technique used by Seshu and Freeman (73) is known as a sequential analyser; although this method is capable of diagnosing a large fraction of all possible faults in a circuit, it might not yield tests for some diagnosable faults.

Poage and McCluskey (74) presented a procedure for deriving the shortest test sequences which are to be applied to a sequential machine to guarantee that no fault from a set exists within the circuit. Although this method is elegant in that it produces optimal test sequences, it is quite clearly impractical for all but small circuits and small classes of faults.

The technique of D-Algorithm was first extended for the test generation of synchronous sequential circuits by Kubo (75). In this approach the sequential circuit under test is thought of as a cascaded connection of identical combinational circuits each one representing

the sequential circuit at a given instant of time. A similar technique to the above has been applied to the asynchronous sequential circuits by Putzolu et al (76). Although the D-Algorithm is thought to be a very powerful technique for combinational circuits, it has certain drawbacks when applied to sequential circuits (77), these are:-

(i) The method cannot generate a test in some cases due to the inadequacy of circuit modelling.

(ii) If a circuit has N elements and x cascaded connections of combinational circuits are developed, then this method will have to handle the circuit of xN elements. If xN becomes very large, then the computing time for generating tests of a circuit becomes prohibitive.

Other techniques for generating tests for sequential circuits have also been developed. Chang et al (78) have described procedures based on cause-effect equations (79), for detecting and locating multiple stuck-type failures in synchronous sequential networks. Hsiao and Chia (80) have extended the Boolean difference technique to deal with asynchronous sequential circuits. Two new methods of sequential circuit testing have been described by Arima et al (81) and Muth (82).

4.1.2 Machine identification approach

The problem of determining the internal behaviour of a synchronous sequential circuit (finite-state machine) by observing its response to various input sequences was introduced by Moore (83). The process of verifying that a given sequential network does operate according to its state-table, is referred to as 'checking' or 'fault-detection experiment'. The usual criterion of efficiency is the length of the experiment i.e. the number of input symbols applied. Moore's

approach results in experiments which are far too long to be practicable even for moderate sized circuits. Gill (84) treated in detail the initial state identification (distinguishing sequence) and final-state identification (homing sequence) problems. Hennie (85) treated the design of fault-detection experiments based on the knowledge of the state-table of the fault-free machine. This approach is heavily dependent upon the existence of a distinguishing sequence if the length of the experiments is to be within some practical limits. Kime (86) proposed modifications on Hennie's procedure yielding reduced upper bounds on the length of the experiment. Gönenc (87) formalised Hennie's method by introducing graph theoretic techniques, resulting in an algorithmic test sequence generation scheme. Generally the checking experiments for sequential circuits are capable of detecting a larger class of faults than the circuit testing approach and are relatively easy to derive (88). The derivation of checking experiments will be described later in the thesis.

The main reason for checking experiments not gaining practical acceptance is that they often lead to lengthy experiments. Friedman et al (52) have indicated that the checking experiments which do not take actual circuitry into account tend to be unnecessarily long. However as far as the author is aware no attempt has been made so far to devise a technique for testing synchronous sequential circuits in which both the structure and the behaviour of a given circuit are considered simultaneously; a method to derive behavioural and structural test procedures for asynchronous networks has been described by Thayse (89). An algorithmic test procedure, based on the structure and the behaviour, will be described in this chapter for generating a test sequence for synchronous sequential circuits; the procedure is derived by a combination of previously described fault-folding techniques and

the conventional checking experiments.

4.1.3 Notations and definitions

The sequential circuits or machines to be considered in this thesis are deterministic, strongly connected and completely specified. The machines are of the Mealey model (90) where

$X = \{x_1, x_2, \ldots x_m\}$ is a finite set of input symbols,

$S = \{s_1, s_2, \ldots s_m\}$ is a finite set of internal states, and

$Z = \{z_1, z_2, \ldots z_p\}$ is a finite set of output symbols.

There exists an output function $\omega(s_i, x_j)$ that specifies an output symbol for every $s_i \varepsilon S$, $x_j \varepsilon X$. There exists a next-state function, $\delta(s_i, x_j)$, that specifies a next state for every $s_i \varepsilon S$, $x_j \varepsilon X$.

A machine is said to be 'deterministic' if and only if the future behaviour of the machine can be completely determined by the present state and the input sequence to be applied.

A machine is a 'strongly connected machine' if and only if it is possible to reach any state s_j from any other state s_i.

A 'checking experiment' is a sequence of input symbols (input sequence) which, when applied to the machine under test, results in a sequence of output symbols (response) which establishes whether or not the machine is equivalent to the correctly working machine. At the beginning of the experiment the network is said to be in an initial (or starting) state and at the end of an experiment the network is said to be in a 'final state'.

An experiment is said to be 'adaptive' or 'preset' depending on whether the next input signal to apply is or is not based upon the output signals previously produced by the machine.

A 'synchronizing sequence' for a sequential machine is an input sequence whose application is guaranteed to leave the machine in a

certain final state, regardless of the particular initial state of the machine. Some machines possess such sequences, others do not. For example the application of the sequence 000 to the machine described by Table 4.1 always leaves the machine in state B (Fig.4.1) regardles of what initial state it was in i.e. 000 is a synchronizing sequence. A 'homing sequence' for a sequential machine is an input sequence whose application makes it possible to determine the final state of the machine by observing the corresponding output sequence that the machine produces. For example, the machine of Table 4.2 has no synchronising sequences although it does have a number of homing sequences. Among these is 101, for as indicated in Fig.4.2, each of the output sequences that might result from the application of 101 is associated with just one final state. Every reduced sequential machine has a homing sequence.

A 'distinguishing sequence' is an input sequence whose application makes it possible to determine the initial state of the machine by observing the corresponding output sequence that the machine produces. For example, 11001 is a distinguishing sequence for the machine of Table 4.3. As shown in Fig.4.3, the output sequence that the machine produces in response to 11001 uniquely specifies its initial state. But the knowledge of the initial state and the input sequence is always sufficient to determine uniquely its final state as well. Consequently, every distinguishing sequence is also a homing sequence; on the other hand, not every homing sequence is a distinguishing sequence. While every machine has at least one homing sequence, not every machine has a distinguishing sequence.

A transfer sequence from state s_i to state s_j is an input sequence which transfers the machine from s_i to s_j. Such sequences can always be found for a strongly connected machine, and require at most (m-1) symbols for a m-state machine. Fig.4.4 illustrates the transfer

PS	NS,z X=0	X=1
A	B,0	C,0
B	B,1	D,0
C	A,0	E,1
D	C,0	E,0
E	C,1	C,0

Table 4.1 Machine M1

PS	NS,z x=0	x=1
A	C,0	D,0
B	A,0	C,0
C	D,1	B,0
D	C,1	A,1

Table 4.2 Machine M2

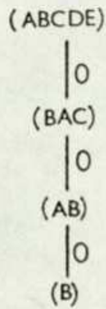


Fig 4.1 Synchronizing sequence derived

Initial state	Output sequence	Final state
A	010	B
B	011	A
C	000	D
D	100	B

Response to 101

Fig 4.2 A homing sequence for machine M2

PS	NS,z X=0	X=1
A	B,0	C,0
B	D,0	E,1
C	D,0	A,0
D	E,0	B,0
E	E,1	C,1

Table 4.3 Machine M3

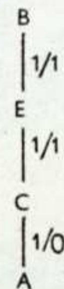


Fig 4.4 A transfer sequence (111)

Initial state	Output sequence	Final state
A	00000	B
B	11000	C
C	00001	C
D	01111	C
E	10000	B

Response to 11001

Fig 4.3 A distinguishing sequence for machine M3

sequence necessary to take the machine of Table 4.3 from state B to state A; the transfer sequence is 111.

4.2 Checking Experiments for Sequential Circuits

A checking experiment may be summarised as follows (91):-

(i) A checking experiment starts with a homing sequence followed by the appropriate transfer sequence, so as to manoeuvre the machine to an initial pre-specified state.

This part of the experiment is termed the 'initialising part'.

(ii) The machine is next supplied with an input sequence which causes it to visit each state and to display its response to the distinguishing sequence.

(iii) Finally, the machine is made to go through every state transition and in each case the transition is verified by displaying its response to the distinguishing sequence.

The second and the third part of the experiment can be termed as the 'checking part'.

However the method outlined above can only be applied to a reduced, strongly connected machine provided it has a distinguishing sequence; otherwise the checking experiment becomes extremely complicated and very long. A different procedure for carrying out the checking part of the experiment will be described in section 4.3; this procedure takes into account the structure of the circuit and does not require distinguishing sequences.

4.2.1 A computer-aided design procedure for homing experiments

In this section a technique derived by the author for the construction of minimal homing experiments to identify the final state of a given machine is described. Hibbard (92) has shown that a homing sequence of length < $\frac{1}{2}n(n-1)$ symbols exists for every reduced n-state

sequential machine. However before describing the technique it is necessary to introduce some of the terminology which is used in the design of minimal homing experiments.

A machine M which has n states $(s_1, s_2, \ldots s_n)$ can initially be in any one of the n states before the beginning of the experiment; this set of n states is known as the 'initial uncertainty' regarding the state of the machine. A collection of states of M which is known to contain the present state is referred to as the 'uncertainty'. The 'uncertainty' of M is thus any finite sub-set of the states of M. The uncertainty following from the application of an input sequence x_i is called the 'x_i successor'. The 'successor tree' which is defined for a specified machine M and a given initial uncertainty is a structure which displays graphically the x_i successor uncertainties for every possible x_i. A collection of uncertainties is referred to as an 'uncertainty vector'. An uncertainty vector whose components contain a single state each is said to be a 'trivial uncertainty', while an uncertainty vector whose components either contain a single state or identical repeated states is said to be a 'homogeneous uncertainty'. The vectors (AA)(B)(C) and (A)(B)(A)(C) are homogeneous and trivial, respectively.

A homing sequence for given a machine may be obtained by constructing the successor tree for the machine, and then determining the shortest path leading from the initial uncertainty to a trivial or a homogeneous uncertainty.

A 'homing tree' is a successor tree whose branches are terminated when any of the following occur:

(i) an uncertainty vector in the kth level is also associated with some branch in a preceeding level.

(ii) a branch is associated with a trivial or a homogeneous uncertainty vector.

The construction of the homing tree for a machine M4 (Table 4.4) and an initial uncertainty (ABCD) is illustrated in Fig.4.5; the shortest path leading from the zeroth level to a homogeneous uncertainty is indicated by arrows, hence 010 is the shortest homing sequence.

A procedure has been developed by the author to find the homing sequence directly from the state-table without having to construct the homing tree first. This procedure consists of the following steps:-

(1) Set the initial values of s (the total number of states) and j equal to zero, where 2^j is the number of the sets of states at the jth level.

(2) Apply input x(=0 or 1) to each of the set of states at jth level and set $k=2^j$.

(3) If the kth set to which x is applied is a unit or a null set, go to step 11, otherwise go to step 4.

(4) Partition the kth set into 2kth and $(2k+1)^{th}$ sub-sets containing the states with output 0 and 1 respectively.

(5) If either of the sub-sets in step 4 have the same number of elements as the kth set, go to step 6 otherwise go to step 9.

(6) Go to step 8 if the kth set is equal to the subset otherwise go to step 7.

(7) Go to step 9 if k=1 otherwise set k=k/2 or (k-1)/2 depending on whether k is even or odd respectively and go to step 6.

(8) Go to step 9 if a change of the input value has already been made for the jth level, otherwise set $x=\bar{x}$ and go to step 2.

(9) If there are p identical copies of an element in a sub-set, remove (p-1) of these and increase s by (p-1); similarly for other elements in the sub-set.

(10) Increase s by 1 if a sub-set is a unit sub-set. Repeat

PS	NS, z	
	X=0	X=1
A	B,0	D,0
B	A,0	B,0
C	D,1	A,0
D	D,1	C,0

Table 4.4 Machine M4

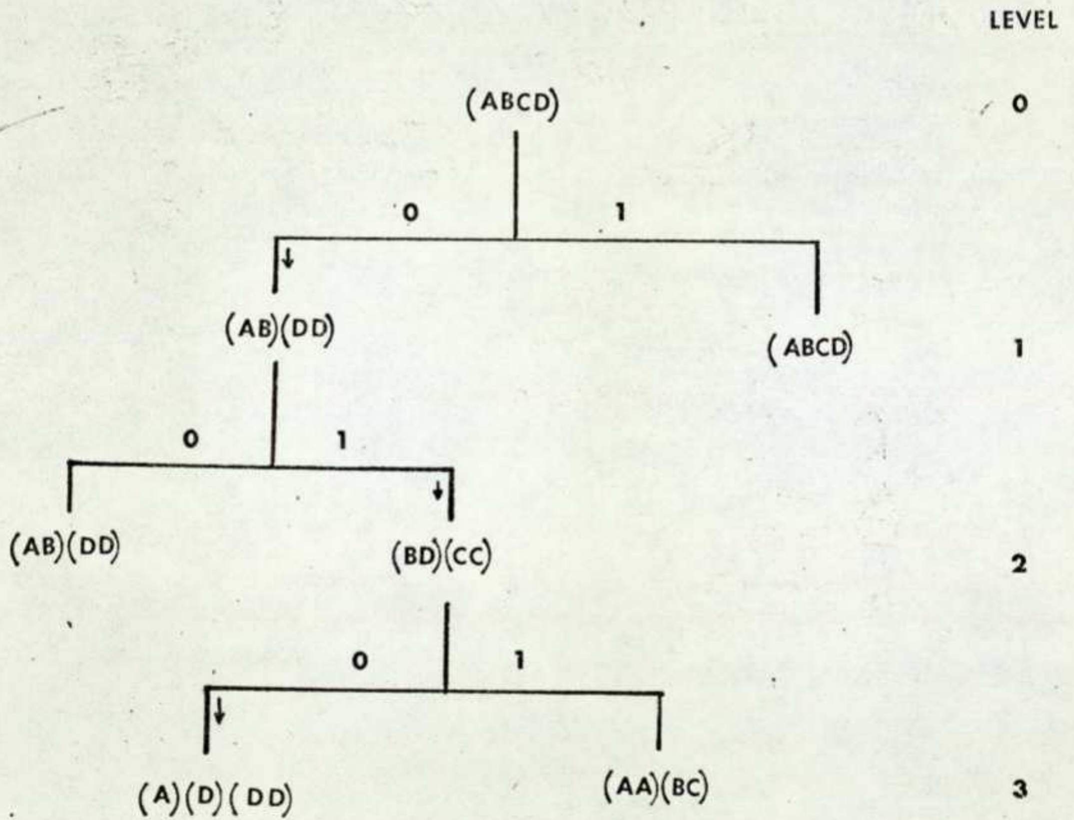


Fig 4.5 Homing tree for M4

this step in conjunction with step 9 for both sub-sets obtained in step 4 only if step 9 is entered directly from step 5.

(11) Go to step 14 if s is equal to the total number of states; otherwise go to step 12.

(12) If input x has been applied to the 2^j sets in the jth level go to step 13, otherwise increase k by 1 and go to step 3.

(13) Increase j by 1 and go to step 2.

(14) Print the input values recorded so far; this is the homing sequence.

The procedure has been computer programmed; the flow-diagram of the program HOME, is shown in Fig.4.6. The named arrays and variables have the following definition and uses

NL	total number of states in a given state table
KX	initial input value assigned to partition the set
NI	number of inputs in the state table
NPRS	a one-dimensional array of variables NPRS(I) in which is stored the present states of the state-table
NE	a two-dimensional array of variables NE(I,J) which stores the next-state of the jth state, resulting from the application of the ith input
NO	a two-dimensional array of variables NO(I,J) storing the output value resulted from the application of the ith input to the jth state.
NFINI	an integer variable used to hold the number of states
NFLAG,MFLAG	two integer variables used to identify the initial and the final set in a level

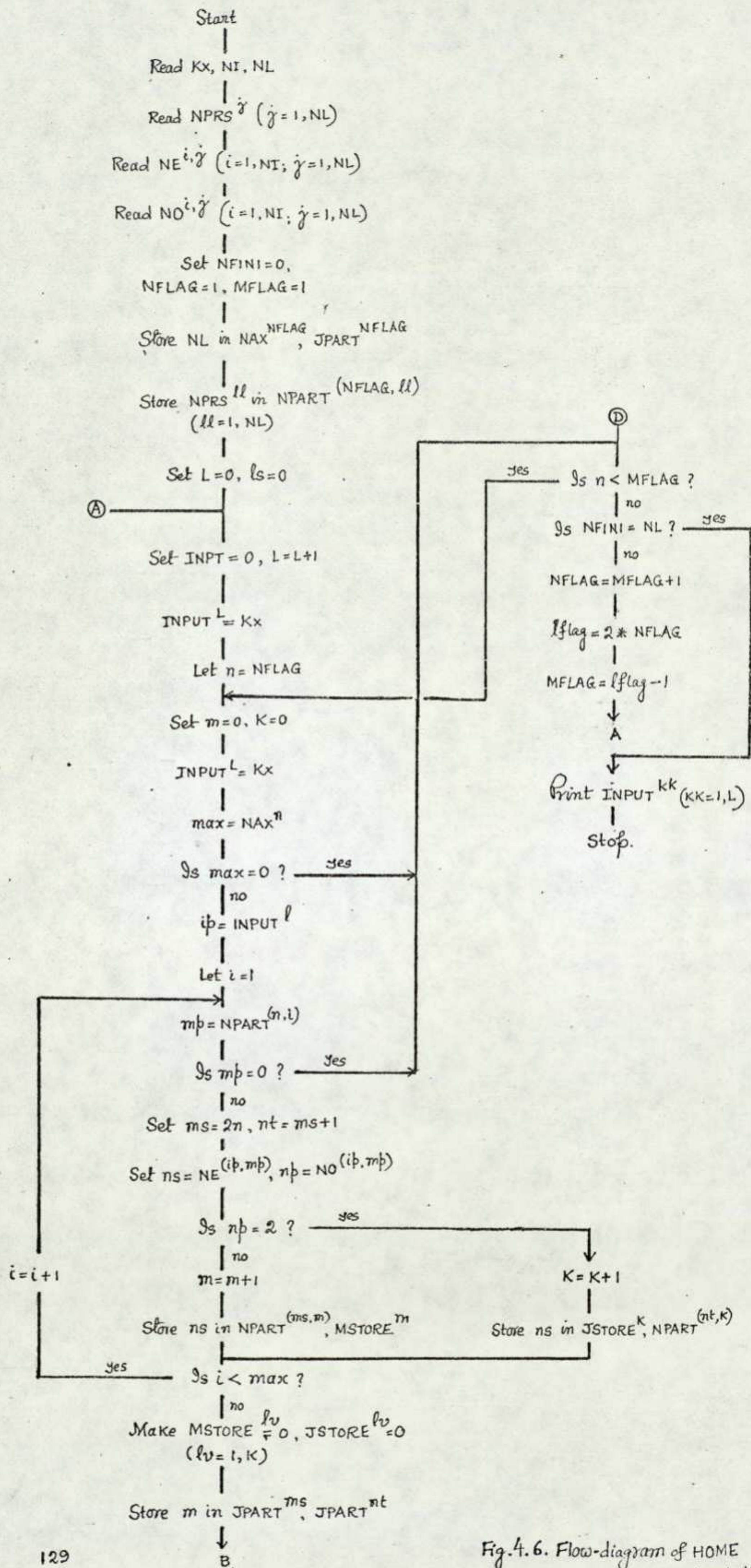
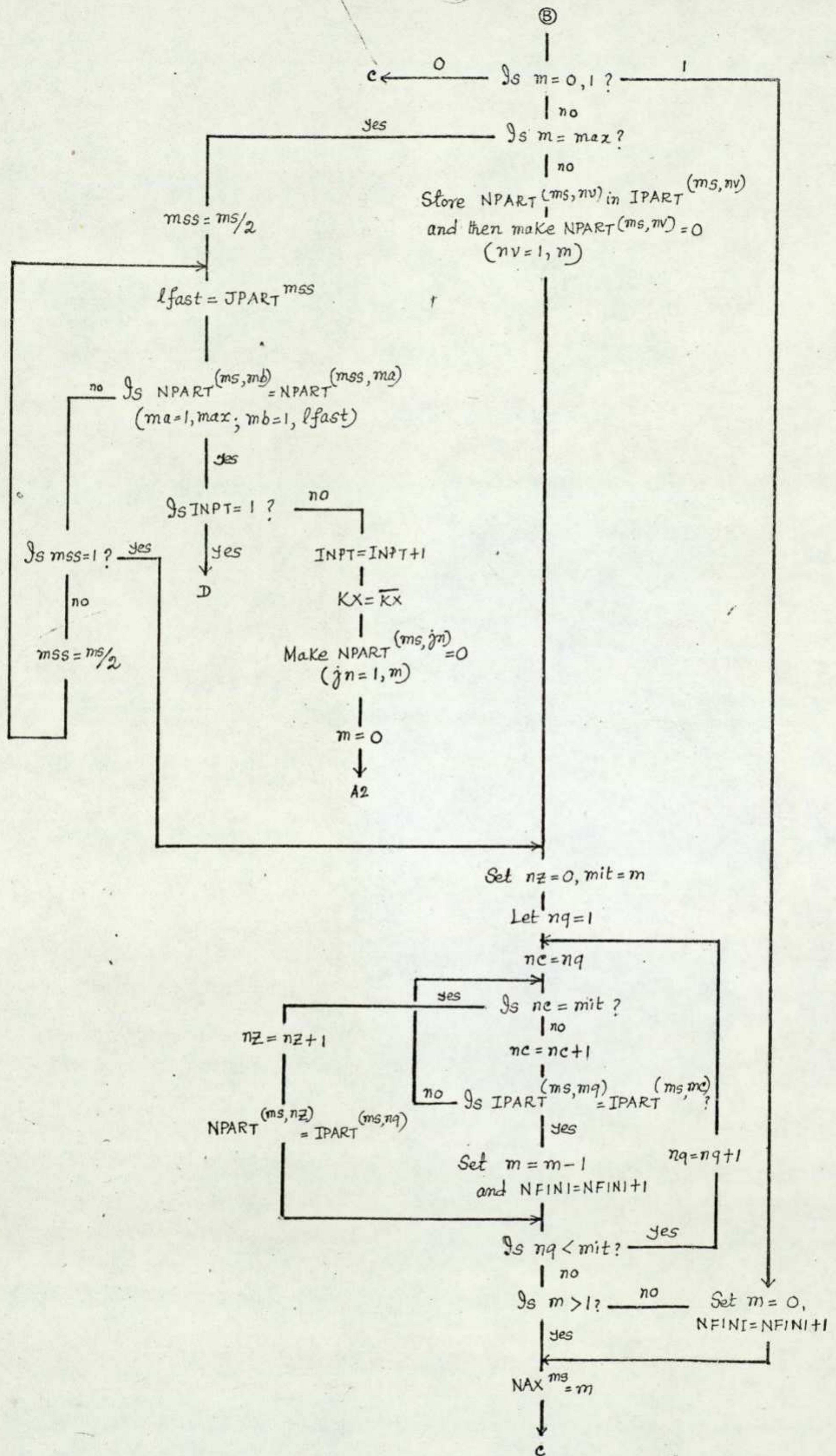
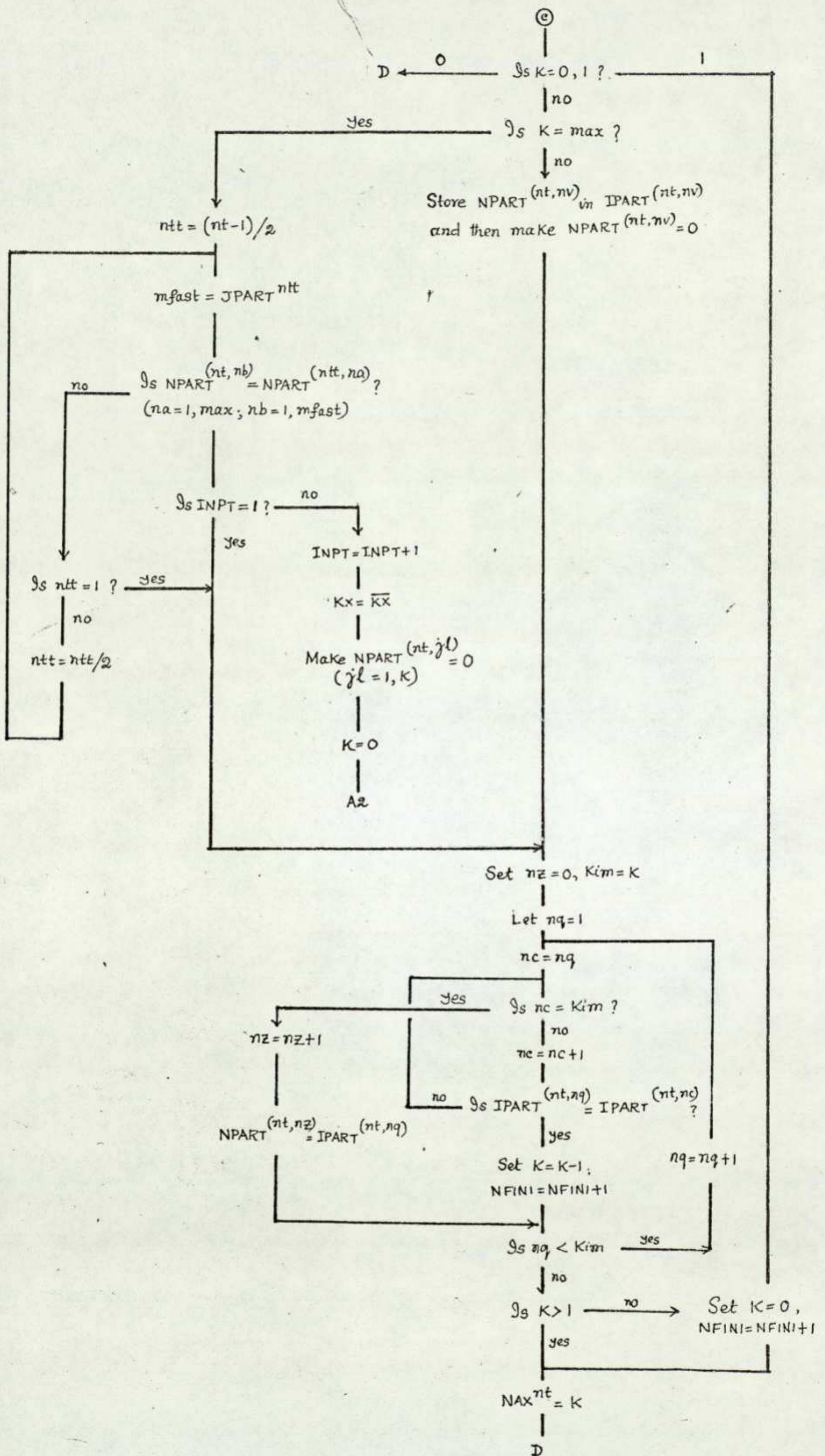


Fig. 4.6. Flow-diagram of HOME (cont'd)



(cont'd)



C
D
0
Is K=0, 1 ?
1
no
yes
Is K = max ?
no
Store NPART(nt,nv) in IPART(nt,nv)
and then make NPART(nt,nv) = 0
ntt = (nt-1)/2
mfast = JPART ntt
Is NPART(nt,nb) = NPART(ntt,na) ?
(na=1, max; nb=1, mfast)
no
Is INPT=1 ?
no
yes
INPT = INPT+1
KX = KX
Make NPART(nt,jl) = 0
(jl = 1, K)
K=0
A2
Is ntt = 1 ?
yes
no
ntt = ntt/2
Set nz=0, Kim=K
Let nq=1
nc=nq
Is nc = Kim ?
yes
no
nz=nz+1
nc=nc+1
Is IPART(nt,nq) = IPART(nt,nc) ?
no
yes
NPART(nt,nz) = IPART(nt,nq)
Set K=K-1;
NFINI=NFINI+1
nq=nq+1
Is nq < Kim
yes
no
Is K>1
no
Set K=0,
NFINI=NFINI+1
yes
NAX nt = K
D

JPART	a one-dimensional array of variables JPART(I) used to store the number of elements in each set
NPART	a two-dimensional array of variables NPART(I,J) in which the jth position is used to store the number of elements of the ith set
IPART	a two-dimensional array having the same function as NPART
MSTORE	a one-dimensional array of variables MSTORE(I) in which next-states with associated output 1, are stored
JSTORE	a one-dimensional array of variables JSTORE(I) in which next-states with associated output 0, are stored
INPT	an integer variable which is set to 1 when the input applied to a level has to be complemented, otherwise it is equal to 0
L	an integer variable which holds the length of the homing sequence
INPUT	a one-dimensional array of variables INPUT(I) in which the homing sequence is stored

As an illustration, the programme HOME is applied to find the homing sequence for a circuit whose state-table is shown in Table 4.5; the coded form of the state-table, to be used as input data for programme HOME, is shown in Fig.4.7. The programme output is reproduced in Fig.4.8; the computer time required to calculate the homing sequence was 5 secs. The programme uses a core store of about 20K and can handle state-tables having up to 16 states, but with suitable modifications this can be increased to handle state-tables of much larger circuits.

PS	NS,z x=0	x=1
A	B , 1	C,0
B	B , 1	E,1
C	C , 1	A,0
D	A , 1	F,1
E	E , 1	B,0
F	E , 1	D,1

Table 4.5 State table

```
2 2 6
1 2 3 4 5 6
2 3 2 5 3 1 1 6 5 2 5 4
2 1 2 2 2 1 2 2 2 1 2 2
```

Fig 4.7 Data input for program HOME
(A=1, B=2, C=3, D=4, E=5, F=6)

```
                NFLAG=  1     MFLAG=  1
                INPUT FOR THE SET= 2
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 3  1  2
                                     5  6  4
                NFLAG=  2     MFLAG=  3
                INPUT FOR THE SET= 2
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 1  3
                                     5
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 2
                                     4  6
                NFLAG=  4     MFLAG=  7
                INPUT FOR THE SET= 2
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 3  1
                                     0
                INPUT REVERSED
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 0
                                     2  3
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 0
                                     1  5
                NFLAG=  8     MFLAG= 15
                INPUT FOR THE SET= 1
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 0
                                     2  3
                INPUT REVERSED
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 1
                                     5
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 3  2
                                     0
                NFLAG= 16     MFLAG= 31
                INPUT FOR THE SET= 2
STATES WITH OUTPUT=1
                                    STATES WITH OUTPUT=2
 1
                                     5
HOMING SEQUENCE 2 2 1 2 2
```

Fig 4.8 Homing seqence derived

4.2.2 Programme development for automatic transfer sequence generation

A homing sequence transfers a machine to a unique final state. It is then necessary to apply an input sequence to transfer the machine into a pre-specified state, as explained previously, to complete the initialising part of the checking experiment. It may be recalled that the input sequence that takes a machine from a state s_i to state s_j is the transfer sequence $T(s_i,s_j)$. A procedure has been developed to automate the transfer sequence generation; it consists of the following steps:-

(1) Let ℓ=m=k=1 and j=0.

(2) Apply input 0 to the present state of the circuit.

(3) Go to step 8 if the next state is the same state to which it is desired to transfer the machine; otherwise go to step 4.

(4) Increase m by 1 and record the mth input value and the corresponding next state.

(5) Go to step 7 if both inputs 0 and 1 have been applied to the present state of the machine, otherwise go to step 6.

(6) Apply input 1 to the present state of the circuit and go to step 3.

(7) Increase ℓ by 1; make the ℓth next state, recorded in step 4, as the present state and go to step 2.

(8) Increase j by 1 and record the mth input value in the jth position.

(9) Go to step 10 if m=2 or 3 otherwise set m=m/2 and go to step 8.

(10) Select jth input value in step 8 as the kth symbol of the transfer sequence.

(11) Decrease j by 1. If j=0, stop, otherwise increase k by 1 and go to step 10.

The above procedure has been computer-programmed as a subroutine TRASEQ; the flow-diagram is shown in Fig.4.9. The arrays and variables used in the programme have the following definition and uses

NEW	an integer variable used to hold the present state of the circuit.
NOLD	an integer variable specifying the state to which NEW has to be transferred.
NSTORE	a one-dimensional array of variables NSTORE(I) in which the next-states entered during the execution of the programme are stored.
MO	a two-dimensional array of variables MO(I,J) in which the next state of the jth state, corresponding to the ith input, is stored.
NSIM	a one-dimensional array of variables NSIM(I) in which the input values applied during the execution of the programme are stored.

The external data required for the execution of the programme are the values of NEW, NOLD and array MO (the state table). The application of the subroutine will be found in section 4.3.4.

4.3 A Method for Test Sequence Generation of Sequential Circuits Based on Behaviour and Structure

4.3.1 Application of fault-folding to a synchronous sequential circuit

The application of fault-folding techniques in the derivation of diagnostic tests for combinational circuits has been described previously (section 3.5). Synchronous sequential circuits in a particular state behave like a combinational circuit with feedback lines acting as primary

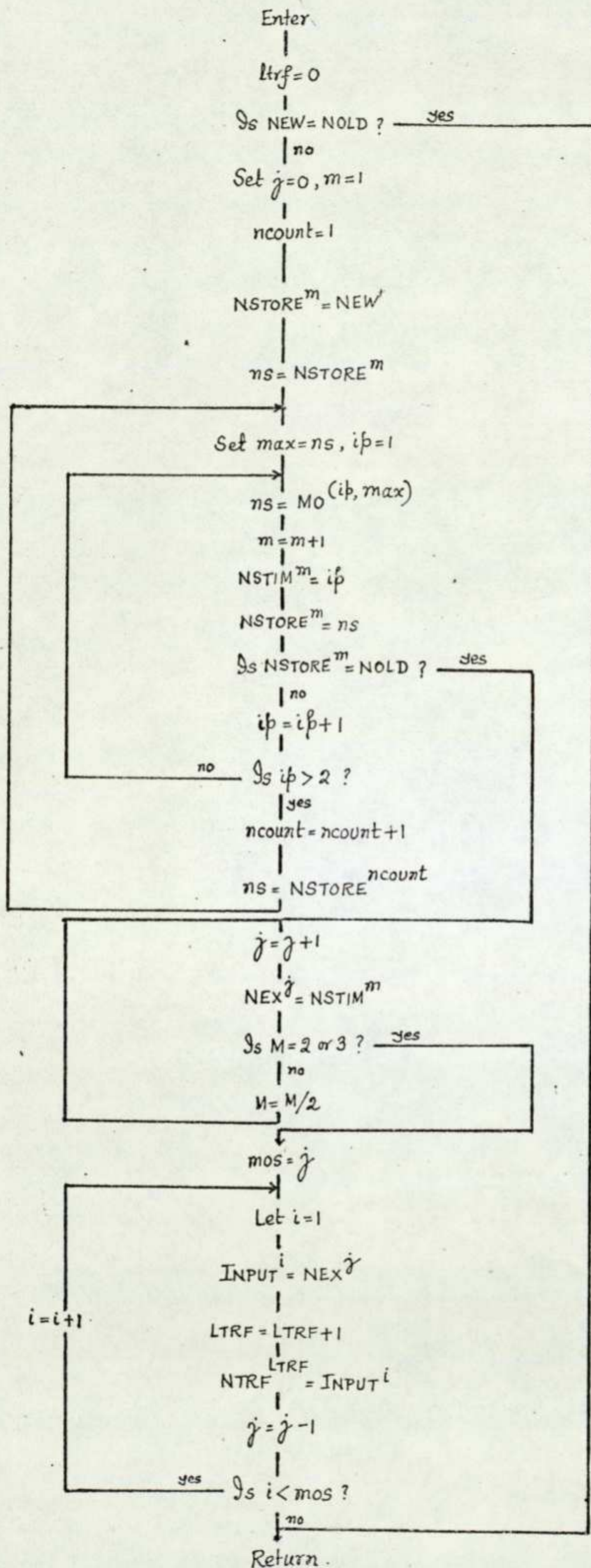


Fig 4.9 Flow-diagram of TRASEQ.

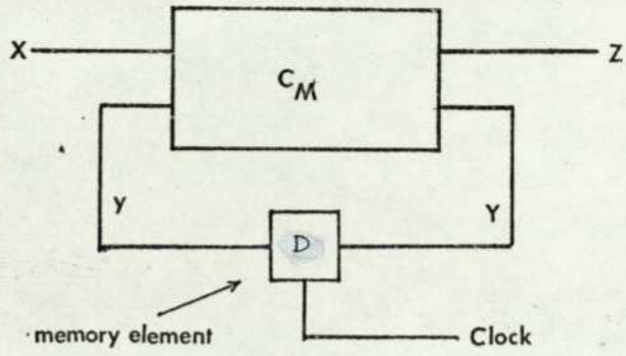
inputs. Hence if all the feedback loops are cut, assuming the secondary variables as pseudo-primary inputs, a pure combinational circuit is obtained as shown in Fig.4.10a and b. By applying the previously described algorithm to the circuit of Fig.4.10b, a minimal test set for the circuit can be obtained. The test values at the pseudo-primary inputs are in fact the states at which the circuit has to be taken before the test value at the primary input can be applied to test the combinational part of the network. Thus the minimal test set becomes the minimum number of states at which the circuit is to be tested before concluding whether it is fault-free or not. In other words, fault-folding can be applied to find the minimum number of states to use in deriving a checking sequence which will detect all faults.

It is clear from Fig.4.10a that is is impossible to apply the test input combination directly to the combinational part of the circuit and observe the output response. Hence an indirect approach is used (93). Since it is not possible to use the secondary inputs and secondary outputs in real time, a sequence of inputs is applied at the primary input to bring the circuit into the pre-determined states and then apply the test value at the primary input such that together they correspond to the desired input to the combinational logic part of the network. Henceforth, the pre-determined states obtained by fault-folding will be referred to as the "test-states".

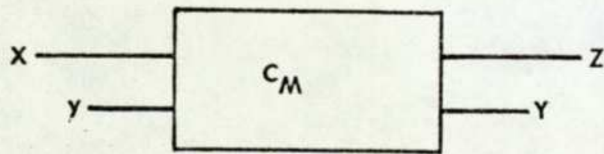
4.3.2 An algorithm for checking sequence generation of synchronous sequential circuits

In this section a procedure will be described to derive a checking sequence for synchronous sequential circuits. Before describing the procedure, the following assumptions will be made:

(i) Faults may occur at the combinational or at the feedback part of the sequential circuit.



a. Synchronous sequential circuit model



b. Combinational version of (a)

Fig 4.10

(ii) If some gate in the input circuit of the flip-flop becomes faulty which makes the output of one side of the flip-flop stuck-at some value, then the output of another side of that flip-flop will take the value complementary to the stuck-at value i.e. if the output of one side of a flip-flop Q_i is stuck-at-1(0) the output of another side of that flip-flop $\bar{Q}_i$ takes on the complementary value of 0 (1).

(iii) The clock circuit is fault-free.

(iv) The input operates normally at any time, since it is assumed to be provided from an external circuit.

The following steps are followed sequentially for implementing the test-sequence generating algorithm:-

(1) Transform the synchronous sequential circuit (Fig.4.10a) into the combinational model as shown in Fig.4.10b, where C_M is the combinational part of the circuit.

(2) Apply the test generating algorithm to the combinational circuit of Fig.4.10b. Test values at the pseudo-primary inputs for a particular test, is the state to which the sequential circuit is to be taken before the test value at the primary input is applied.

(3) Find a homing sequence for the circuit from its state-table; a procedure has been described in section 4.2.1.

(4) Apply the homing sequence and determine what state the circuit is in.

(5) If the circuit is in one of the 'test states', apply the test value associated with that state, otherwise apply the transfer sequence to take the circuit to a test state and then apply the test value. Determine what state the circuit is in after the application of the test value.

(6) If the circuit has been tested at all the test states determined in step 2, go to step 7, otherwise go to step 5.

(7) Concatenate the input sequences of steps 4-6 which form the checking sequence for the circuit. Concatenation of two sequences I_1 and I_2, denoted by $I_1.I_2$, represents the sequence consisting of the sequence I_1 followed by the sequence I_2. Usually the symbol for concatenation (.) is omitted.

The output response, corresponding to each input symbol of the checking sequence, is recorded both for the present state (before clocking) and the next state (after clocking) of the circuit (94). Rutman (95) showed that certain faults are not detected if the output measurement before clocking is not taken.

The flow-diagram of the test sequence generation procedure is shown in Fig.4.11.

4.3.3 An example of the application of the algorithm

The procedure outlined in the previous algorithm will be used to generate the test sequence for the circuit of Fig.4.12. The state table of the circuit is shown in Table 4.6.

(1) The combinational form of the circuit is shown in Fig.4.13. It was assumed that if the Q output of the flip-flop was stuck-at-x (0 or 1), the $\bar{Q}$ output of the flip-flop took the value $\bar{x}$. Hence while converting a sequential circuit to its combinational model, dummy inverters were put (to simulate $\bar{Q}$ output) in between the Q output and the gate of the combinational circuit which the $\bar{Q}$ output was supposed to feed. The dummy inverters are shown within dotted rings in Fig.4.13, where 1 is the primary input and 2,3 are pseudo-primary inputs.

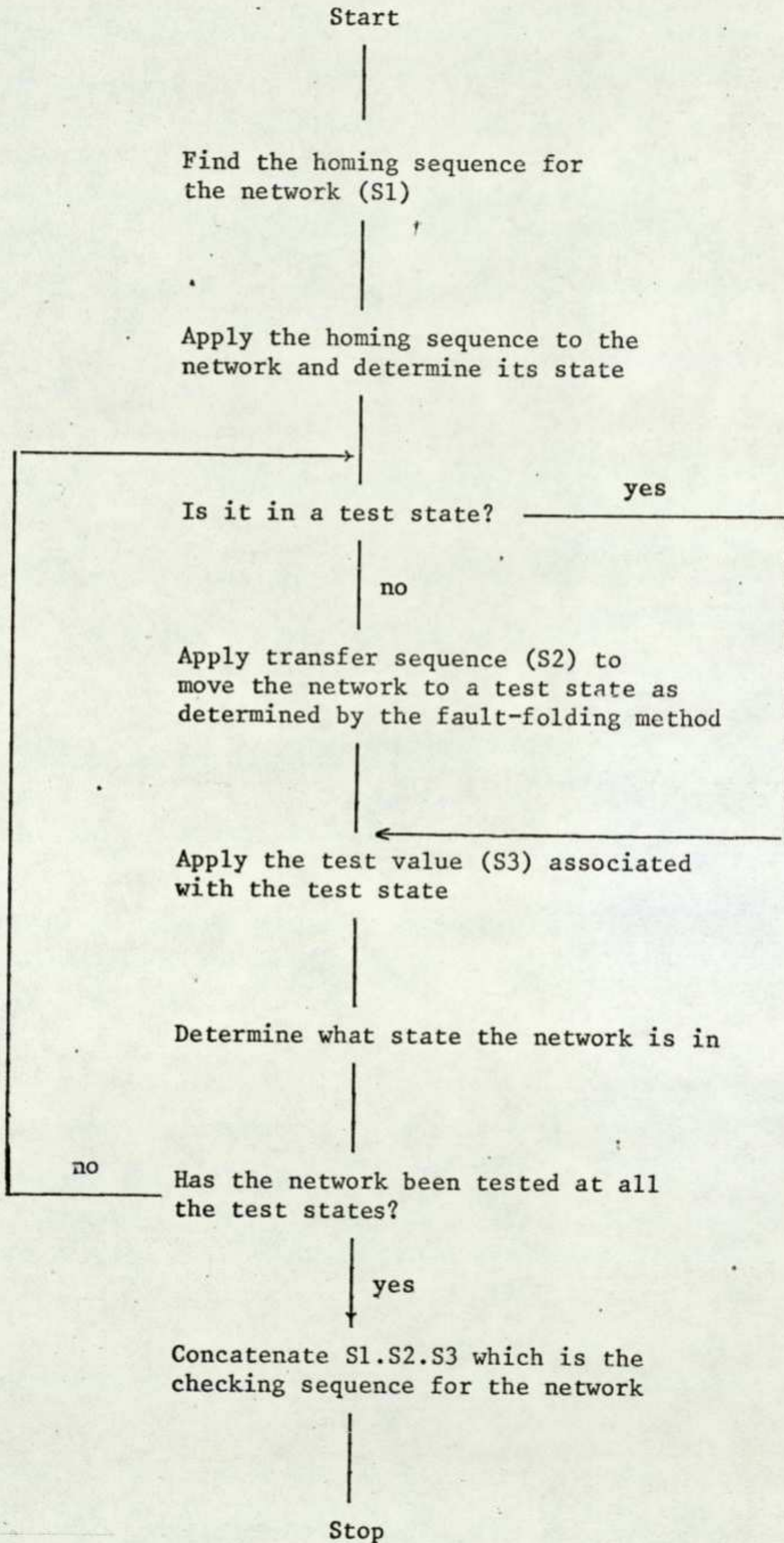


Fig.4.11 The Checking Sequence Generation Algorithm

Fig 4.12 Circuit example

Q_1Q_2 \ x	0	1
00	00, 0	01, 0
01	00, 0	11, 1
11	01, 0	10, 0
10	11, 1	10, 0

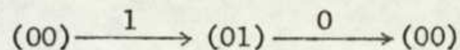
Table 4.6 State table of the circuit

(2) The fault-folded graphs for the circuit of Fig.4.13 is shown in Fig.4.14. By subsequent application of the test generating algorithm for the combinational circuits, the test set |110,010,101,001| is found for the combinational circuit of Fig.4.13.

Hence the sequential circuit has to be tested at state 4 (Q_1Q_2=10)and at state 2(Q_1Q_2 = 01) with test values 1 and 0.

(3) 10 is a homing sequence for the sequential circuit.

(4) Assuming the circuit is initially in state 1(Q_1Q_2=00), it will remain at state 1 after the application of the homing sequence,



(5) By applying the transfer sequence 111 the circuit is
and
(6) manoeuvred to test state 4 from state 1 and the test value 1 is applied. The circuit still remains at test state 4 after the application of the test value 1 so the other test value 0 is applied. The circuit now moves to state 3 (Q_1Q_2=11); the circuit is then manoeuvred to the test state 2 with the help of the transfer sequence 0 and the test value 1 is applied. The circuit moves to state 3; it is then manoeyred back to state 2 by applying 0 and finally the test value 0 is applied.

(7) By concatenating the input sequences, the checking sequence for the circuit is

Preset	Adaptive
1 0	1 1 1 1 0 0 1 0 0

and the corresponding output sequences are

0 0	0 1 0 0 1 0 1 0 0	before clocking
1 0	1 0 0 0 0 0 0 0 0	after clocking

Obviously the adaptive part of the checking sequence will

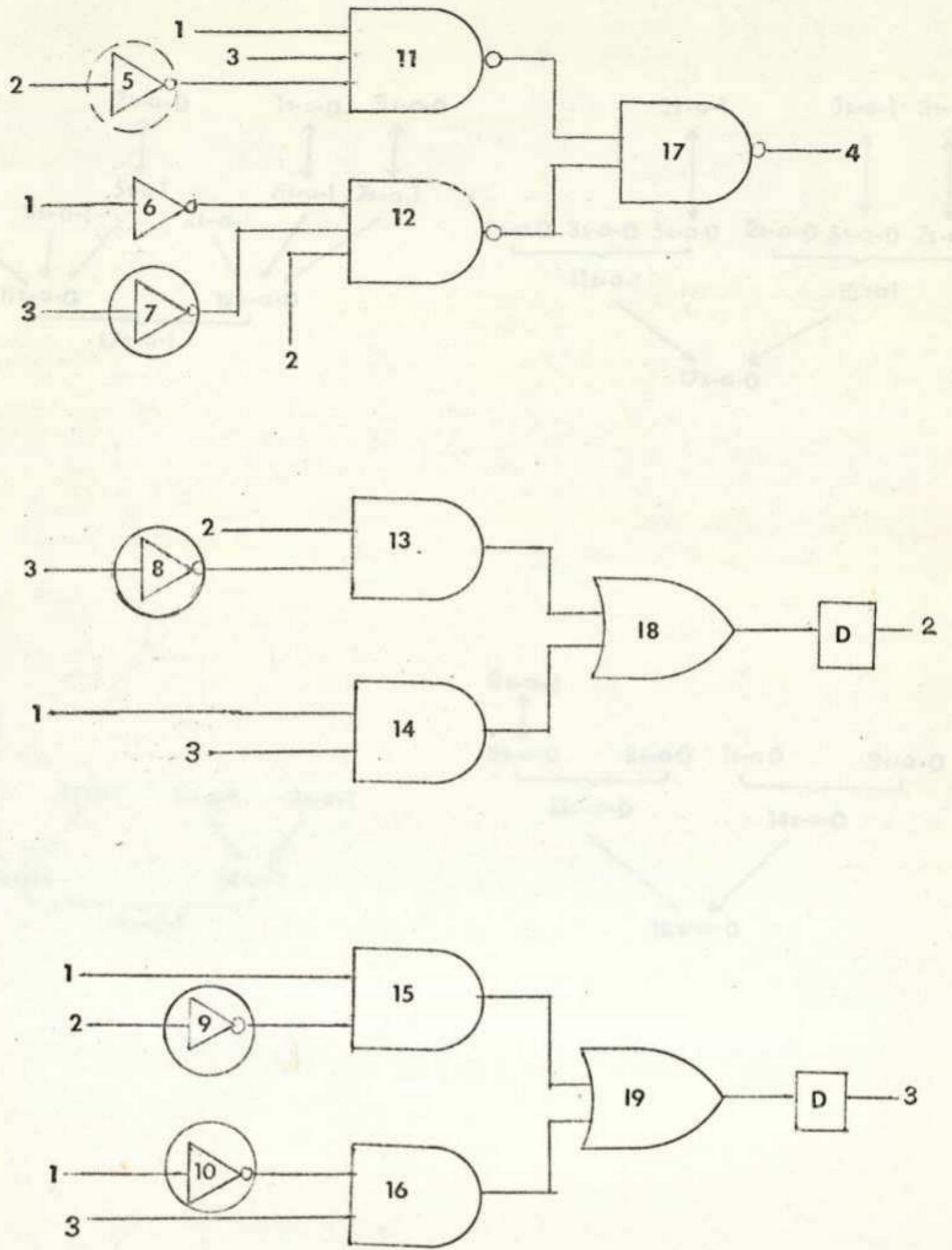


Fig 4.13 Combinational form of the circuit of Fig 4.12

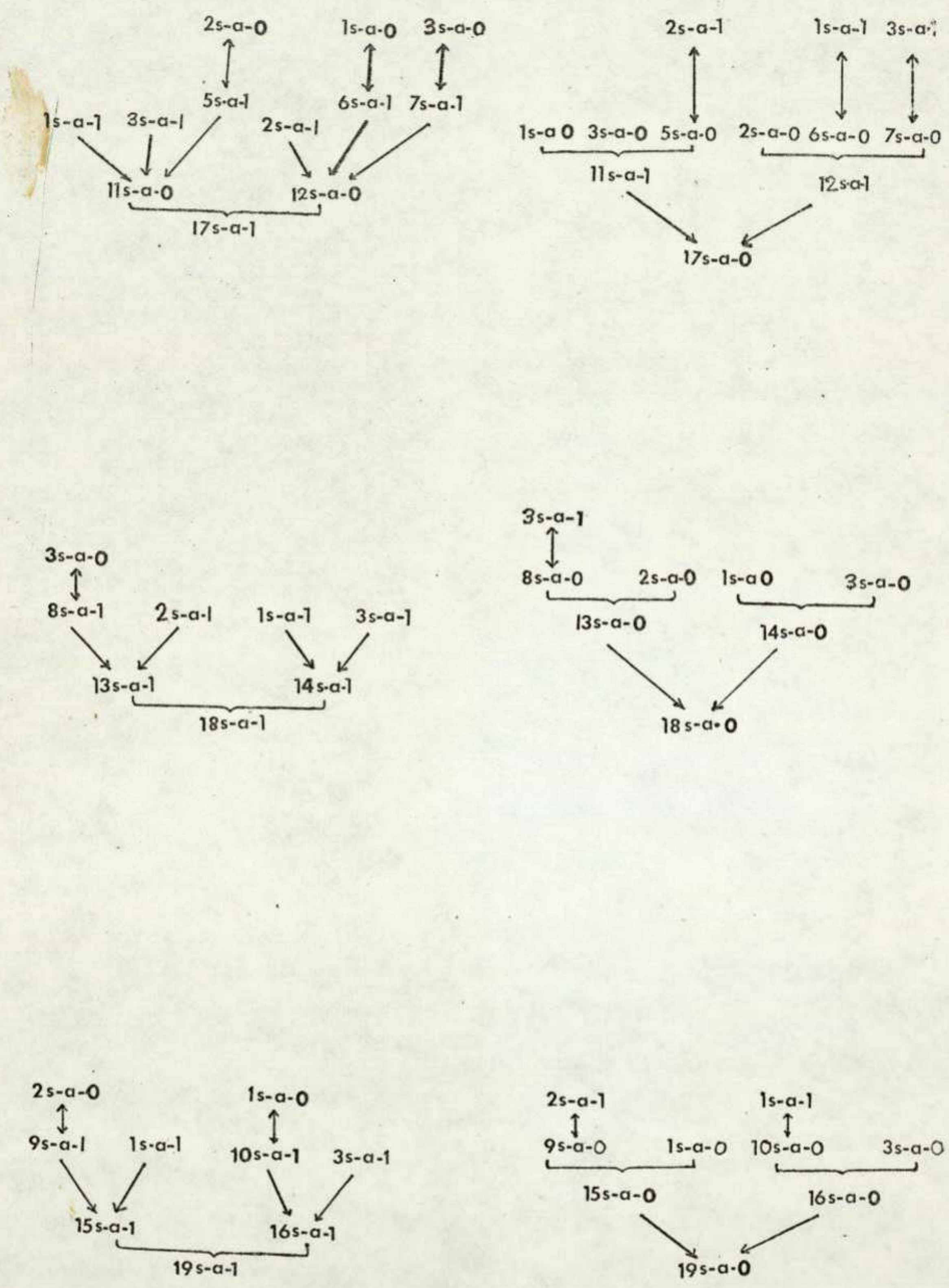


Fig 4.14 Fault-folded graphs

differ if the circuit response to the homing sequence is different i.e. the circuit is in a different final state. If the circuit were initially in state 2 the test input and output sequences would have been

Input 1 0 1 1 1 0 0 1 0 0

Output 1 0 1 0 0 1 0 1 0 0 before clocking

0 0 0 0 0 0 0 0 0 0 after clocking

Similarly for the initial state 3 or 4,

Input 1 0 1 1 0 0 1 0 0

Output 0 1 0 0 1 0 1 0 0 before clocking

0 0 0 0 0 0 0 0 0 after clocking

4.3.4 Computer simulation of the checking sequence generating algorithm

A programme has been written to automate the checking sequence generation procedure; it requires as input the circuit data structure (see section 2.3.3), an assumed initial state, a homing sequence, the test states and associated test values. The flow-diagram of the programme LINE is shown in Fig.4.15a, it has three subroutines LOGIC, NEWSTATE and TRASEQ. The variables and arrays used in the programme have the following definitions and uses:-

LHMO	length of the homing sequence
NS	number of test states
N	total number of elements in the circuit
NFF	total number of flip-flops in the circuit
NPRIM, NOUT	number of primary inputs and outputs respectively in the circuit
NTYPE	type of flip-flop being used in the circuit; only one type of flip-flop is assumed to be used in the circuit. NTYPE = 1,2 for D,JK flip-flop respectively.

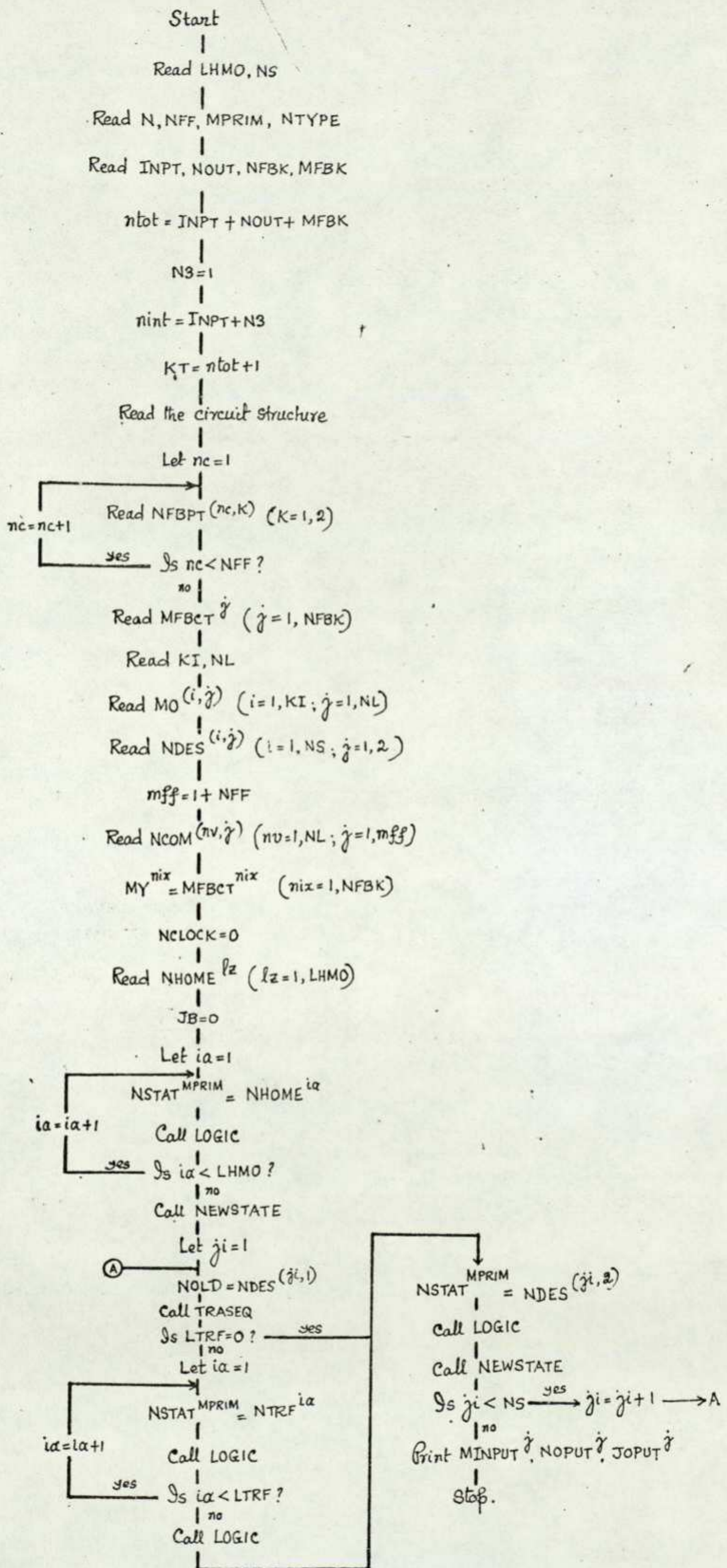


Fig. 4.15a Flow-diagram of LINE

INPT	total number of inputs (primary and pseudo primary) in the combinational circuit
MFBK, NFBK	number of inputs and outputs respectively of the feedback circuit
MFBCT	a one-dimensional array of variables MFBCT(I) which stores the logical state of the circuit and its complement value
NFBPT	a two-dimensional array of variables NFBPT(I,L) which stores the J and K input connections of the ith flip-flop in ℓth positions. (Note: L=2 and for the D flip-flop NFBPT(I,2) is set to zero.)
MO	a two-dimensional array of variables MO(I,L) which stores the next-state of the ℓth state, resulting from the ith input application.
NDES	a two-dimensional array of variables NDES(I,L) which stores a test state in the ith position and the associated test value in the ℓth position
NCOM	a two-dimensional array of variables NCOM(I,L) which stores the abstract symbol of the ith state of the circuit in the ℓth position.
NHOME	a one-dimensional array of NHOME(I) variables which stores the homing sequence of the circuit
MY	a one-dimensional array of variables MY(I) which stores the logical state of the circuit and its complement value
JB	an input variable specifying the length of the test sequence
NCLOCK	an integer variable which holds the clock pulse being applied

Subroutine LOGIC : it calculates the logical states of the various elements in the circuit, corresponding to a test input; the circuit output both before and after the application of the clock pulse is recorded. The flow-diagram of LOGIC is shown in 4.15b; the function evaluation routines for the gates and the flip-flops have been described previously in section 2.3. The arrays and variables have the same meanings as before; additional arrays used in the subroutine have the following uses

MINPUT	a one-dimensional array of variable MINPUT(I) which stores the input symbols of the test sequence
NOPUT	a one-dimensional array of variables NOPUT(I) which stores the output symbols before the clock pulse
JOPUT	a one-dimensional array of variables JOPUT(I) which stores the output symbols after the clock pulse.

Subroutine NEWSTATE : the flow-diagram of the subroutine is shown in Fig.4.15c; it records the state of the circuit in its symbolic form (integer variable NEW in the flow-diagram) after the application of an input sequence.

Subroutine TRASEQ : the function of this subroutine has been described in section 4.2.2.

4.3.5 Computer-aided generation of a test sequence : a circuit example

The digital traffic light controller circuit described in (96) will be used for illustration. The circuit diagram is shown in Fig.4.16. The decimal codes assigned to each memory state of the circuit are recorded in Table 4.7a, the state table is shown in Table 4.7b.

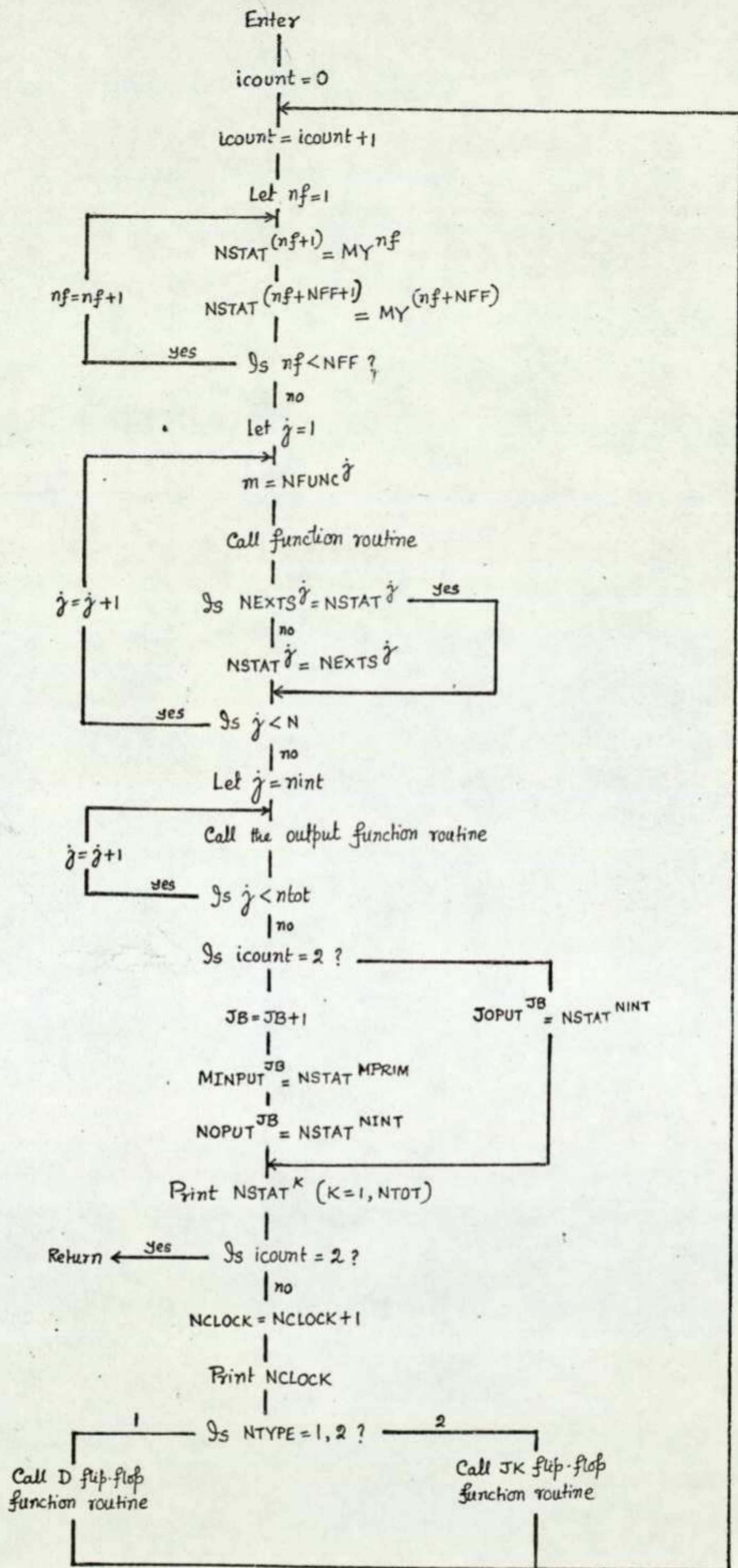


Fig 4.15b Flow-diagram of LOGIC

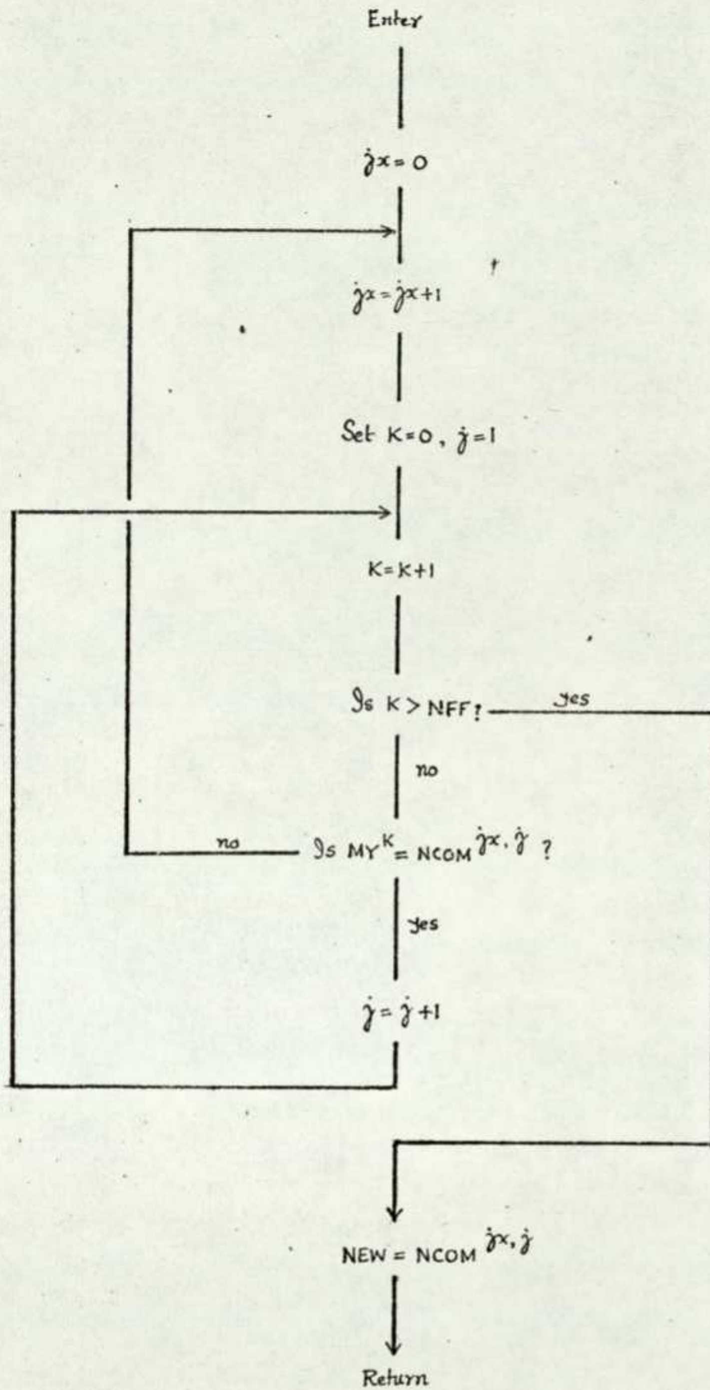


Fig 4.15c Flow-diagram of NEWSTATE

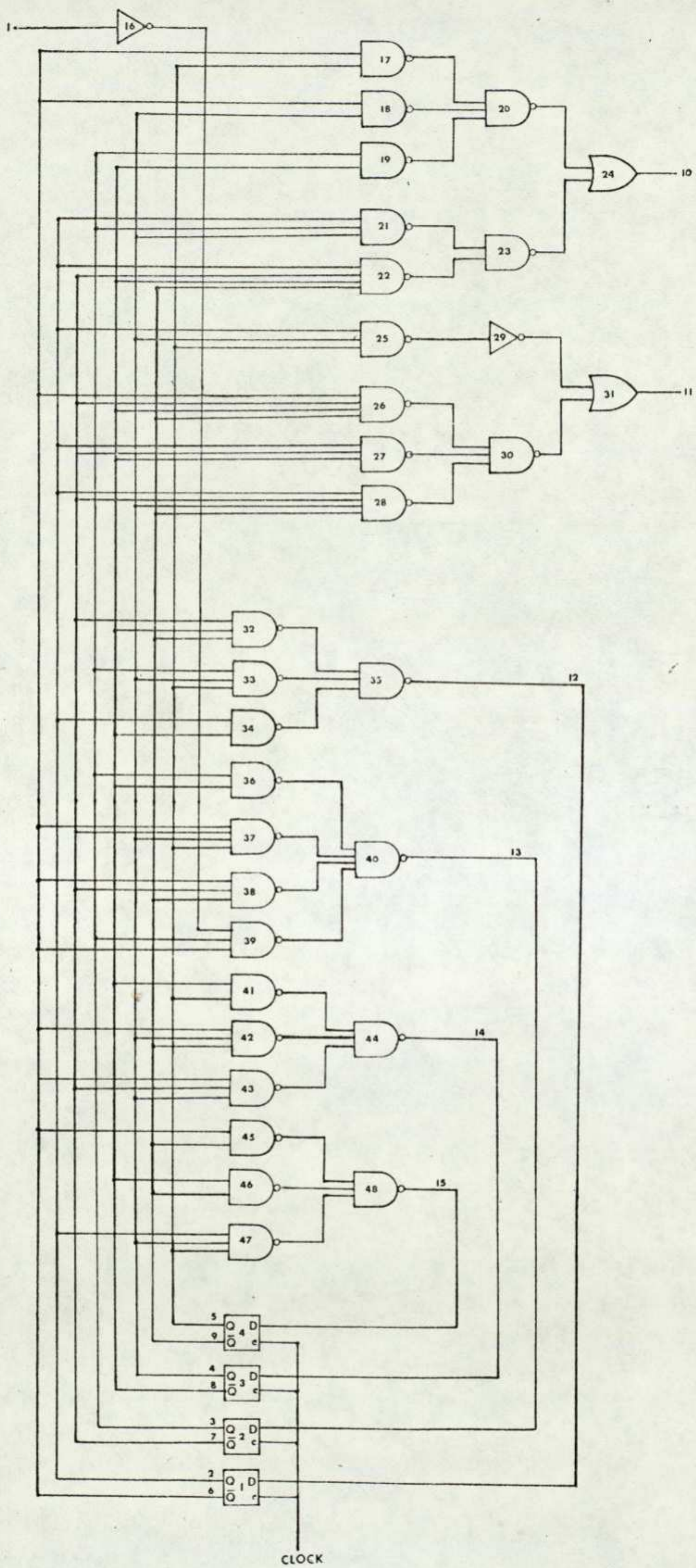


Fig 4.16 Traffic Light Controller Circuit

STATE NUMBER	ASSIGNMENT (BINARY CODE)
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	0000
12	1011
13	1100
14	1101
15	1110

a. Assignment table

PRESENT STATE	NEXT STATE / OUTPUT X=0	X=1
1	2,1	2,1
2	3,1	3,1
3	4,1	4,1
4	5,1	5,1
5	6,1	6,1
6	7,1	7,1
7	13,1	8,1
8	9,0	9,1
9	10,0	10,0
10	11,0	11,0
11	12,0	12,0
12	1,1	1,0
13	14,1	14,1
14	15,1	15,1
15	11,1	11,1

b. State table
(Output observed at element 10)

Table 4.7

The circuit was first converted into its combinational form (Fig 4.17). The test-set obtained for the circuit of Fig 4.17, after applying step 2 of the algorithm is shown in Fig 4.18. Each test corresponds to a test state for the sequential circuit as shown in Fig 4.19. In this case, test values are applied at the primary input to change the circuit to a test state; once it reaches a test state the primary input is assigned with a "dont care" test value. By this means, more than 80% of the total faults in the circuit can be detected.

The data input for automatic test sequence generation for the circuit of Fig.4.16 is shown in Fig.4.20. A homing sequence (11111111) was derived by using the programme HOME; the computer time used to generate it was about 5 secs. Assuming that the circuit is initially in state 15 ($Q_1Q_2Q_3Q_4$= 1110), the checking sequence generated is shown in Fig.4.21. The derivation procedure may be followed step by step in Fig.4.21; only the logical states of the inputs and outputs (primary and secondary) are printed.

The programme has been designed to handle synchronous sequential circuits having up to 50 elements and 15 states, and uses a core store of about 8K. It took about 16 secs. of computer time to generate the test sequence shown in Fig.4.21.

4.4 Fault Location in Synchronous Sequential Circuits

4.4.1 Programme development for fault simulation in synchronous sequential circuits.

A computer programme has been written to simulate the fault detection procedure in synchronous sequential circuits; the flow-diagram of the programme LAND is shown in Fig.4.22. The arrays and variables

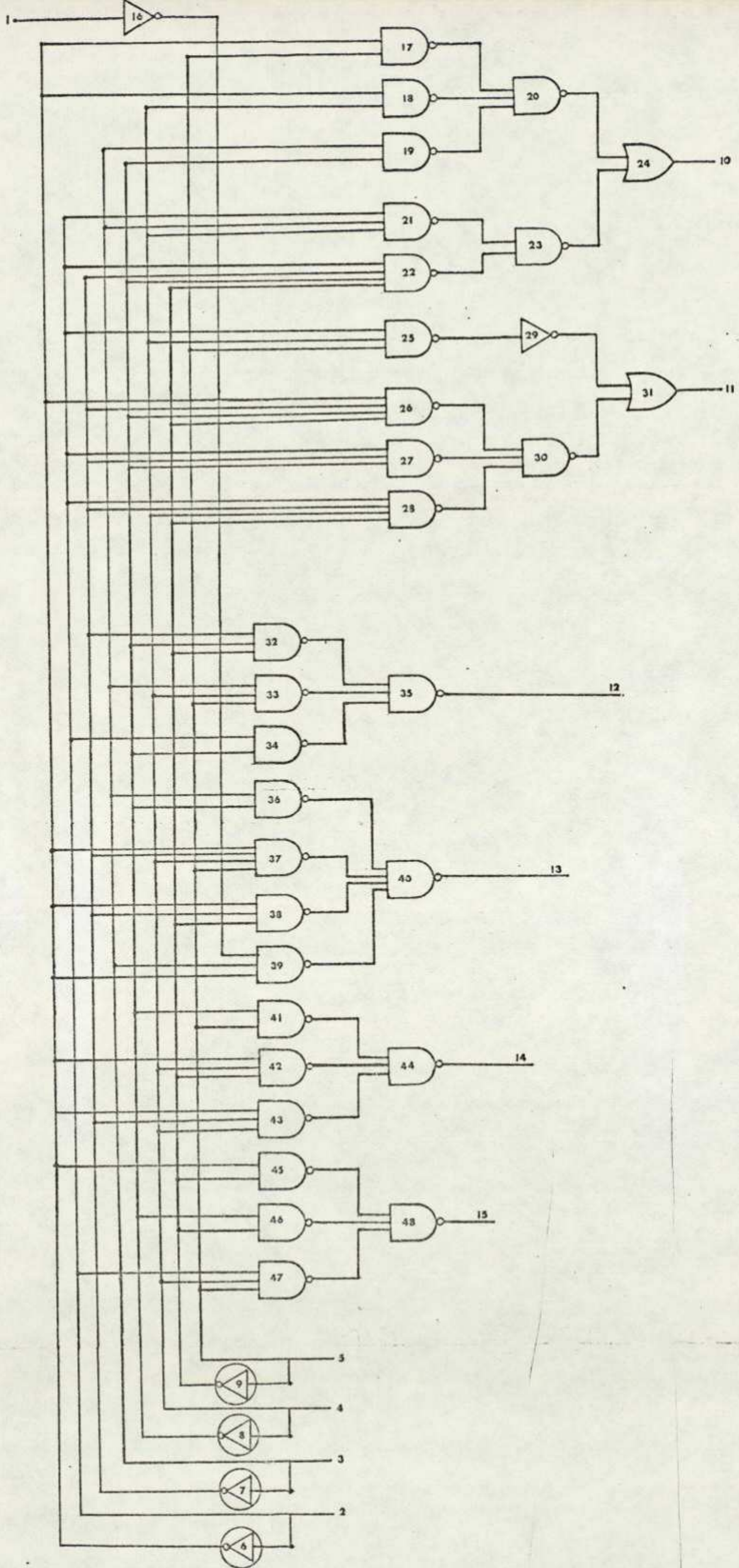


Fig 4.17 Combinational form of the circuit of Fig 4.16

Test Input 2 3 4 5	Indistinguishable fault-set
0 0 1 0	34 s-a-0, 35 s-a-1, 33 s-a-0, 32 s-a-0, 18 s-a-1, 20 s-a-0, 24 s-a-0, 45-s-a-1, 48 s-a-0
0 0 0 0	32 s-a-1, 35 s-a-0, 17 s-a-0, 20 s-a-1, 18 s-a-0, 19 s-a-0, 24 s-a-1, 23 s-a-1, 21 s-a-0, 22 s-a-0, 26 s-a-1, 30 s-a-0, 31 s-a-0, 38 s-a-1
1 0 0 1	34 s-a-1, 35 s-a-0, 27 s-a-1, 30 s-a-0, 31 s-a-0, 36 s-a-0, 37 s-a-0, 38 s-a-0, 39 s-a-0, 40 s-a-1
0 1 0 0	36 s-a-1, 40 s-a-0, 19 s-a-1, 20 s-a-0, 24 s-a-0
1 0 1 0	45 s-a-0, 46 s-a-0, 47 s-a-0, 48 s-a-1, 28 s-a-1, 30 s-a-0, 31 s-a-0, 41 s-a-0, 42 s-a-0, 43 s-a-0, 44 s-a-1
1 0 0 0	46 s-a-1, 48 s-a-0, 22 s-a-1, 23 s-a-0, 24 s-a-0
1 0 1 1	47 s-a-1, 48 s-a-0, 25 s-a-1, 29 s-a-0, 31 s-a-0
0 0 0 1	41 s-a-1, 44 s-a-0, 17 s-a-1, 20 s-a-0, 24 s-a-0
1 1 1 0	26 s-a-0, 30 s-a-1, 27 s-a-0, 28 s-a-0, 31 s-a-1, 29 s-a-1, 25 s-a-0

Fig 4.18 Test set.

Q_1	Q_2	Q_3	Q_4	State
0	0	0	1	1
0	0	1	1	2
0	1	0	0	4
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
0	0	0	0	11
1	0	1	1	12
1	1	1	0	15

Fig 4.19 Test states.

```
8 9
48 4 1 1
9 2 8 4
1 1 0 16
2 1 0 21 22 25 27 28 34 47
3 1 0 19 21 36 33 38
4 1 0 18 21 25 28 33 37 42 47
5 1 0 17 25 27 33 37 41 47
6 1 0 17 18 26 37 38 39 42 45
7 1 0 22 26 27 28 32 37 43
8 1 0 19 22 26 27 32 34 36 41 46
9 1 0 22 26 28 32 38 42 43 45 46
10 2 24
11 2 31
12 2 35
13 2 40
14 2 44
15 2 48
16 7 1 0 39
17 5 5 6 0 20
18 5 4 6 0 20
19 5 3 8 0 20
20 5 17 18 19 0 24
21 5 2 3 4 0 23
22 5 2 7 8 9 0 23
23 5 21 22 0 24
24 4 20 23 0 10
25 5 2 4 5 0 29
26 5 6 7 8 9 0 30
27 5 2 5 7 8 0 30
28 5 2 4 7 9 0 30
29 7 25 0 31
30 5 26 27 28 0 31
31 4 29 30 0 11
32 5 7 8 9 0 35
33 5 3 4 5 0 35
34 5 2 8 0 35
35 5 32 33 34 0 12
36 5 3 8 0 40
37 5 4 5 6 7 0 40
38 5 3 6 9 0 40
39 5 3 6 16 0 40
40 5 36 37 38 39 0 13
41 5 5 8 0 44
42 5 4 6 9 0 44
43 5 6 7 9 0 44
44 5 41 42 43 0 14
45 5 6 9 0  48
46 5 8 9 0 48
47 5 2 4 5 0 48
48 5 45 46 47 0 15
12 0
13 0
14 0
15 0
2 2 2 1 1 1 1 2
2 15
2 2 3 3 4 4 5 5 6 6 7 7 13 8 9 9 10 10 11 11 12 12 1 1
14 14 15 15 11 11
1 2 2 2 4 2 8 2 9 2 10 2 11 2 12 2 15 2
1 1 1 2 1
1 1 2 1 2
1 1 2 2 3
1 2 1 1 4
1 2 1 2 5
1 2 2 1 6
1 2 2 2 7
2 1 1 1 8
2 1 1 2 9
2 1 2 1 10
1 1 1 1 11
2 1 2 2 12
2 2 1 1 13
2 2 1 2 14
2 2 2 1 15
2 2 2 2 2 2 2 2
```

Fig 4.20 Data input for program LINE

```
APPLY HOMING SEQUENCE

APPLY INPUT
 2 2 2 2 1 1 1 1 2 2 1 1 1 1 1
APPLY  CLOCK PULSE  1
 2 1 1 1 1 2 2 2 2 1 2 2 1 2 2

APPLY INPUT
 2 1 1 1 1 2 2 2 2 1 2 2 1 2 2
APPLY  CLOCK PULSE  2
 2 2 1 2 2 1 2 1 1 1 2 1 1 1 2

APPLY INPUT
 2 2 1 2 2 1 2 1 1 1 2 1 1 1 2
APPLY  CLOCK PULSE  3
 2 1 1 1 2 2 2 2 1 2 1 1 1 2 1

APPLY INPUT
 2 1 1 1 2 2 2 2 1 2 1 1 1 2 1
APPLY  CLOCK PULSE  4
 2 1 1 2 1 2 2 1 2 2 1 1 1 2 2

APPLY INPUT
 2 1 1 2 1 2 2 1 2 2 1 1 1 2 2
APPLY  CLOCK PULSE  5
 2 1 1 2 2 2 2 1 1 2 1 1 2 1 1

APPLY INPUT
 2 1 1 2 2 2 2 1 1 2 1 1 2 1 1
APPLY  CLOCK PULSE  6
 2 1 2 1 1 2 1 2 2 2 1 1 2 1 2

APPLY INPUT
 2 1 2 1 1 2 1 2 2 2 1 1 2 1 2
APPLY  CLOCK PULSE  7
 2 1 2 1 2 2 1 2 1 2 1 1 2 2 1

APPLY INPUT
 2 1 2 1 2 2 1 2 1 2 1 1 2 2 1
APPLY  CLOCK PULSE  8
 2 1 2 2 1 2 1 1 2 2 1 1 2 2 2

PRESENT STATE  6

APPLY TRANSFER SEQUENCE

APPLY INPUT
 1 1 2 2 1 2 1 1 2 2 1 1 2 2 2
APPLY  CLOCK PULSE  9
 1 1 2 2 2 2 1 1 1 2 1 2 2 1 1

APPLY INPUT
 1 1 2 2 2 2 1 1 1 2 1 2 2 1 1
APPLY  CLOCK PULSE 10
 1 2 2 1 1 1 1 2 2 2 1 2 2 1 2

APPLY INPUT
 1 2 2 1 1 1 1 2 2 2 1 2 2 1 2
APPLY  CLOCK PULSE 11
 1 2 2 1 2 1 1 2 1 2 1 2 2 2 1

APPLY INPUT
 1 2 2 1 2 1 1 2 1 2 1 2 2 2 1
APPLY  CLOCK PULSE 12
 1 2 2 2 1 1 1 1 2 2 1 1 1 1 1

APPLY INPUT
 1 2 2 2 1 1 1 1 2 2 1 1 1 1 1
APPLY  CLOCK PULSE 13
 1 1 1 1 1 2 2 2 2 1 2 2 1 2 2

APPLY INPUT
 1 1 1 1 1 2 2 2 2 1 2 2 1 2 2
APPLY  CLOCK PULSE 14
 1 2 1 2 2 1 2 1 1 1 2 1 1 1 2

APPLY INPUT
 1 2 1 2 2 1 2 1 1 1 2 1 1 1 2
APPLY  CLOCK PULSE 15
 1 1 1 1 2 2 2 2 1 2 1 1 1 2 1

PRESENT STATE  1
```

```
APPLY TEST VALUE

APPLY INPUT
 2 1 1 1 2 2 2 2 1 2 1 1 1 2 1
APPLY  CLOCK PULSE 16
 2 1 1 2 1 2 2 1 2 2 1 1 1 2 2

PRESENT STATE  2

APPLY TEST VALUE

APPLY INPUT
 2 1 1 2 1 2 2 1 2 2 1 1 1 2 2
APPLY  CLOCK PULSE 17
 2 1 1 2 2 2 2 1 1 2 1 1 2 1 1

PRESENT STATE  3

APPLY TRANSFER SEQUENCE

APPLY INPUT
 1 1 1 2 2 2 2 1 1 2 1 1 2 1 1
APPLY  CLOCK PULSE 18
 1 1 2 1 1 2 1 2 2 2 1 1 2 1 2

PRESENT STATE  4

APPLY TEST VALUE

APPLY INPUT
 2 1 2 1 1 2 1 2 2 2 1 1 2 1 2
APPLY  CLOCK PULSE 19
 2 1 2 1 2 2 1 2 1 2 1 1 2 2 1

PRESENT STATE  5

APPLY TRANSFER SEQUENCE

APPLY INPUT
 1 1 2 1 2 2 1 2 1 2 1 1 2 2 1
APPLY  CLOCK PULSE 20
 1 1 2 2 1 2 1 1 2 2 1 1 2 2 2

APPLY INPUT
 1 1 2 2 1 2 1 1 2 2 1 1 2 2 2
APPLY  CLOCK PULSE 21
 1 1 2 2 2 2 1 1 1 2 1 2 2 1 1

APPLY INPUT
 2 1 2 2 2 2 1 1 1 2 1 2 1 1 1
APPLY  CLOCK PULSE 22
 2 2 1 1 1 1 2 2 2 2 1 2 1 1 2

PRESENT STATE  8

APPLY TEST VALUE

APPLY INPUT
 2 2 1 1 1 1 2 2 2 2 1 2 1 1 2
APPLY  CLOCK PULSE 23
 2 2 1 1 2 1 2 2 1 1 2 2 1 2 1

PRESENT STATE  9

APPLY TEST VALUE

APPLY INPUT
 2 2 1 1 2 1 2 2 1 1 2 2 1 2 1
APPLY  CLOCK PULSE 24
 2 2 1 2 1 1 2 1 2 1 2 1 1 1 1

PRESENT STATE 10
```

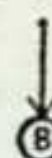


Fig 4.21 Output of program LINE

(cont'd)

Ⓑ

APPLY TEST VALUE

APPLY INPUT
2 2 1 2 1 1 2 1 2 1 2 1 1 1 1
APPLY CLOCK PULSE 25
2 1 1 1 1 2 2 2 2 1 2 2 1 2 2

PRESENT STATE 11

APPLY TEST VALUE

APPLY INPUT
2 1 1 1 1 2 2 2 2 1 2 2 1 2 2
APPLY CLOCK PULSE 26
2 2 1 2 2 1 2 1 1 1 2 1 1 1 2

PRESENT STATE 12

APPLY TEST VALUE

APPLY INPUT
2 2 1 2 2 1 2 1 1 1 2 1 1 1 2
APPLY CLOCK PULSE 27
2 1 1 1 2 2 2 2 1 2 1 1 1 2 1

PRESENT STATE 1

Ⓒ

Ⓒ

APPLY TRANSFER SEQUENCE

APPLY INPUT
1 1 1 1 2 2 2 2 1 2 1 1 1 2 1
APPLY CLOCK PULSE 28
1 1 1 2 1 2 2 1 2 2 1 1 1 2 2

APPLY INPUT
1 1 1 2 1 2 2 1 2 2 1 1 1 2 2
APPLY CLOCK PULSE 29
1 1 1 2 2 2 2 1 1 2 1 1 2 1 1

APPLY INPUT
1 1 1 2 2 2 2 1 1 2 1 1 2 1 1
APPLY CLOCK PULSE 30
1 1 2 1 1 2 1 2 2 2 1 1 2 1 2

APPLY INPUT
1 1 2 1 1 2 1 2 2 2 1 1 2 1 2
APPLY CLOCK PULSE 31
1 1 2 1 2 2 1 2 1 2 1 1 2 2 1

APPLY INPUT
1 1 2 1 2 2 1 2 1 2 1 1 2 2 1
APPLY CLOCK PULSE 32
1 1 2 2 1 2 1 1 2 2 1 1 2 2 2

APPLY INPUT
1 1 2 2 1 2 1 1 2 2 1 1 2 2 2
APPLY CLOCK PULSE 33
1 1 2 2 2 2 1 1 1 2 1 2 2 1 1

APPLY INPUT
1 1 2 2 2 2 1 1 1 2 1 2 2 1 1
APPLY CLOCK PULSE 34
1 2 2 1 1 1 1 2 2 2 1 2 2 1 2

APPLY INPUT
1 2 2 1 1 1 1 2 2 2 1 2 2 1 2
APPLY CLOCK PULSE 35
1 2 2 1 2 1 1 2 1 2 1 2 2 2 1

APPLY INPUT
1 2 2 1 2 1 1 2 1 2 1 2 2 2 1
APPLY CLOCK PULSE 36
1 2 2 2 1 1 1 1 2 2 1 1 1 1 1

PRESENT STATE 15

APPLY TEST VALUE

APPLY INPUT
2 2 2 2 1 1 1 1 2 2 1 1 1 1 1
APPLY CLOCK PULSE 37
2 1 1 1 1 2 2 2 2 1 2 2 1 2 2

PRESENT STATE 11

<u>I/P</u> 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 2 2 1 2 1 1 2 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 2

<u>O/P</u> 2 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 1 1 1 1 2 2 2 2 2 2 2
2 2 2
BEFORE CLOCK PULSE

<u>O/P</u> 1 1 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 1 1 1 1 2 2 2 2 2 2 2
2 2 1
AFTER CLOCK PULSE

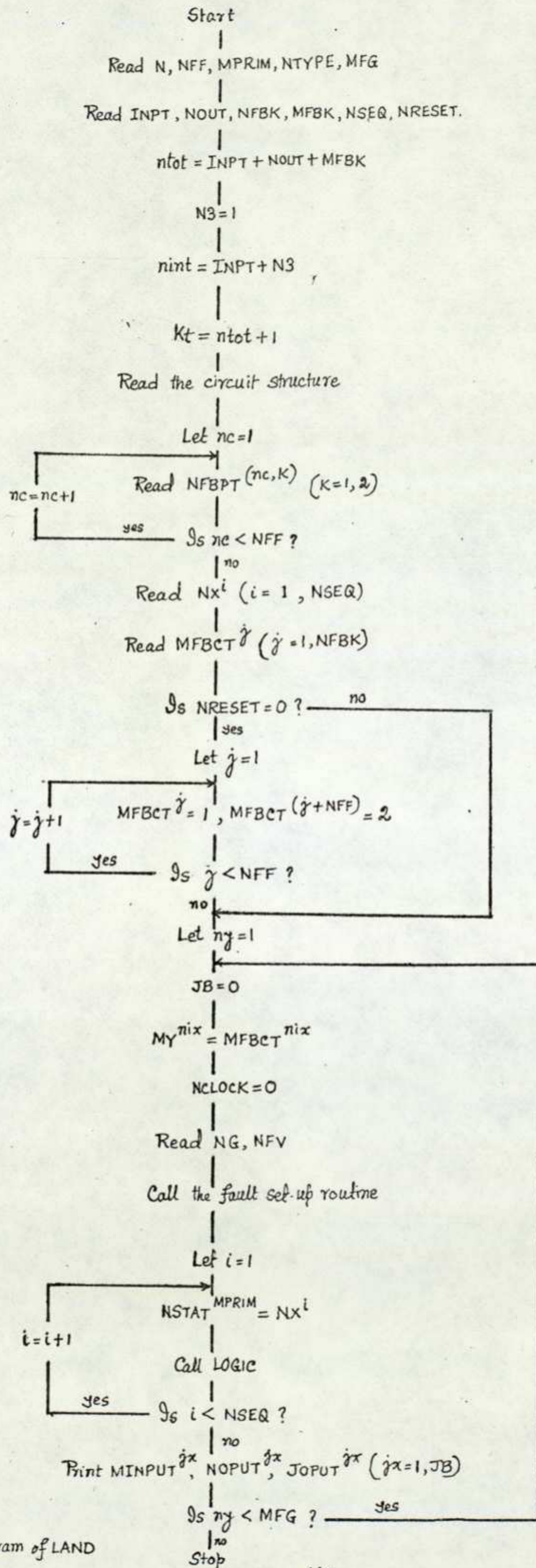


Fig. 4. 22 Flow-diagram of LAND

used in the programme have the same meaning and uses as in the programme for checking sequence generation in section 4.3.4; subroutine LOGIC has also been described in that section. The additional arrays and variables used in the programme are defined as follows:-

NSEQ	length of the test sequence
NX	a one-dimensional array of variables NX(I) which stores the checking sequence
NRESET	an integer variable which is set to 0 to simulate the application of master reset to the circuit; otherwise it is set to 1.
MFG	total sets of faults to be simulated
MLTFLT	number of faults in each fault-set

It is assumed that the circuit of Fig.4.16 has a multiple fault (30 s-a-0, 22 s-a-1, 33 s-a-1). Supposing that the circuit is initially at state $15(Q_1Q_2Q_3Q_4=1110)$, the response to the homing sequence at output 10 is recorded in Fig.4.23. Up to this point the circuit has produced the expected response at output 10, as can be found by comparing the response with that of the fault-free circuit (Fig.4.21). Therefore it may be assumed that the circuit actually started in state 15 and terminated in state 6; hence the rest of the test sequence is applied to the circuit and the result is shown in Fig.4.24. It took about 4 secs of computer time to simulate the behaviour of the circuit in presence of the multiple fault; an incorrect response was recorded at output 10 after the 13th clock pulse was applied. The response of the fault-free and faulty circuit to the test sequence at output 11 are shown in Fig.4.25a and b respectively.

4.4.2 A Procedure for Locating Faults in Synchronous Sequential Circuits

The procedure is based on the assumption that the state of the fault-free circuit after the application of each input symbol in the

FAULT SIMULATION OF THE SEQ.CIRCUIT

ELEMENT 30 STUCK-AT 1

ELEMENT 22 STUCK-AT 2

ELEMENT 33 STUCK-AT 2

APPLY INPUT
2 2 2 2 1 1 1 1 2 2 1 1 1 1 1
APPLY CLOCK PULSE 1
2 1 1 1 1 2 2 2 2 1 1 2 1 2 2

APPLY INPUT
2 1 1 1 1 2 2 2 2 1 1 2 1 2 2
APPLY CLOCK PULSE 2
2 2 1 2 2 1 2 1 1 1 2 1 1 1 2

APPLY INPUT
2 2 1 2 2 1 2 1 1 1 2 1 1 1 2
APPLY CLOCK PULSE 3
2 1 1 1 2 2 2 2 1 2 1 1 1 2 1

APPLY INPUT
2 1 1 1 2 2 2 2 1 2 1 1 1 2 1
APPLY CLOCK PULSE 4
2 1 1 2 1 2 2 1 2 2 1 1 1 2 2

APPLY INPUT
2 1 1 2 1 2 2 1 2 2 1 1 1 2 2
APPLY CLOCK PULSE 5
2 1 1 2 2 2 2 1 1 2 1 1 2 1 1

APPLY INPUT
2 1 1 2 2 2 2 1 1 2 1 1 2 1 1
APPLY CLOCK PULSE 6
2 1 2 1 1 2 1 2 2 2 1 1 2 1 2

APPLY INPUT
2 1 2 1 1 2 1 2 2 2 1 1 2 1 2
APPLY CLOCK PULSE 7
2 1 2 1 2 2 1 2 1 2 1 1 2 2 1

APPLY INPUT
2 1 2 1 2 2 1 2 1 2 1 1 2 2 1
APPLY CLOCK PULSE 8
2 1 2 2 1 2 1 1 2 2 1 1 2 2 2

I/P 2 2 2 2 2 2 2 2

O/P 2 1 1 2 2 2 2 2
BEFORE CLOCK PULSE

O/P 1 1 2 2 2 2 2 2
AFTER CLOCK PULSE

Fig 4.23 Response to the homing sequence

FAULT SIMULATION OF THE SEQ.CIRCUIT

ELEMENT 30 STUCK-AT 1

ELEMENT 22 STUCK-AT 2

ELEMENT 33 STUCK-AT 2

I/P 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 1 2 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 2

O/P 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2
2 2 2
BEFORE CLOCK PULSE

O/P 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2
2 2 2
AFTER CLOCK PULSE

Fig 4.24 Response to the test sequence

I/P 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 1 2 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 2

O/P 1 2 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 2 2 2 2 1 1 1 1 1 1 1
1 1 1
BEFORE CLOCK PULSE

O/P 2 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 2 2 2 2 1 1 1 1 1 1 1 1
1 1 2
AFTER CLOCK PULSE

a. Expected response at output 11

FAULT SIMULATION OF THE SEQ.CIRCUIT

ELEMENT 30 STUCK-AT 1

ELEMENT 22 STUCK-AT 2

ELEMENT 33 STUCK-AT 2

I/P 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 1 2 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1
1 1 2

O/P 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1
1 1 1
BEFORE CLOCK PULSE

O/P 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1
1 1 1
AFTER CLOCK PULSE

b. Actual response

Fig 4.25

checking sequence is known; this may be determined by computer simulation of the circuit state-transition with each input symbol. The procedure starts as soon as the output of the circuit under test differs from that of the fault-free circuit and consists of the following steps:-

(1) Determine the state of the circuit immediately before the fault was detected i.e. if a fault is detected after the application of the Nth symbol in the checking sequence, then determine the state of the circuit when the (N-1)th symbol was applied. If the fault is detected after the application of the Nth clock-pulse, then the state immediately before the Nth clock pulse application is the pre-fault state.

(2) Find the state to which the circuit would move if there were no fault i.e. the state after the application of the (N-1)th input or Nth clock pulse, depending on when the fault was detected, and the input symbol applied at that state. The state of the circuit at this stage together with the input symbol form the test value which checks the combinational part of the circuit.

(3) Record the correct logical states of the elements in the combinational part of the fault-free circuit when the test value is applied in step 2.

(4) Find out those elements in step 3, which if they are at incorrect logical states will give a false output for the test value. Since a fault in the feedback circuit will be propagated to the pseudo-primary inputs, it will be assumed that any stuck-at fault at pseudo-primary inputs is either due to the output stuck-at fault in the corresponding flip-

flop or the input circuit associated with that flip-flop.

(5) Further diagnostic information about the fault in the feed-back circuit may be obtained by the following procedures which are shown in Table 4.8 for D and JK flip-flop. For example, if the input to a JK flip-flop is such that the output of the flip-flop is to go to 1 from 0, but instead remains at 0, then either the flip-flop output itself or its J input is stuck-at-0.

4.4.3 A circuit example

The circuit of Fig.4.12 will be considered and it is assumed that the circuit has a multiple fault (16 s-a-1, 14 s-a-1). The response of the circuit to the homing sequence is shown below;

Homing sequence	1 0	
Output	0 0	before clocking
	0 0	after clocking

The output response resulting from the applications of the homing sequence is different to what is expected if the circuit was initially in any of the four states, as derived in section 4.3.3; hence a fault has been detected. Since the response produced is close to what would have resulted if the circuit were initially in state 3 or state 4, it will be assumed that the homing sequence has transferred the circuit to state 3; the corresponding test sequence is

1 1 0 0 1 0 0

The expected and the actual output are as follows;

Test sequence	1 1 0 0 1 0 0	
Expected output	0 0 1 0 1 0 0	before clocking
	0 0 0 0 0 0 0	after clocking

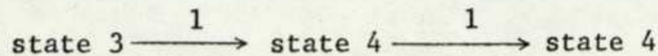
Flip-flop type	Present State	Next State		Fault
		Desired	Actual	
D	0	1	0	D stuck-at-0
	1	0	1	D stuck-at-1
JK	0	1	0	J stuck-at-0
	0	0	1	J stuck-at-1
	1	0	1	K stuck-at-0
	1	1	0	K stuck-at-1

Table 4.8 Fault location at flip-flop input

Actual output   0 0 0 0 0 0 0 before clocking

0 0 0 0 0 0 0 after clocking

By comparing the expected and the actual output, it may be found that an incorrect response is given by the third input symbol of the test sequence. If the circuit was in state 3 after the application of the homing sequence, it would be in state 4 after the application of the second input symbol of the test sequence



if there were no fault in the circuit. Since a fault was detected when the third input symbol was applied, it may be assumed that either the circuit did not remain at state 4 after the second input symbol was applied, i.e. a fault from the feedback part of the circuit was propagated to the flip-flop output, or a fault in the combinational part affected the output response. The fault-locating algorithm may now be applied

(1) The state immediately before the detection of the fault is 4, i.e. the state when the second input was applied.

(2) The circuit should remain at state 4 after the application of the second input i.e. when the third input (0) is applied.

(3) The test value to the combinational part of the circuit at this stage is 01001.

(4) The correct logical states of the elements in the combinational part, for 01001, would be

Element	2	3	4	5	11	16	17	18
Logic state	1	0	0	1	1	1	0	1

If the test fails, then the suspect failures would be

Element	2	4	3	5	11	17	18
Stuck-at	0	1	1	0	0	1	0

which forms the equivalent class of faults for 01001.

(5) It was mentioned before that if the circuit is fault-free it should remain at state 4 (Q_1Q_2=10) after the application of the second input, 1. If the pseudo-primary input failures of step 4 do occur then the circuit would have moved either to state 1 (Q_1Q_2=00) when 2 s-a-0 or to state 3(Q_1Q_2=11) when 3 s-a-1. On this basis, the following result may be obtained:

Expected state	Actual state	Fault
4(Q_1Q_2=10)	1(Q_1Q_2=00)	13 s-a-1
4(Q_1Q_2=10)	3(Q_1Q_2=11)	14 s-a-1

After correcting the fault the homing sequence is applied again and the response is

0 1 before clocking

0 0 after clocking

hence the test sequence would be (11001). The expected and the actual output responses are shown below:

Test sequence	1	1	0	0	1	00	
Expected output	0	0	1	0	1	00	before clocking
	0	0	0	0	0	00	after clocking
Actual output	0	0	1	0	0	00	before clocking
	0	0	0	0	0	00	after clocking

(1) The state immediately before the detection of the fault is state 3 i.e. the state when the 4th input symbol was applied. The state transitions for the fault-free circuit is

state 3 $\xrightarrow{1}$ state 4 $\xrightarrow{1}$ state 4 $\xrightarrow{0}$ state 3 $\xrightarrow{0}$ state 2 $\xrightarrow{1}$ state 3

(2) The circuit should be in state 2 when the 5th input symbol, 1, is applied.

(3) The test value for the combinational part of the circuit is 10110.

(4) The correct logical states are

Element	2	3	4	5	11	16	17	18
Logic state	0	1	1	0	0	0	1	1

The indistinguishable fault set for test 10110 is

Element	2	4	3	5	16	18
Stuck-at	1	0	0	1	1	1

4.4.4 Computer simulation of the fault-locating procedure with an example

The fault-locating procedure described in the previous section has been computer-programmed so that together with the automatic checking sequence generation programme, it may satisfy the software requirements for the diagnostic testing of synchronous sequential circuits. The flow-diagram of the programme LOCATION is shown in Fig.4.26a; the subroutine FEEDBACK (flow-diagram in Fig.4.26b) is used to locate fault associated with the flip-flop input circuit. The arrays and variables used in the programme have the following definitions and uses

N	total number of elements in the circuit
NPRIM,NOUT	number of primary inputs and outputs in the circuit
NFBK	number of pseudo-primary inputs in the circuit
NFF	number of flip-flops used
NTYPE	type of flip-flop used
MSTATE	a two-dimensional array of variables MSTATE(I,J) storing the ith state of the circuit in the jth position

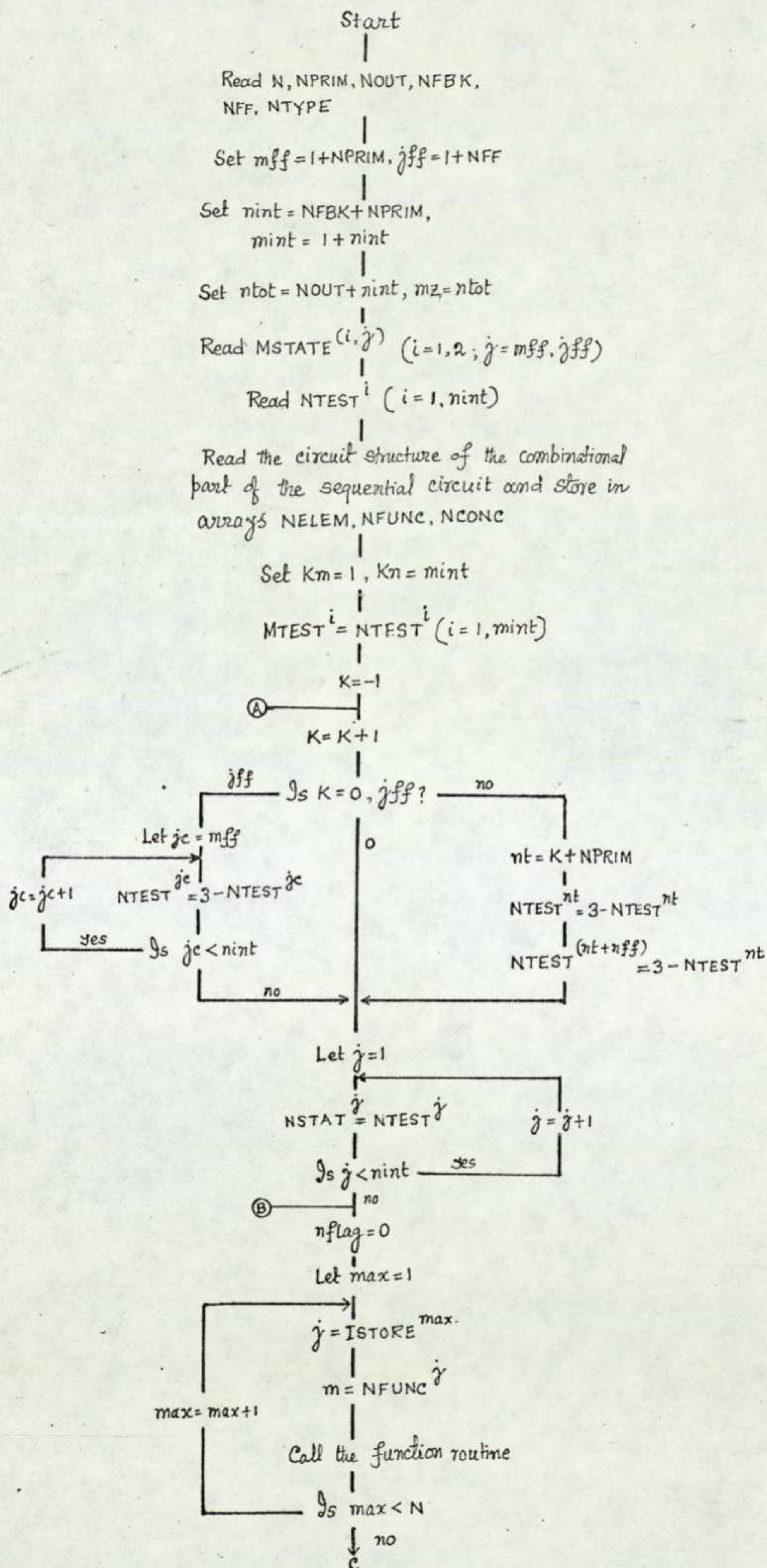
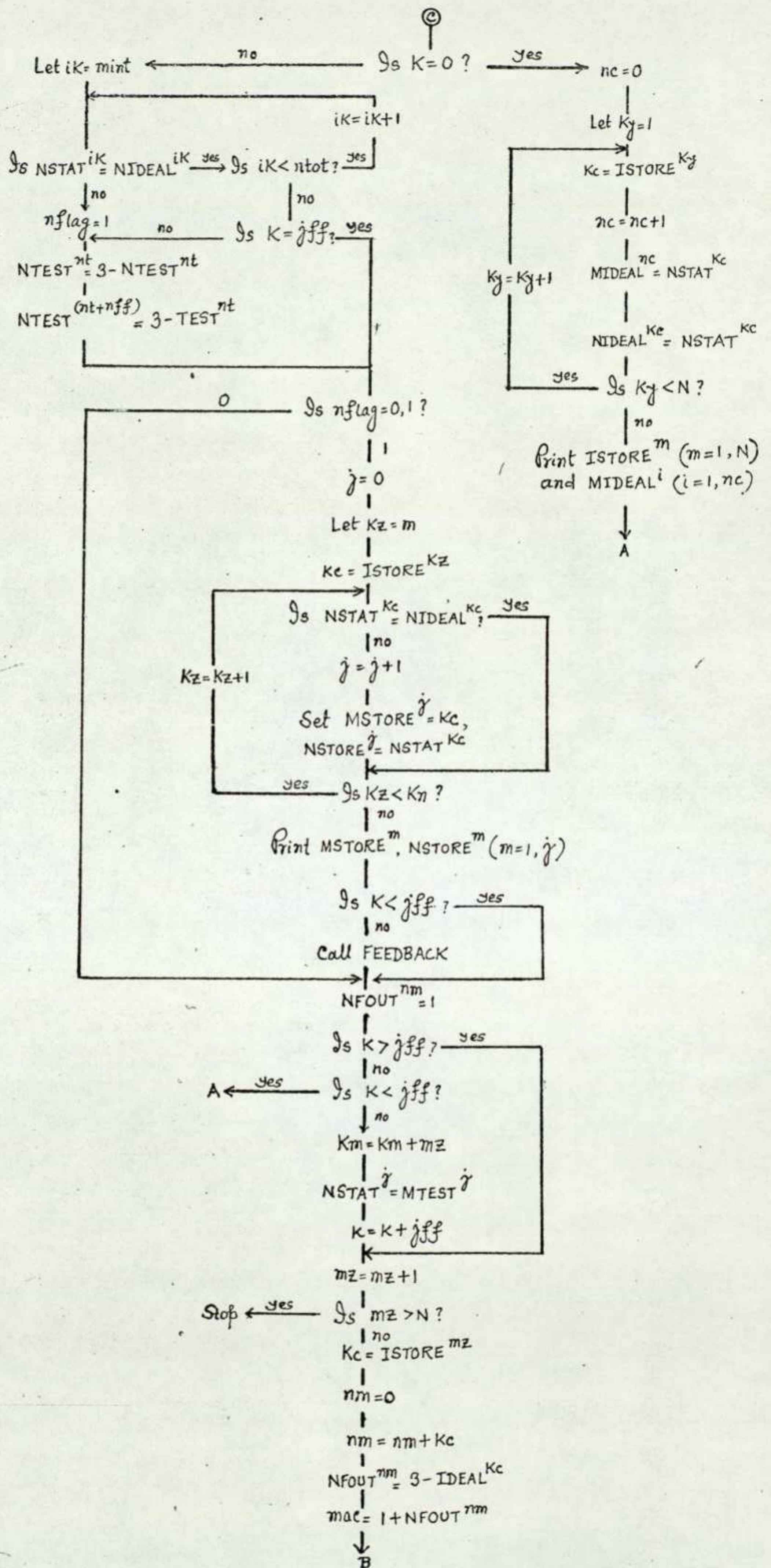


Fig 4.26a Flow-diagram of LOCATION

cont'd



C
Is K=0 ?
no
yes
Let iK= mint
nc=0
iK=iK+1
Let Ky=1
Is NSTAT^iK = NIDEAL^iK
yes
Is iK< ntot?
yes
Kc = ISTORE^Ky
no
no
nflag=1
no
Is K=jff?
yes
nc=nc+1
NTEST^nt = 3-NTEST^nt
Ky=Ky+1
MIDEAL^nc = NSTAT^Kc
NTEST^(nt+nff) = 3-TEST^nt
NIDEAL^Kc = NSTAT^Kc
yes
Is Ky<N ?
0
Is nflag=0,1 ?
no
1
Print ISTORE^m (m=1, N)
and MIDEAL^i (i=1, nc)
j=0
A
Let Kz=m
Kc = ISTORE^Kz
Is NSTAT^Kc = NIDEAL^Kc ?
yes
no
j=j+1
Kz=Kz+1
Set MSTORE^j = Kc,
NSTORE^j = NSTAT^Kc
yes
Is Kz<Kn ?
no
Print MSTORE^m, NSTORE^m (m=1, j)
Is K<jff ?
yes
no
Call FEEDBACK
NFOUT^nm =1
Is K>jff ?
yes
no
A
yes
Is K<jff ?
no
Km=Km+mz
NSTAT^j = MTEST^j
K=K+jff
mz=mz+1
Stop
yes
Is mz>N ?
no
Kc = ISTORE^mz
nm=0
nm=nm+Kc
NFOUT^nm = 3-IDEAL^Kc
mac= 1+NFOUT^nm
B

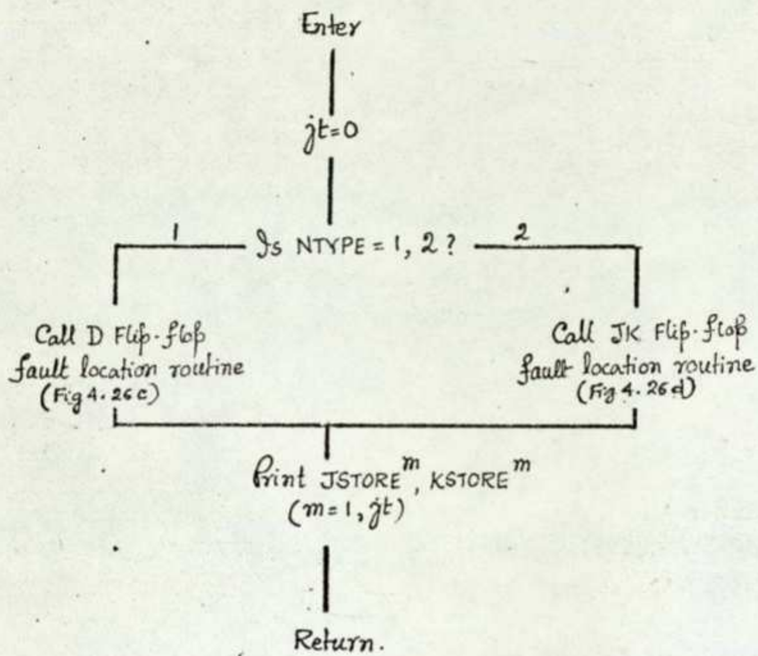
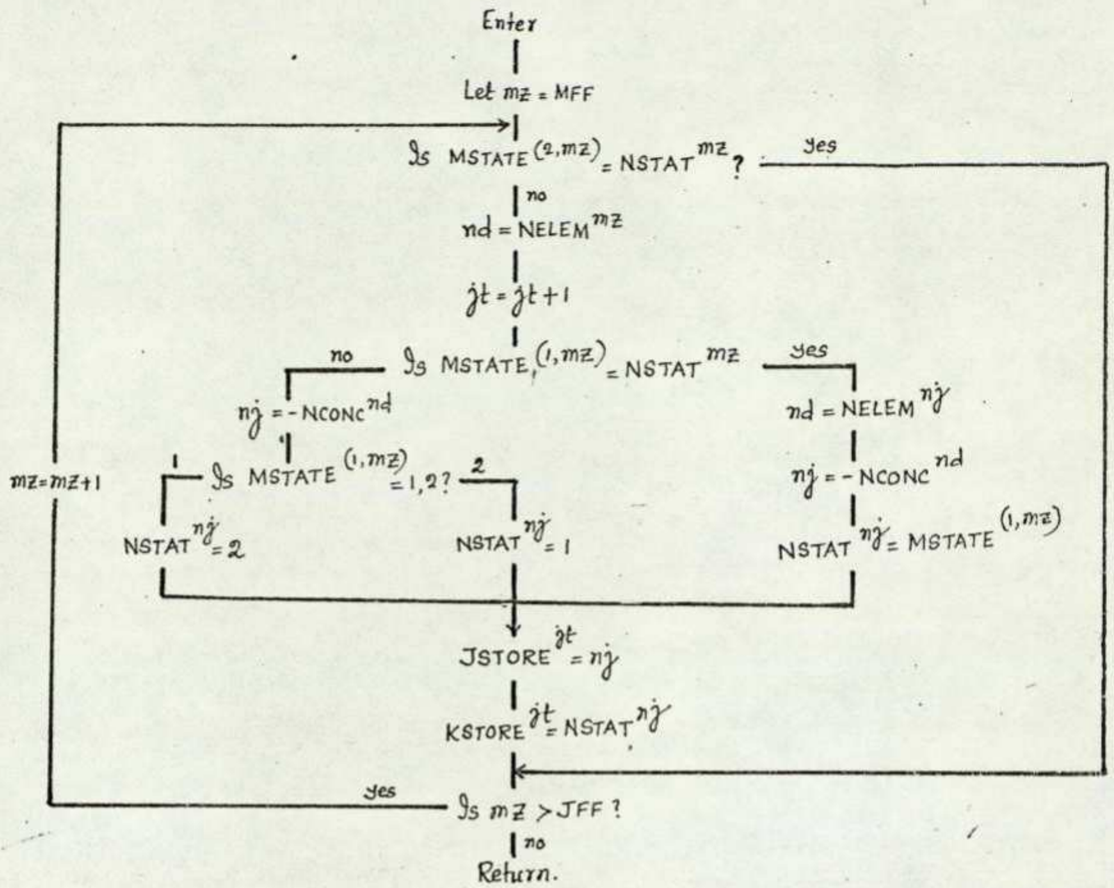
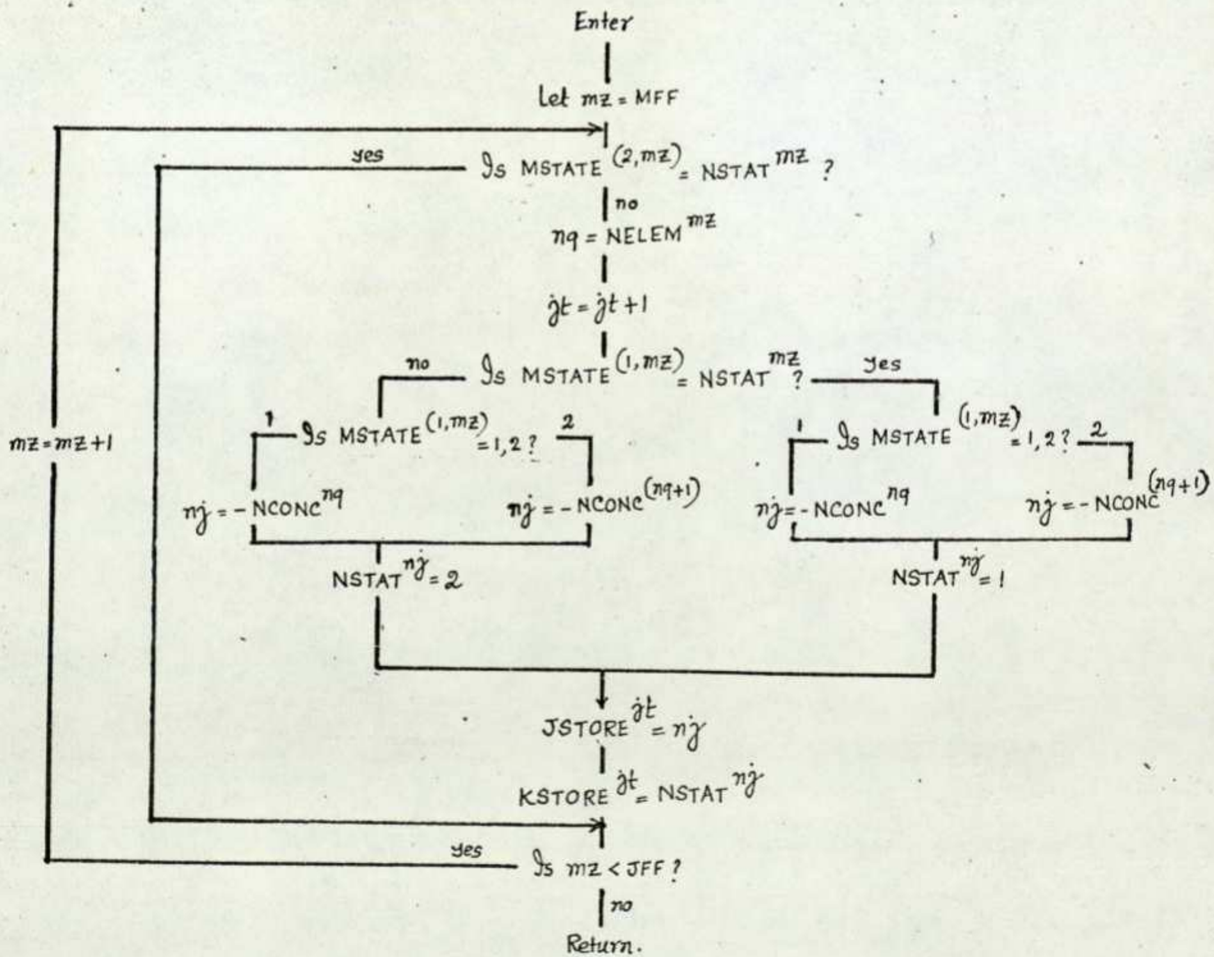


Fig 4.26b Flow-diagram of FEEDBACK



c. D Flip-flop



d. JK Flip-flop

Fig 4.26 c&d. Flow-diagrams of flip-flop fault location routines.

NTEST	a one-dimensional array of variables NTEST(I) in which the test input to the circuit is stored
ISTORE	a one-dimensional array of variables ISTORE(I) in which are stored the element numbers of the combinational part of the circuit
MTEST	a one-dimensional array having the same function as NTEST
NIDEAL	a one-dimensional array of variables NIDEAL(I) in which the logical states of the elements of array ISTORE are stored
MIDEAL	a one-dimensional array having the same function as NIDEAL
MSTORE	a one-dimensional array of variables MSTORE(I) in which the faults located by the applied test are stored
NFOUT	a one-dimensional array whose function has been described in section 2.4.2.
NSTORE	a one-dimensional array of variables NSTORE(I) in which the logical states of the elements of MSTORE are stored.

To show the application of the algorithm it is assumed that the circuit of Fig.4.16 has a fault (13 s-a-1) which is to be detected and located. The fault is detected by applying the appropriate test sequence as recorded in Fig.4.27. The programme LOCATION is then supplied with the proper data input (Fig.4.28); the output result is shown in Fig.4.29. As it may be seen, 13 s-a-1 is included among the indistinguishable fault set; the computer time taken to generate the indistinguishable fault set of Fig.4.29 was about 12 secs.

FAULT SIMULATION OF THE SEQ.CIRCUIT

ELEMENT 13 STUCK-AT 2

```
I/P 2 2 2 2 2 2 2 2 1 1 1 1 1 2 2 1 2 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
2

O/P 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2
BEFORE CLOCK PULSE

O/P 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2
 AFTER CLOCK PULSE
```

Fig 4.27 Output response

```
27 1 2 8 4 1
2 1 1 1
2 1 1 2
1 2 1 1 2 1 2 2 1
4 9 6 8 7 6 8 8 7 3 3 5 6 6 6 7 7 8 6 6 7 8 8 8 5 7 6
1 1 0 16
2 1 12 0 21 22 25 27 28
3 1 13 0 19 21
4 1 14 0 18 21 25 28
5 1 15 0 17 25 27
6 1 12 0 18 26
7 1 13 0 22 26 27 28
8 1 14 0 19 22 26 27
9 1 15 0 22 26 28
10 2 24
11 2 31
16 7 1 0 39
17 5 5 6 0 20
18 5 4 6 0 20
19 5 3 8 0 20
20 5 17 18 19 0 24
21 5 2 3 4 0 23
22 5 2 7 8 9 0 23
23 5 21 22 0 24
24 4 20 23 0 10
25 5 2 4 5 0 29
26 5 6 7 8 9 0 30
27 5 2 5 7 8 0 30
28 5 2 4 7 9 0 30
29 7 25 0 31
30 5 26 27 28 0 31
31 4 29 30 0 11
```

Fig 4.28 Data input for program LOCATION

```
STATE 1 = 2 1 1 1
STATE 2 = 2 1 1 2

                    CORRECT LOGICAL STATES

 1  2  3  4  5  6  7  8  9 10 11 16 17 18 19 20 21 22 23 24 25 26 27 28

29 30 31
 1  2  1  1  2  1  2  2  1  1  2  2  2  2  2  1  2  2  1  1  2  2  1  2

 1  2  2

    SUSPECT FAILURES

      2  6 10
      1  2  2
AND FEEDBACK PATH ASSOCIATED WITH 12

          12
           1
IN PREVIOUS TEST

      3  7 10
      2  1  2
AND FEEDBACK PATH ASSOCIATED WITH 13

          13 <------ Fault located
           2
IN PREVIOUS TEST

      5  9 10
      1  2  2
AND FEEDBACK PATH ASSOCIATED WITH 15

          15
           1
IN PREVIOUS TEST

      2  3  4  5  6  7  8  9 10
      1  2  2  1  2  1  1  2  2
AND FEEDBACK PATH ASSOCIATED WITH 12 13 14 15

          12 13 14 15
           1  2  2  1
IN PREVIOUS TEST

     17 20 24
      1  2  2

     18 20 24
      1  2  2

     19 20 24
      1  2  2

     20 24
      2  2

     21 23 24
      1  2  2

     22 23 24
      1  2  2

     23 24
      2  2

     24
      2

     27 30 31
      2  1  1

     30 31
      1  1

     31
      1
```

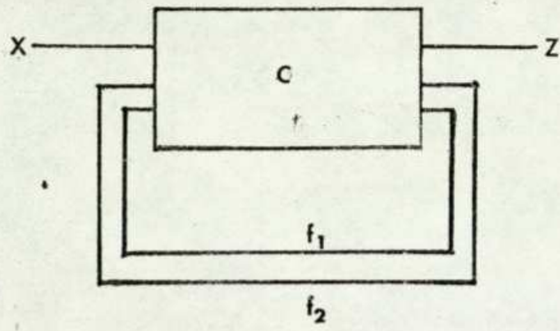
Fig 4.29 Output of program LOCATION

4.5 Fault-folding Techniques In Asynchronous Sequential Circuit Testing

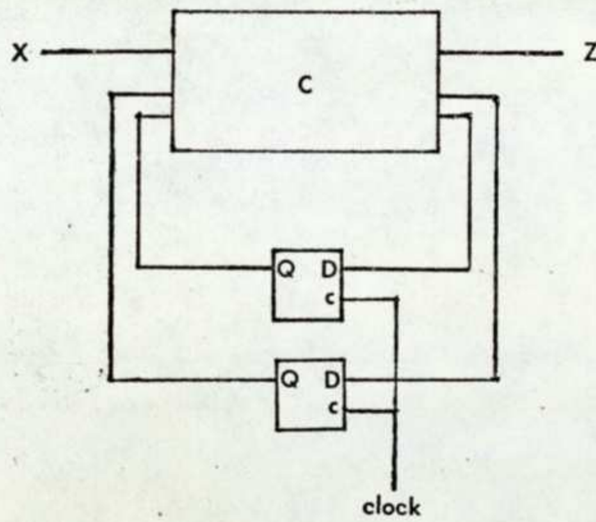
4.5.1 Test generation

Applications of fault-folding methods to combinational and synchronous sequential circuits have already been treated. In this section the application of the method to detect faults in asynchronous sequential circuits will be demonstrated. A synchronous model of a given asynchronous circuit is made up and the test sequences are generated for this synchronized version. However when this test sequence is applied to the asynchronous circuit, it might be invalid because of the races or hazards created by faults. Hence the test sequence generated by this procedure has to be verified by processing them through a three valued simulator which carries out race and hazard analysis according to the theory developed by Eichelberger (97); those tests which produced static hazards or races are to be discarded. For simplicity it will be assumed here that faults do not introduce races or hazards in the circuit and the test sequence obtained for the synchronised version can be directly used for testing the real circuit.

A combinational circuit may be obtained from a asynchronous sequential circuit by cutting a selected number of feedback lines. A procedure for selecting the feedback lines to be cut has been described by Putzolu et al (76); its objective is to make the cuts where the insertion of delays will not alter the logical behaviour of the circuit. After cutting the feedback lines of the asynchronous circuit (Fig.4.30a) a synchronous version is constructed by inserting D flip-flops in these feedback lines as shown in Fig.4.30b; this synchronous version of an asynchronous circuit can now be handled like a normal synchronous circuit for test generation.



a. Asynchronous circuit S



b. Synchronous version S*

Fig 4.30

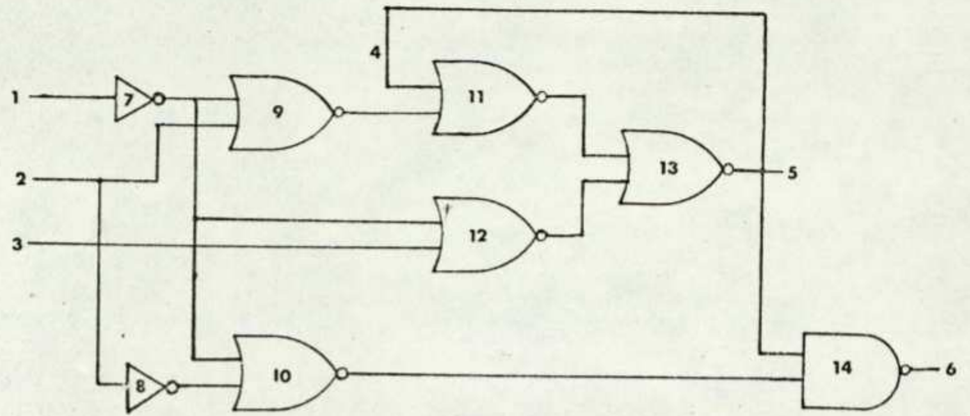
4.5.2 A circuit example

A simple asynchronous circuit is shown in Fig.31a (82). The feedback line from element 13 to 11 can be cut by applying the loop cutting procedure of (76); the synchronous version of the circuit, shown in Fig.4.31b, is obtained by inserting a D flip-flop in the cut feedback line of the asynchronous circuit. The state/output table describing the sequential network is given by Table 4.9; stable-states have been ringed in the table. An internal state is stable if the next state vector Y equals the present state vector y.

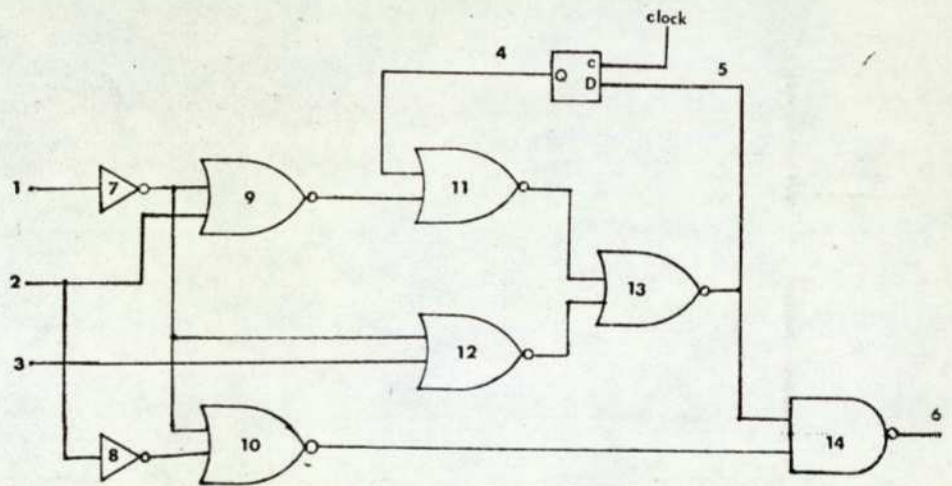
It will be assumed that the test procedure satisfies the fundamental mode restrictions i.e. the input cannot change until the internal state variables become stable. With reference to the synchronized version of the circuit it will mean application of an input, recording the output and then clocking the D flip-flop before the next input is applied. An additional constraint will also be imposed so that each input value in the test pattern differs only in one value; it helps to minimize problems due to races and hazards (94). Ashkinazy (98) has shown that for any asynchronous circuit S, there exists a synchronous circuit S' such that checking sequence for S' is also a checking sequence for S under the single input change assumption.

The combinational version of the circuit of Fig.4.31b is shown in Fig.4.32. By applying the test generating algorithm of section 3.5 the minimal test set which detects all detectable faults in the combinational circuit are recorded in Table 4.10. Hence the synchronous circuit has to be checked at state 1(D=0) with input value 111 and at state 2(D=1) with input values 110,111,011 and 101. It may be found from the state table that input value 111 is a homing sequence for the synchronous circuit.

Assuming the circuit is initially in state 1, the input pattern



a. Asynchronous circuit example



b. Synchronous form of (a)

Fig 4.31

PS \ $x_1x_2x_3$ (NS, z)	000	001	101	100	110	111	011	010
1	(1),1	(1),1	2,1	(1),1	(1),1	(1),1	(1),1	(1),1
2	(2),1	(2),1	(2),1	1,1	1,1	(2),0	(2),1	(2),1

Tabie 4.9 State-output table

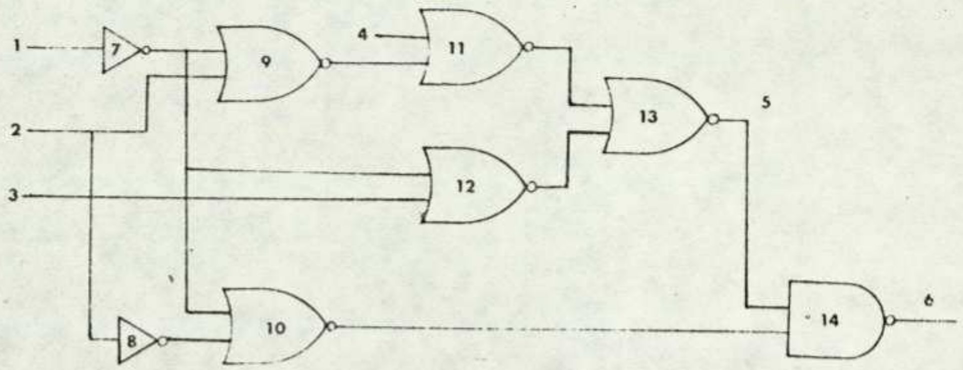


Fig 4.32 Combinational form of the circuit of Fig 4.31b

Test	Indistinguishable Fault Set
1 1 1 1	7 s-a-1,10 s-a-0,8 s-a-1,14 s-a-1, 13 s-a-0,11 s-a-1,12 s-a-1
1 1 0 1	12 s-a-0,13 s-a-1,14 s-a-0
1 1 1 0	9 s-a-1,11 s-a-0,13 s-a-1,14 s-a-0
0 1 1 1	7 s-a-0,10 s-a-1,14 s-a-0
1 0 1 1	8 s-a-0,10 s-a-1,14 s-a-0

Table 4.10 Test set

to test the circuit (see section 4.3.2) is

111 . 111 . 101 . 111 . 011 . 001 . 011 . 111 . 110

and the corresponding output pattern is

1 1 1 0 1 1 1 1 1.

Since there is no change in the first two input values of the test pattern, they may be combined to one input value, hence the test input and output pattern becomes

$$\left.\begin{array}{cccccccc} 111 & . 101 & . 111 & . 011 & . 001 & . 011 & . 111 & . 110 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{array}\right\} \quad \ldots . (A)$$

Similarly if the circuit is initially at state 2, the test input pattern and the resulting output pattern would be

$$\left.\begin{array}{cccccc} 111 & . 110 & . 111 & . 011 & . 001 & . 101 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{array}\right\} \quad \ldots . (B)$$

Depending on whether the output response to 111 is 1 or 0, test pattern A or B is selected.

A. Detection of a Single Fault

It is assumed that the circuit of Fig.4.31a has a fault, 7 s-a-1. The response to the homing sequence 111 is found to be 1, the corresponding input pattern (A) is applied.

Input 111 . 101 . 111

Output 1 . 1 . 1

A fault is detected by the input 111. In a fault-free circuit the state of the circuit when the input value 111 of the test pattern (A) is applied, would have been 2 i.e. test value at pseudo-input 4 would be 1. From Table 4.10, it may be found that 7 s-a-1 is an element of the

fault-set for test 1111.

B. Detection of Multiple Faults

Assuming the circuit has a multiple fault 9 s-a-1 and 12 s-a-0, the homing sequence is applied; the response is 0 hence test input pattern (B) is applied.

Input	111 . 110
Output	0 . <u>0</u>

Input value 110 gives an incorrect response. If there were no failures, the circuit state when 110 is applied would be 2 i.e. test value at pseudo-input is 1. Again from Table 4.10 it may be found that 12 s-a-0 is included in the indistinguishable class of faults for test 1101. After removing the fault the homing sequence is applied again, the response is 0 therefore input pattern B is re-applied.

Input	110 . 110 . 111
Output	0 . 1 . <u>0</u>

The fault can be localised as before.

5. INTERMITTENT FAULT DETECTION IN LOGIC CIRCUITS

Intermittent failures are the most frequently occurring faults in data processing systems once the installation period is terminated. To detect a permanent fault any particular test need only be applied once. However a fault, which is intermittent in nature, may escape detection when the test is applied. A detection procedure, based on repeated application of tests that test for a permanent fault in the circuit has been presented here, to trap intermittent faults. Only 'well-behaved' intermittent stores will be considered i.e. either the circuit behaves as if it is fault-free or seems to possess a solid fault during the duration of a test (34). The time-period, during which a test or a test sequence is repeated, is selected on the basis of the probability of detection desired and is derived from the familiar Poisson distribution of statistics.

5.1 A Probabilistic Model of Intermittent Fault Occurrence

5.1.1 Intermittent fault model

A method of modelling intermittent faults, based on classical probability theory has been suggested by Parker and McCluskey (36). If p_f is the probability of a fault occurring in the circuit, then $p_f=0$ means that the fault does not exist whereas $p_f=1$ indicates that it is present permanently (solid fault) in the circuit. When p_f is set between the two extremes, $0<p_f<1$, the fault can be assumed to be intermittent because it appears randomly in time. This model will be used for intermittent fault simulation in this thesis. The model works

on the assumption that an a-priori probability value may be assigned to the occurrence of a fault. Kamal and Page (34) have indicated that these values can be estimated empirically based on the familiarity with the circuit.

An example is quoted from (34) for illustration.

> "Among the gates produced by a certain manufacturer, it is estimated that for about 0.01 percent of them, the gap between ON and OFF voltages is smaller than some critical value. If the gap is below the critical value, the gate will malfunction 5 percent of the time. ......"
>
> The probability of failure in this case is assumed to be 0.005.

By definition, intermittency is a time-dependent phenomenon and can be represented by a random variable whose distribution depends on the components and the environment. Although an intermittent fault may, at least theoretically, be detected by repeating a test which would detect the fault if it were solid (34), the number of repetitions themselves would not give any significant information because by varying the rate, the same number of tests may be cycled in shorter or longer periods of time. The main problem in intermittent fault detection is to make sure that a test is applied to the circuit when the fault is activated (99). Given the probability of failure, the problem becomes that of determining the length of time in which the fault has a high probability of occurrence. Therefore if the test is repeated for that length of time (say T), the probability of detecting the fault will also be high. If ΔT is the duration of the test, then it has to be cycled $N(= T/\Delta T)$ times to detect the fault. Hence if only N, the number of tests to be applied is quoted, they must be applied over a time $N \times \Delta T$ with the assumption that a fault when it appears, exists for time ΔT.

5.1.2 Simulation of intermittent faults

The fault model described in section 5.1.1 has been used to simulate an intermittent fault which is assumed to occur with a constant probability. A random number generator is used to generate a number x which lies between 0 and 1. If $x \leq p_f$, the probability of failure, then the desired intermittent fault is set up as a solid fault for the test concerned and at the end of the test, it is removed; the fault set-up and removal procedures have been described in section 2.4. In a similar way multiple intermittent faults can also be simulated provided they all have identical probabilities of failures and appear simultaneously. In general this requirement for multiple intermittent faults is very restrictive hence only single intermittent faults will be considered in this thesis.

A method of generating pseudo-random numbers has been described by Rotenberg (100); this has been implemented as a library function, called RAND, at The City University Computer Centre. It generates random numbers between 0 and 1 with uniform distribution. RAND is used by initially defining a seed number Y of value $0 \leq Y < 1$. Thereafter a random number is generated by calling the function Y = RAND(Y). Y is also the seed for the next random number to be generated. Different initial seeds generate different sequences of random numbers and conversely the same sequence of (pseudo-) random numbers is generated by the same initial seeds.

The flow diagram of the computer-programmed implementation of the intermittent fault model is shown in Fig.5.1; the named variables have the following definitions:-

PROB an input variable specifying probability of fault occurrence

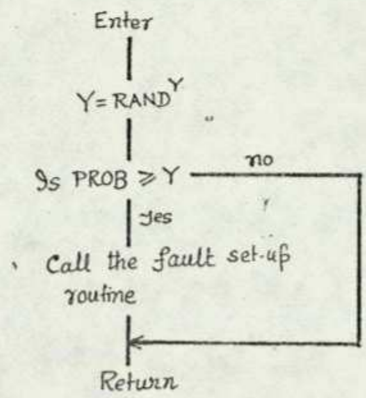


Fig 5.1 Intermittent fault simulation

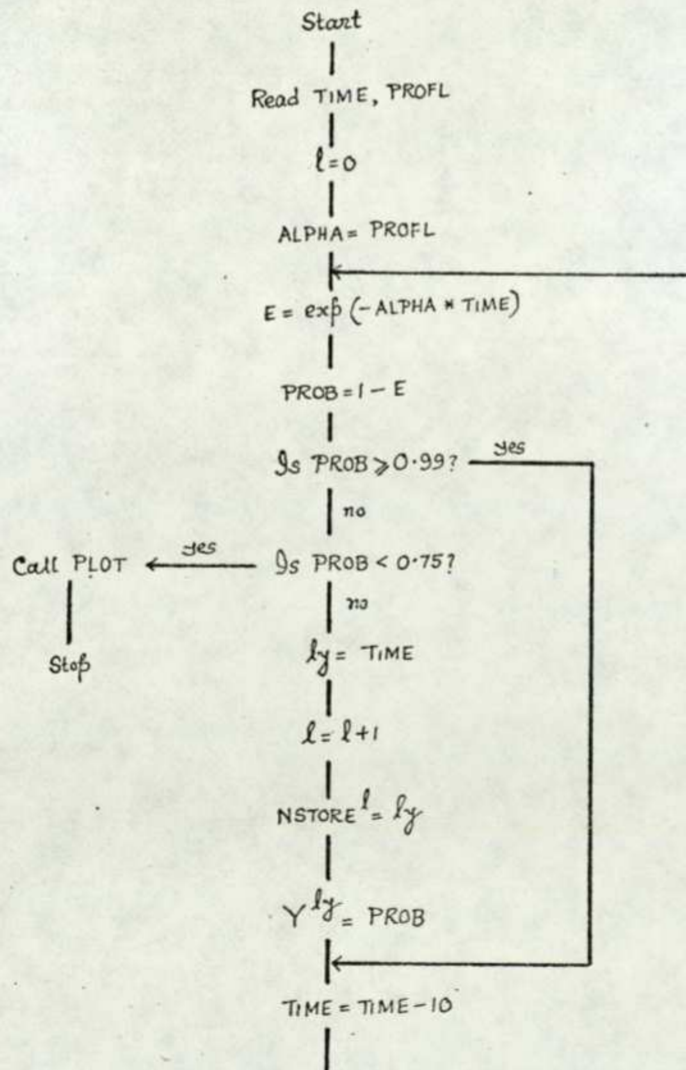


Fig 5.2 Flow-diagram of POSS.

Y A real variable holding the random number generated by RAND. Also this is the initial seed to start the sequence of random numbers

5.2 Poisson Distribution in Intermittent Fault Detection

5.2.1 The Poisson distribution

Consider a random variable x as the number of successes in n independent trials of a two-outcome experiment; this variable assumes the values 0,1,2,....,n. If the probability of success of each event is p, then the probability of x successes in n independent trials is given by

$$P(x) = {}^{n}C_{x}\, p^{x}\, (1-p)^{n-x}.$$

This above expression for P(x) enables one to assign a probability to each value of x from 0 to n . The resulting distribution of probabilities is known as the "Binomial distribution". The mean of the distribution μ is equal to np; in other words np is the most likely number of success.

If in the Binomial distribution $p \ll 1$ but n is sufficiently large for the mean, np, to be significant, the probability function P (x) may be shown to take the approximate form

$$P(x) = \frac{\mu^{x} e^{-\mu}}{x!} \quad (x = 0,1,2,\ldots n)$$

where $\mu = np$ is the mean of the distribution

A distribution given by the above equation is called the "Poisson distribution". The Poisson distribution strictly speaking is not a continuous one, but gives probabilities for any particular whole number 0,1,2.... of successes when the mean is μ. The factors μx/x! in the successive probabilities are the successive terms in the expansion of

$e^{+\mu}$, so that the sum of all the probabilities is $e^{+\mu}e^{-\mu}=1$, as expected.

The Poisson distribution applies in situations where events are distributed throughout a relatively large region (time or space) and the following conditions are met (101)(101A):-

(i) the events occur randomly within the region.

(ii) the probability of events occurring in one sub-region is not affected by the probability of events occurring in another sub-region.

(iii) the probability of occurrence of two or more events in a very small sub-region is negligible.

The Poisson distribution has two main applications; firstly as an useful approximation to the Binomial distribution when the binomial parameter p is small, as explained in this section, and secondly for describing the number of 'events' which occur randomly in certain time intervals. The latter situation will be described in Appendix A.

5.2.2 Detection of intermittent faults

Let it be assumed that a fault occurs randomly in a circuit and the probability of its appearance is p_f. If a test for the fault is repeated for a sufficiently long time T, the fault will appear $p_f T(=\mu)$ times, the Poisson parameter. But one occurrence of the fault is enough for the detection purpose, hence the problem is to find the time $t(<T)$ in which the fault will appear at least once.

The Poisson distribution for the situation is

$$P(x) = \frac{e^{-p_f T}(p_f T)^x}{x!} \quad (x=0,1,2....).$$

The probability for getting at least 1 fault is,

$$P(x \geqslant 1) = 1 - P(x=0)$$
$$= 1 - e^{-p_f T} \qquad .... (A)$$

$P(x \geqslant 1)$ will be termed as the probability of detecting an intermittent fault. In the expression (A) if, for example, $p_f = 0.04$ and $T = 150$ then

$$P(x \geqslant 1) = 1 - .00248 = 0.99752 \ (\approx 1).$$

The problem may be stated in a different way: if the probability of failure is known, how long does the test has to be repeated to get a desired probability of detection. In fact the solution to this is of main interest in this thesis. The longer the test is applied, the higher is the probability of detecting an intermittent fault. Consequently if the length of the test time is reduced there will be a corresponding decrease in the probability of detection and vice-versa.

For example, if the same intermittent fault is to be detected i.e. having $p_f = 0.04$, but this time the probability of detection desired is approximately 80 per cent, then the appropriate test has to be cycled for 40 units of time only. If the duration of a test is chosen as the unit of test time, then the test has to be repeated 40 times to detect the intermittent fault.

5.2.3 Programme developement for the evaluation of test-time from a given probability of failure

The probability of detecting an intermittent fault varies with test time. A programme has been developed which plots this variation when the probability of fault occurrence is known. In other words, given the value of p_f in equation A of section 5.2.2, it plots $P(x \geqslant 1)$ with different values of time T. Thus, depending on the probability of detection desired, the test time can be directly obtained from the graph. The procedure on which the programme is based consists of the following steps:

(1) Select a value for T which makes $e^{-p_f T}$ (equation A, section

5.2.2) negligibly small so that $P(x \geqslant 1)$ becomes almost equal to 1.

(2) Decrease the value of T by $\tau(=1,2,3....T)$ and calculate $P(x \geqslant 1)$.

(3) If the value of $P(x \geqslant 1)$ lies between two pre-set limits, then go to step 4; otherwise stop.

(4) Plot $P(x \geqslant 1)$ against T; go back to step 2.

The flow diagram of the programme POSS is shown in Fig.5.2. It has been assumed that the two pre-set limits (step 3) are 0.75 and 0.99, and τ (step 2) is equal to 10. The arrays and variables in the program have the following definition and uses

PROFL	an input variable specifying the value of the probability of failure
ALPHA	a real variable which takes on the value of the probability of failure
TIME	an input variable specifying the arbitrary time value (step 1)
E	a real variable which holds the value of the exponential term of equation A in section 5.2.2.
PROB	a real variable which takes on the calculated value of the probability of detection
NSTORE	a one-dimensional array of variables NSTORE(I) in which are stored the values of the abcissa co-ordinates (time) of each of the points that it is desired to plot
Y	a one-dimensional array of variables Y(I) in which are stored the values of the ordinate co-ordinates (probability of detection) of each of the points that it is desired to plot.

Subroutine PLOT is designed to plot the arrays of data stored in NSTORE and Y; the flow-diagram of PLOT is shown in Fig.5.3. Definitions and explanations of the significant variables and arrays are given below:

LINE	a one-dimensional array of 65 variables LINE(I) which is set to desired values, then used to print a given line of the plot
NF	an integer variable which takes on the number of points that it is desired to plot
DY	a one-dimensional array of variables DY(I) in which are stored the values of the ordinate scale
TEMP	an input variable specifying the starting value of the ordinate scale
N	an index used to determine which line of the plot is currently being computed

Subroutine PLOT prints the values of the variables and, in addition, every fifth line it will print an ordinate co-ordinate line and the value of N. The process will continue until the index N reaches a value of NF, at which point control will be returned to the main program.

Fig.5.4a and b shows the output of the programme POSS for probability of failure values of 0.04 and 0.06 respectively.

5.2.4 Example of the application of the Poisson distribution in intermittent fault detection

It has been mentioned before that for detection purposes the fault has to appear at least once during the test-time and the test-time is selected according to the probability of detection desired. For example, if an intermittent fault occurs with a probability of

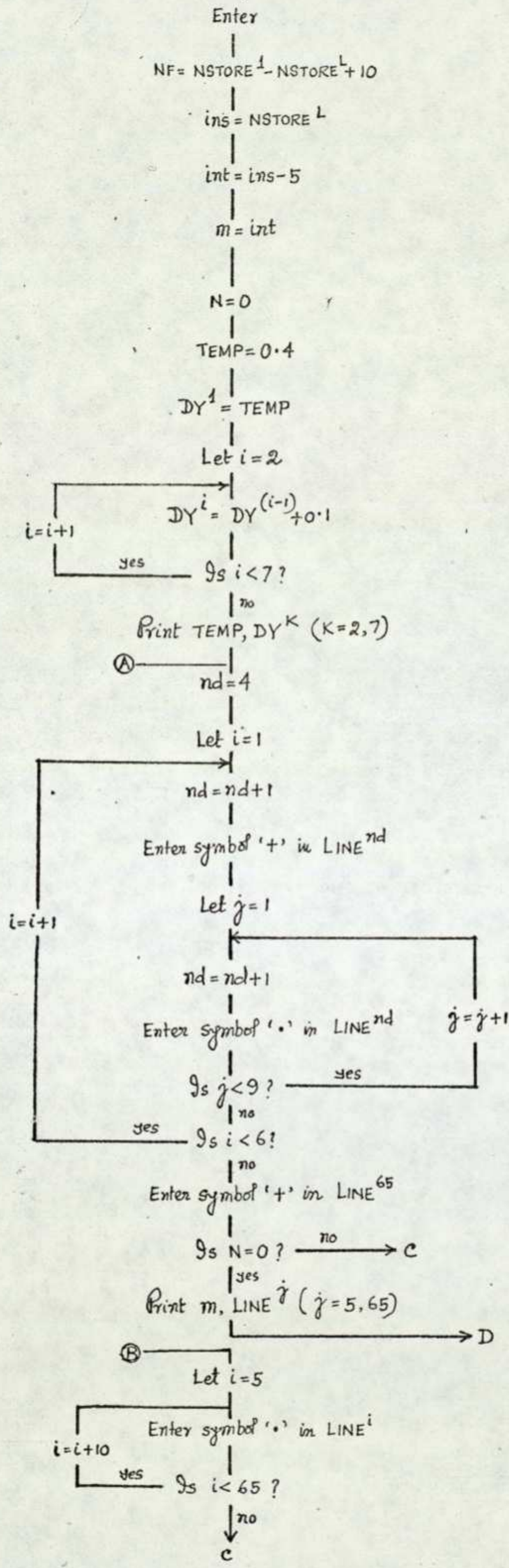
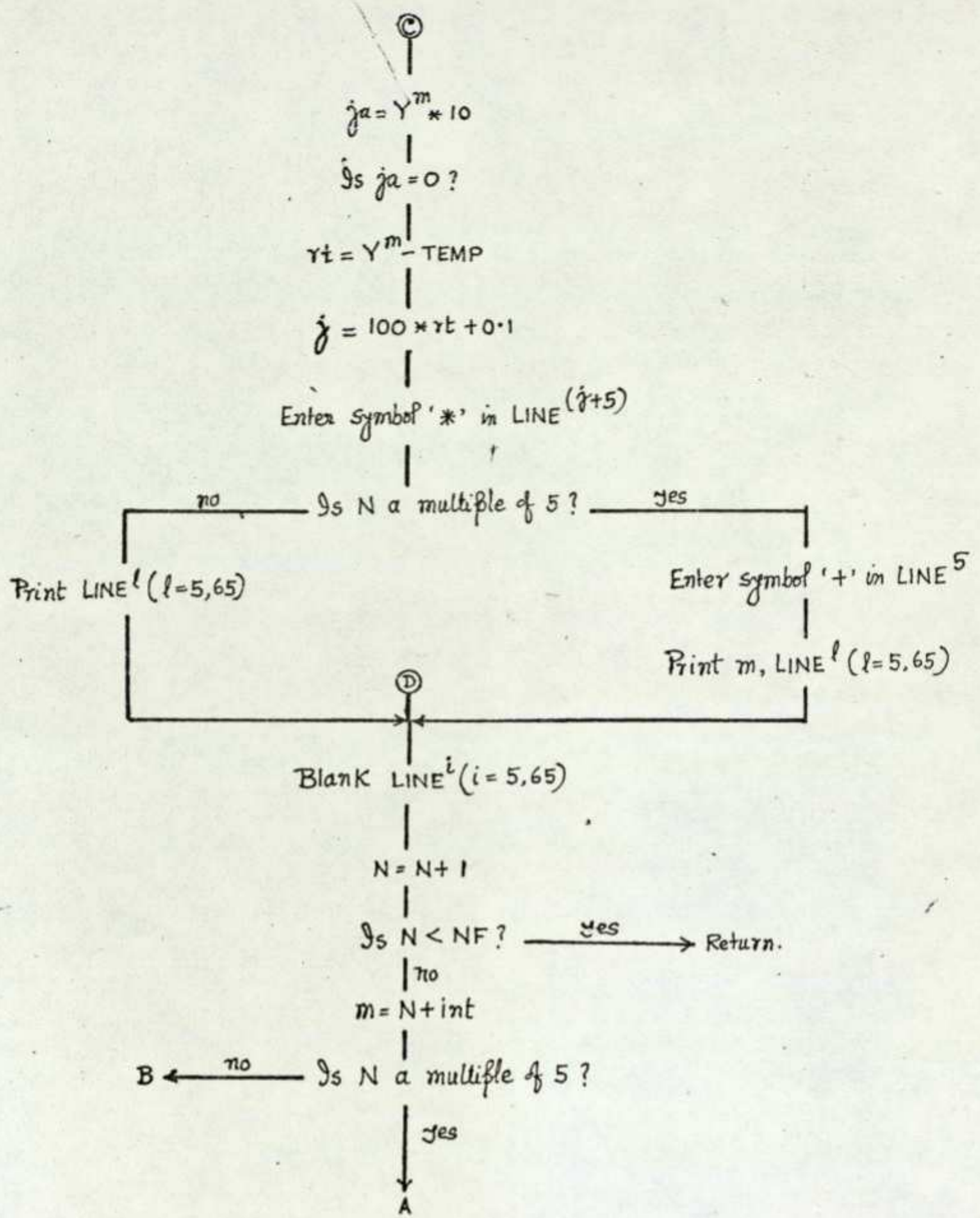


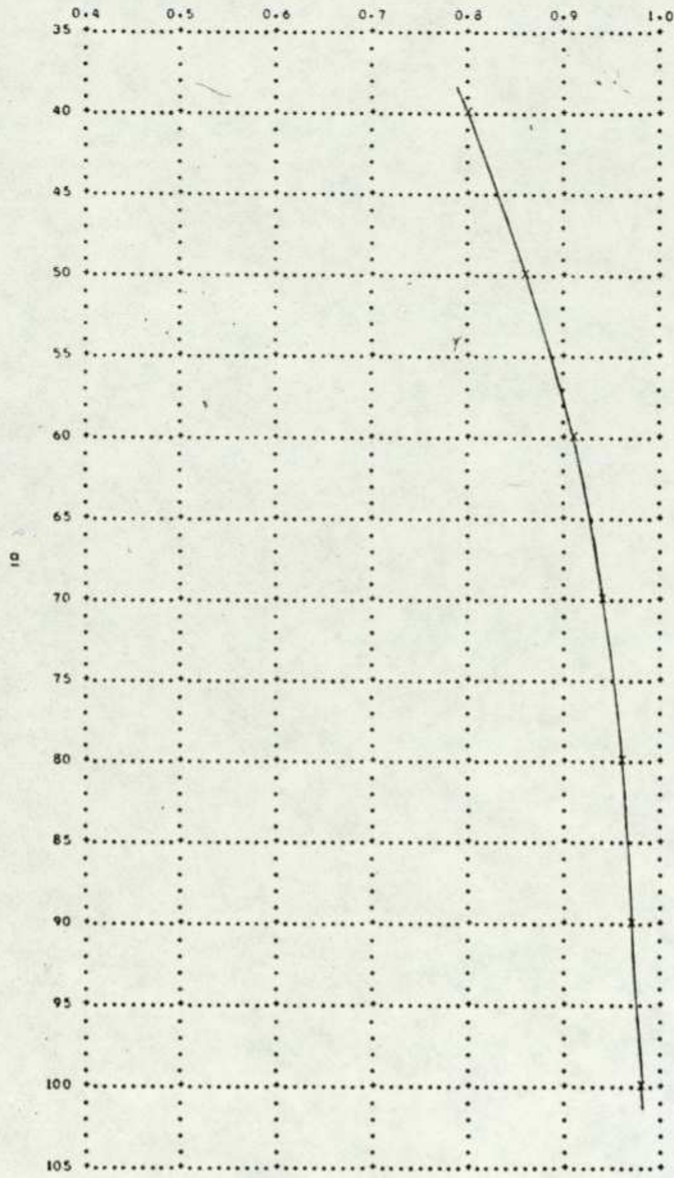
Fig 5.3 Flow-diagram of PLOT

cont'd


C
ja = Y^m * 10
Is ja = 0 ?
rt = Y^m - TEMP
j = 100 * rt + 0·1
Enter symbol '*' in LINE^(j+5)
no
Is N a multiple of 5 ?
yes
Print LINE^l (l=5,65)
Enter symbol '+' in LINE^5
Print m, LINE^l (l=5,65)
D
Blank LINE^i (i = 5,65)
N = N+1
Is N < NF ?
yes
Return.
no
m = N+int
B
no
Is N a multiple of 5 ?
yes
A

PROB. OF FAILURE=0.040

X-AXIS(VERTICAL)=TIME
Y-AXIS(HORIZONTAL)=PROB. OF DETECTION



PROB. OF FAILURE=0.060

X-AXIS(VERTICAL)=TIME
Y-AXIS(HORIZONTAL)=PROB. OF DETECTION

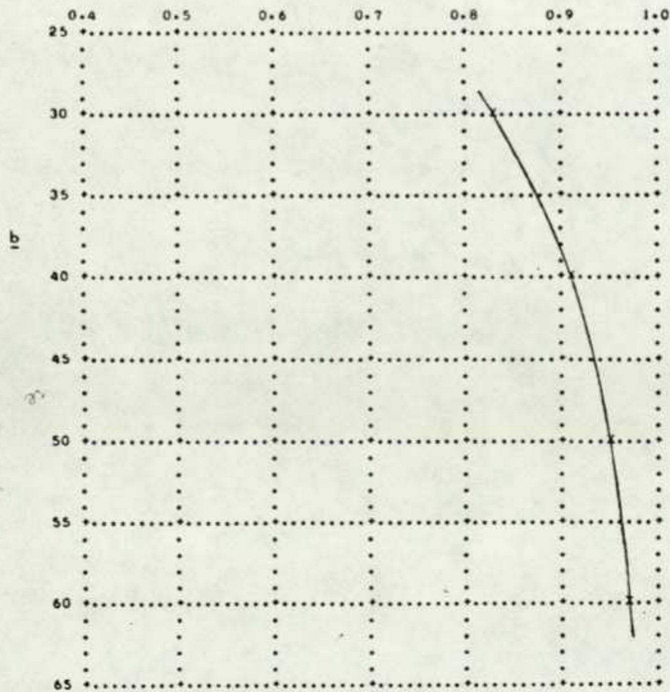


Fig 5.4 Output of program POSS

0.04 and if it is to be detected with probability of 0.80 and 0.95 then the circuit has to be tested for 40 and 80 units of time respectively (from Fig.5.4a). Fig.5.5a and b show the number of times the fault appeared during each of the 20 independent trials with 40 units and 80 units of test time; a different random seed was used in each case to get a different intermittency pattern. In Fig.5.5a in 16 out of the 20 trials at least 1 fault appeared whereas in Fig.5.5b the fault appears at least once in 19 out of 20 trials.

5.3 Detection of Intermittent Faults In Combinational Circuits

5.3.1 Detection procedure

It has been shown previously (section 3.5) how to derive a minimal test set for a combinational circuit under solid fault conditions. The test set is applied to the circuit under test and a fault, if present, is detected and located within an indistinguishable fault-set.

If each test in the same test set is now cycled for the preset time (section 5.2.2), then any well-behaved intermittent fault in the circuit will be detected. During the process of cycling, if a test does not give any false output value, then it may be assumed that the intermittent fault does not belong to the fault-set associated with the test. The pre-set time for cycling a test, in fact, gives the limit to the length of time a circuit has to be tested before one may be reasonably sure that none of the faults, which the test can detect, appears intermittently; hence as soon as a fault is detected, the testing may be discontinued.

The following steps are to be carried out sequentially in order to detect an intermittent fault:-

(1) Apply a test from the test-set derived under the solid-

PROB.OF FAIL=0.040
TEST TIME= 40

RANDOM SEED	NO.OF TIMES FAULT APPEARED
0.01	1
0.49	1
0.68	1
0.02	2
0.80	0
0.42	0
0.66	1
0.45	3
0.72	1
0.82	4
0.99	1
0.65	2
0.34	1
0.60	2
0.43	3
0.77	2
0.08	1
0.80	0
0.42	0
0.66	1

a

PROB.OF FAIL=0.040
TEST TIME= 80

RANDOM SEED	NO.OF TIMES FAULT APPEARED
0.43	4
0.24	3
0.80	0
0.09	3
0.96	1
0.33	4
0.19	1
0.64	4
0.62	7
0.31	3
0.83	3
0.69	1
0.77	5
0.26	5
0.47	3
0.11	7
0.71	6
0.08	1
0.31	3
0.83	3

b

Fig 5.5 Variation of intermittency with random seed

fault assumption.

(2) If the output of the circuit is different from the fault-free circuit then the intermittent fault has been detected, in that case the testing may be discontinued; otherwise, go to step 3.

(3) If the pre-set test-time limit has been exceeded, go to step 4 otherwise apply the test again and go back to step 2.

(4) Make one of the following decisions:

(a) Stop testing; the circuit is intermittent fault-free.

(b) Select another test from the test-set i.e. go back to step 1.

The above procedure has been computer-programmed and was named INTT; the flow diagram is shown in Fig.5.6. In order to show the intermittency of the fault occurrence, in the programme, a test is recycled even after the fault has been detected. It has been assumed for programming convenience that each test occupies one unit time. The named arrays and variables have the same meaning and uses as those described for the programme PERM of section 3.6.1.

5.3.2 A circuit example

As an example of the intermittent fault detection process the circuit of Fig.3.23 is considered. The minimal test set for the circuit, to detect all solid faults in the circuit, can be derived as in section 3.5; the test set for the circuit and the fault-set associated with each test is shown in Table 5.1. It will be assumed that gate element 7 of the circuit of Fig.3.23 gets stuck-at-0 intermittently with a probability of 0.04. If the probability of the fault detection desired is 0.8, then it can be found from Fig.5.4a that the test (0000), for fault 7 s-a-0, has to be cycled for 40 units of time. As has been

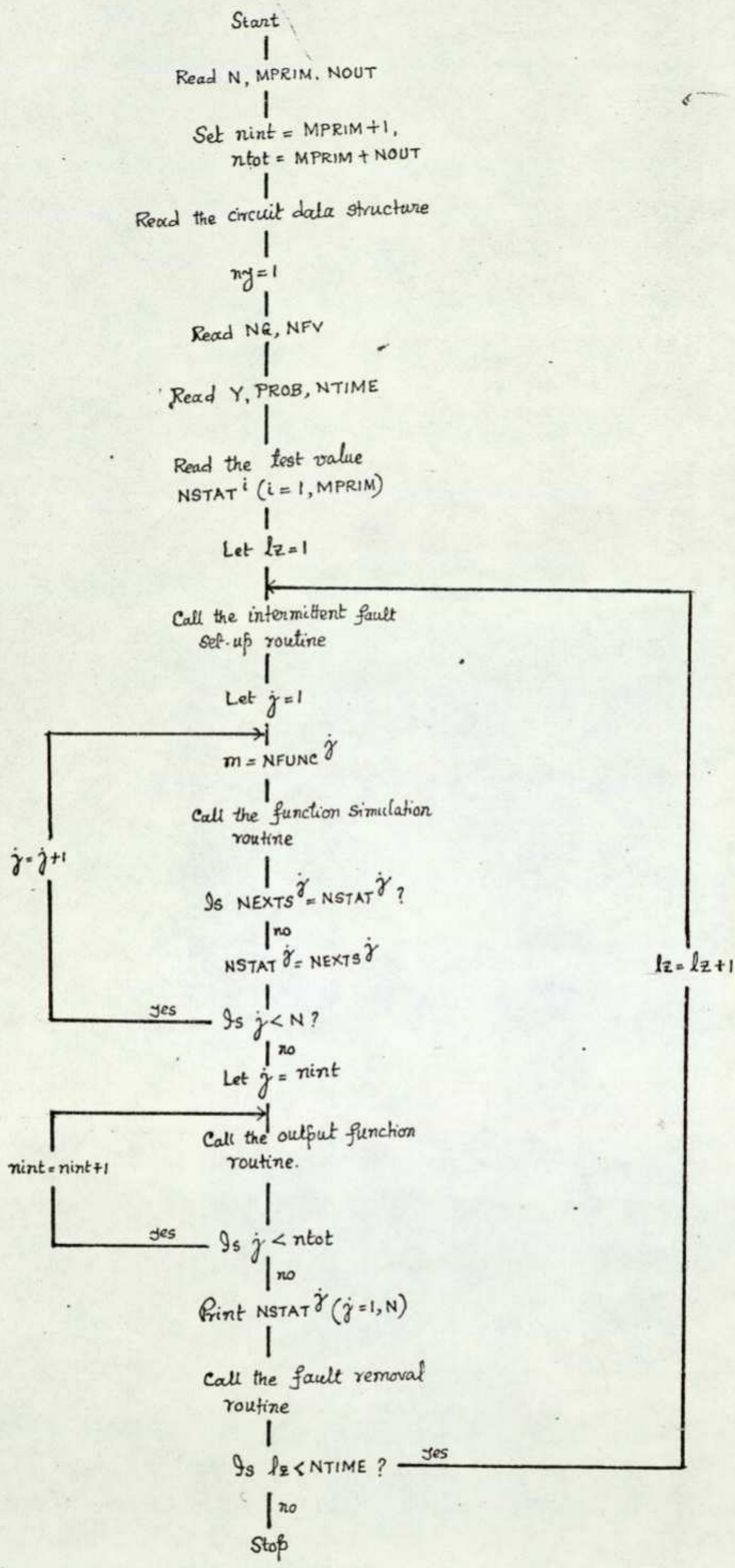


Fig 5.6 Flow-diagram of INTT

MINIMAL TEST SET AND FAULTS DETECTED

```
               8  12  13   9   6  10   7  11
               1   2   1   2   1   2   1   2

PRIM. INPUTS   1   2   3   4
 TEST VALUE    1   1   1   1

               8  12  13
               2   1   2

PRIM. INPUTS   1   2   3   4
 TEST VALUE    2   2   1   2

              11   7  13
               1   2   2

PRIM. INPUTS   1   2   3   4
 TEST VALUE    2   2   2   1

               7  10  13
               2   1   2

PRIM. INPUTS   1   2   3   4
 TEST VALUE    1   2   2   2

               6   9  13
               2   1   2

PRIM. INPUTS   1   2   3   4
 TEST VALUE    2   1   1   1
```

Table 5.1 Diagnostic test set for circuit of Fig 3.23

explained before in section 5.1.2, the pattern of fault occurrence can be varied by choosing different random seeds i.e. by different selection of Y in programme INTT. The output of program INTT used to simulate detection of the fault 7 s-a-0 is shown in Fig.5.7. The logic state of the circuit elements are printed for each application of the test; the intermittent fault and the corresponding incorrect response are underlined in Fig.5.7a. The program outputs, with different random seeds, are shown in Fig.5.7b,c,d.

5.4 Detection of Intermittent Faults in Synchronous Sequential Circuits

5.4.1 Test sequence generation

A procedure to generate test sequences to detect solid faults in synchronous sequential circuits has been described in section 4.3.2. The essence of the method is to minimize the circuit states in such a way so that most of the faults in the circuit can be detected by testing it in those states; henceforth these circuit states will be referred to as 'test states'. In the case of intermittent faults the same procedure will be used but the circuit in each test state will be tested for a pre-set time, depending on the probability of the fault occurrence, by cycling the relevant part of the test sequence. At this point it is necessary to redefine a well-behaved intermittent fault in the context of synchronous sequential circuit operation (33). Let the present state and input to the circuit at time t_q be S_q and X_q respectively and let the next state and output be S_{q+1} and Z_{q+1} if no fault is present, and S'_{q+1} and Z'_{q+1} if a fault is present permanently from t_q to t_{q+1}. A fault is said to be "well-behaved" if during the time interval t_q to t_{q+1}, it stays as a solid fault or does not occur at all.

A procedure for generating a test sequence for an intermittent

```
PROB.OF.FAILURE=0.040
TEST TIME=  40
RANDOM SEED=0.62
ELEMENT  7 STUCK-AT 1     INTERMITTENT

TEST VALUE=  1  1  1  1

      LOGIC STATE OF CIRCUIT ELEMENTS

1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2     FAULT DETECTED= 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2              a
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
        FAULT APPEARS
1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
        FAULT APPEARS
1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
```

```
PROB.OF.FAILURE=0.040
TEST TIME=  40
RANDOM SEED=0.34
ELEMENT  7 STUCK-AT 1     INTERMITTENT

TEST VALUE=  1  1  1  1

      LOGIC STATE OF CIRCUIT ELEMENTS

1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2              FAULT APPEARS
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2      1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2     FAULT DETECTED= 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2              b
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
```

Fig 5.7 Output of program INTT

(cont'd)

```
PROB.OF.FAILURE=0.040
TEST TIME=  40
RANDOM SEED=0.88
ELEMENT  7 STUCK-AT 1      INTERMITTENT

TEST VALUE=  1  1  1  1

     LOGIC STATE OF CIRCUIT ELEMENTS

1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 ! 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
      FAULT APPEARS
1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2

FAULT DETECTED= 1
```

c

```
PROB.OF.FAILURE=0.040
TEST TIME=  40
RANDOM SEED=0.30
ELEMENT  7 STUCK-AT 1      INTERMITTENT

TEST VALUE=  1  1  1  1

     LOGIC STATE OF CIRCUIT ELEMENTS

1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
      FAULT APPEARS
1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
      FAULT APPEARS
1 1 1 1 1 2 1 2 1 2 2 1 1
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2
1 1 1 1 2 2 2 2 1 1 1 1 2

FAULT DETECTED= 2
```

d

fault is given below:-

(1) Take the circuit under test from an unknown starting state to a test state by applying an appropriate initialising sequence e.g. a homing sequence followed by a transfer sequence.

(2) Apply the test input. If the circuit is in the same state as it was at the end of step 1, then go to step 4, otherwise go to step 3.

(3) Apply the appropriate transfer sequence to take the circuit back to the state it was at the beginning of step 2.

(4) Concatenate the test-sequence obtained by step 2 and step 3 (if applicable); this input sequence will be termed as 'iteration sequence'.

The test generating procedure has been computer programmed. The flow-diagram of the program TRAN is shown in Fig.5.8; subroutines LOGIC, NEWSTATE, TRANSEQ have been described in section 4.3.4.

The arrays and variables used in TRAN have the same definition and uses as in programme LINE of section 4:3.4; input variables INTI and ITER are used to specify the length of the initialising and iteration sequences respectively.

The test sequence derived by programme TRAN for the circuit of Fig.4.12 is shown in Fig.5.9. It was assumed that at the beginning of the experiment the circuit was in state 3 (Q_1Q_2=11); the test states are underlined in the Fig.5.9. The programme took about 13 secs of mill time to generate the test sequence.

5.4.2 Detection procedure

The detection procedure is started by applying the initialising sequence (if needed), and then the iteration sequence is cycled for the

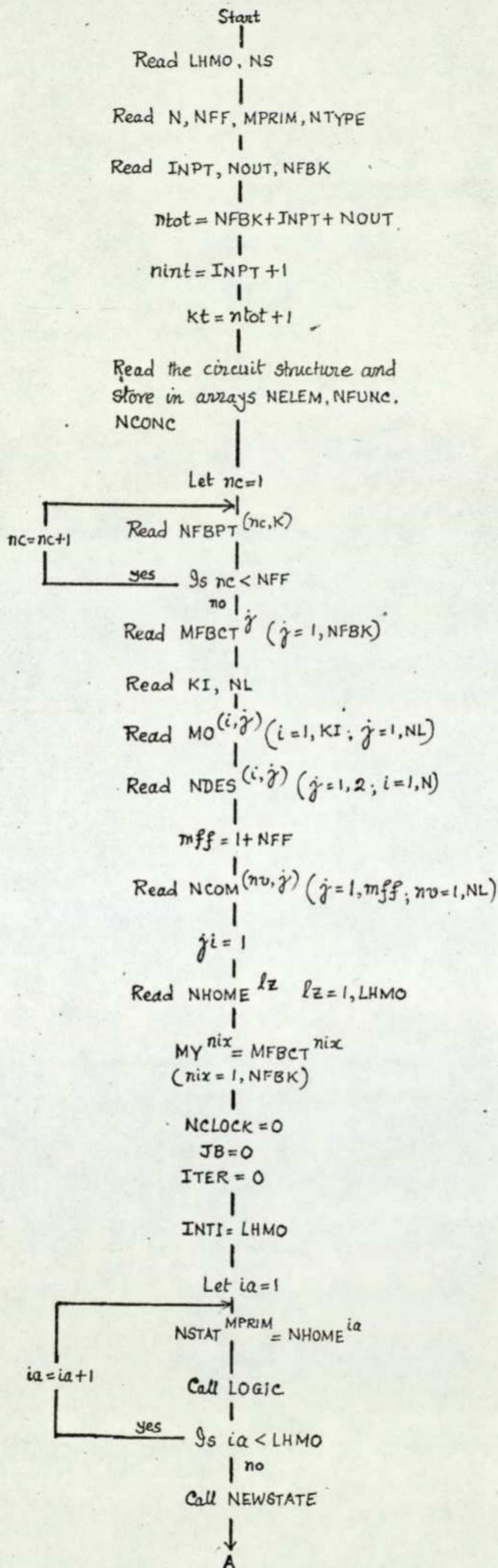
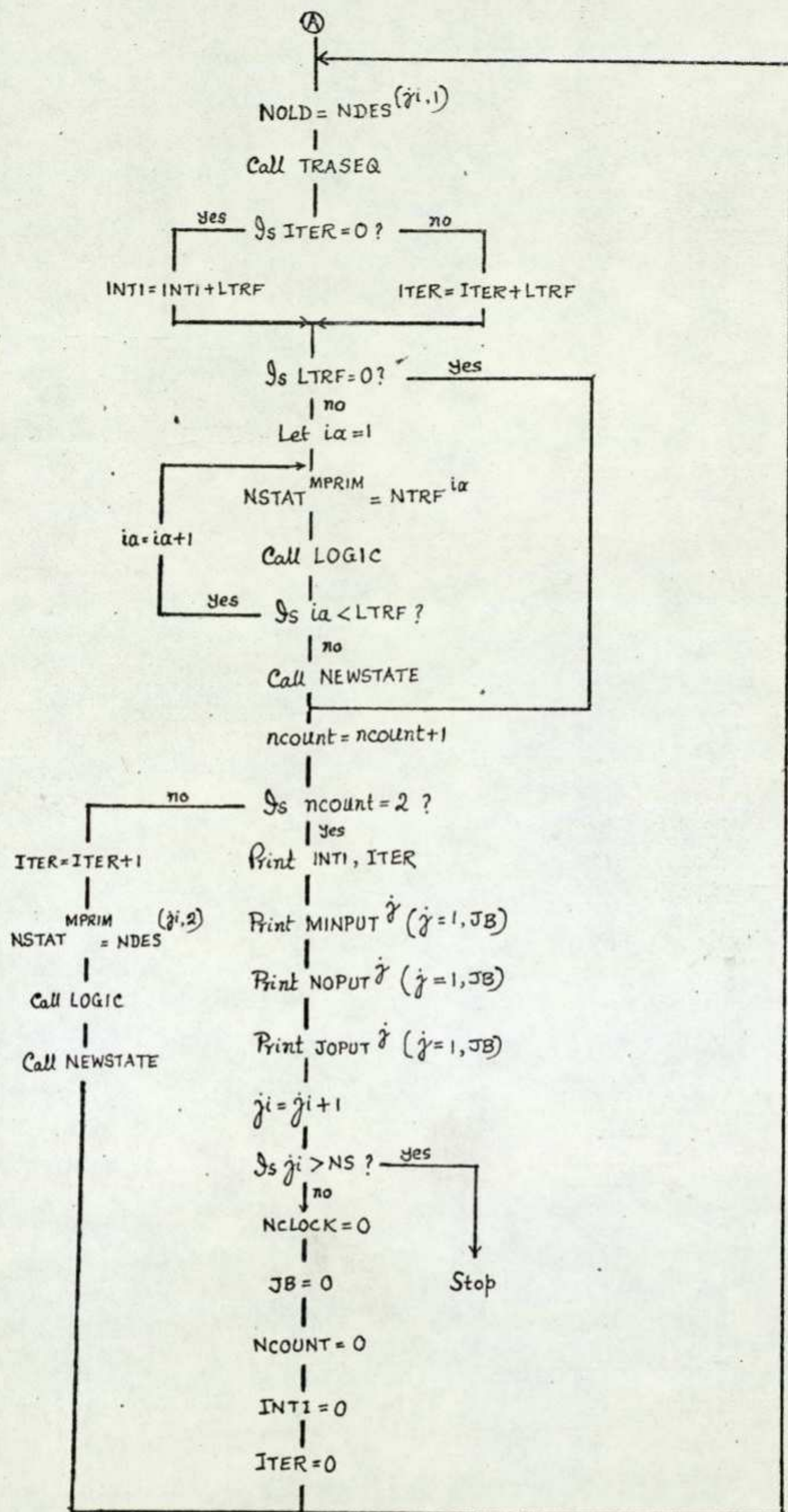


Fig 5.8 Flow-diagram of TRAN

cont'd


A
NOLD = NDES(ji,1)
Call TRASEQ
Is ITER = 0 ?
yes
no
INTI = INTI + LTRF
ITER = ITER + LTRF
Is LTRF = 0?
yes
no
Let iα = 1
NSTAT MPRIM = NTRF iα
iα = iα + 1
Call LOGIC
yes
Is iα < LTRF ?
no
Call NEWSTATE
ncount = ncount + 1
no
Is ncount = 2 ?
yes
ITER = ITER + 1
Print INTI, ITER
NSTAT MPRIM = NDES (ji,2)
Print MINPUT j (j = 1, JB)
Call LOGIC
Print NOPUT j (j = 1, JB)
Call NEWSTATE
Print JOPUT j (j = 1, JB)
ji = ji + 1
Is ji > NS ?
yes
no
NCLOCK = 0
JB = 0
Stop
NCOUNT = 0
INTI = 0
ITER = 0

INTERMITTENT FAULT-FREE SIMULATION

APPLY HOMING SEQUENCE

APPLY INPUT
2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

APPLY INPUT
1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY CLOCK PULSE 2
1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE 3

APPLY TRANSFER SEQUENCE

APPLY INPUT
2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY CLOCK PULSE 3
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

APPLY TEST VALUE

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 4
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INTIALIZING SEQUENCE= 3
ITERATION SEQUENCE= 1

TEST INPUT 2 1 2 2
TEST OUTPUT 1 2 1 1
BEFORE CLOCK PULSE

TEST OUTPUT 1 1 1 1
AFTER CLOCK PULSE

APPLY TEST VALUE

APPLY INPUT
1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY CLOCK PULSE 1
1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE 3

APPLY TRANSFER SEQUENCE

APPLY INPUT
2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY CLOCK PULSE 2
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INTIALIZING SEQUENCE= 0
ITERATION SEQUENCE= 2

TEST INPUT 1 2
TEST OUTPUT 2 1
BEFORE CLOCK PULSE

TEST OUTPUT 1 1
AFTER CLOCK PULSE

APPLY TRANSFER SEQUENCE

APPLY INPUT
1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY CLOCK PULSE 1
1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

APPLY INPUT
1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1
APPLY CLOCK PULSE 2
1 1 2 2 1 1 1 1 2 2 2 1 2 1 2 2 2 1

PRESENT STATE 2

APPLY TEST VALUE

APPLY INPUT
2 1 2 2 1 2 2 2 1 1 1 2 1 2 1 1 2 2
APPLY CLOCK PULSE 3
2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1

PRESENT STATE 3

APPLY TRANSFER SEQUENCE

APPLY INPUT
1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1
APPLY CLOCK PULSE 4
1 1 2 2 1 1 1 1 2 2 2 1 2 1 2 2 2 1

PRESENT STATE 2

INTIALIZING SEQUENCE= 2
ITERATION SEQUENCE= 2

TEST INPUT 1 1 2 1
TEST OUTPUT 2 1 2 1
BEFORE CLOCK PULSE

TEST OUTPUT 1 1 1 1
AFTER CLOCK PULSE

Fig 5.9 Output of program TRAN

pre-set time period. If the output response, during the test cycle, is different from that of the fault-free circuit, then the intermittent fault is detected and testing is discontinued. On the other hand if the fault is not detected within the pre-set time period then there are two possibilities, either

(a) the circuit is intermittent-fault free, or

(b) the circuit has an intermittent fault which is not detectable at that test state, in which case another test state is selected and a new test sequence is derived by applying steps 1-4 of the test-sequence procedure and then the detection process is re-started.

To study the behaviour of a synchronous sequential circuit in the presence of an intermittent fault, the detection procedure described above has been computer-programmed; the flow-diagram of the programme named SEQ5 is shown in Fig.5.10. The arrays and variables used in the program have the same meaning and uses as in program LAND used for solid fault detection (section 4.4.1). Some additional variables and arrays were necessary to simulate the stopping of the test application when an intermittent fault is detected or the test-time has been exceeded; these are defined as follows:-

NDEL	an integer variable which takes on the value of the time interval, $t_{q+1}-t_q$ (explained in section 5.4.1).
NTIME	an input variable specifying the time the circuit has to be tested.
NSEQ,NTV	input variables which specify the length of the initialising and iteration sequence respectively.
NCOMP1	a one-dimensional array of NCOMP1(I) variables used to store the output sequence of a fault-free

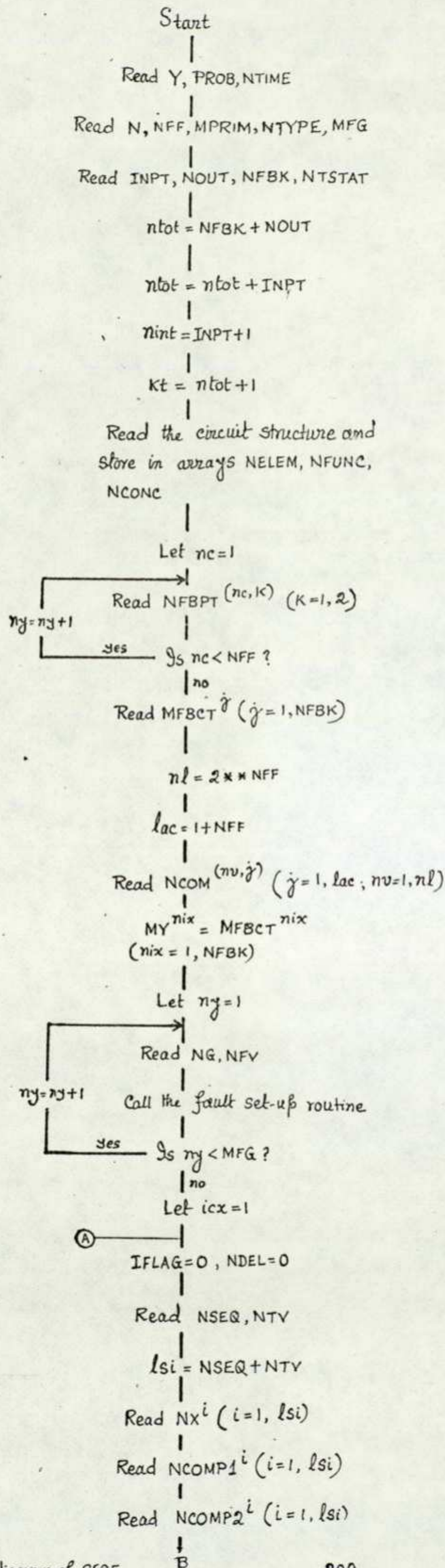
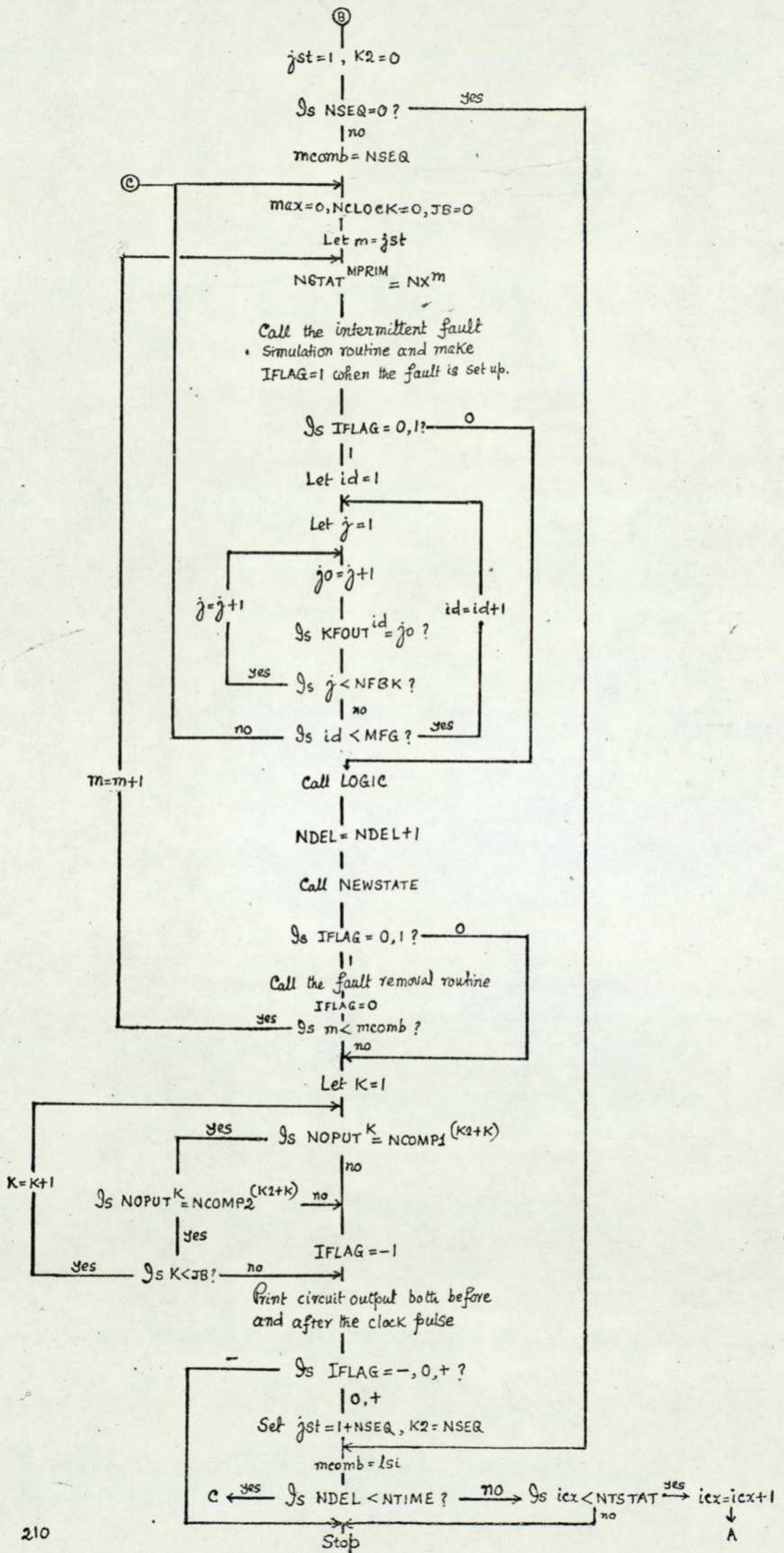


Fig 5.10 Flow-diagram of SEQ5

cont'd



B
jst=1, K2=0
Is NSEQ=0 ?
yes
no
mcomb=NSEQ
C
max=0, NCLOCK=0, JB=0
Let m=jst
NSTAT MPRIM = NX m
Call the intermittent fault simulation routine and make IFLAG=1 when the fault is set up.
Is IFLAG = 0,1?
0
1
Let id=1
Let j=1
jo=j+1
j=j+1
id=id+1
Is KFOUT id = jo ?
yes
Is j < NFBK ?
no
no
Is id < MFG ?
yes
m=m+1
Call LOGIC
NDEL = NDEL+1
Call NEWSTATE
Is IFLAG = 0,1 ?
0
1
Call the fault removal routine
IFLAG=0
yes
Is m < mcomb ?
no
Let K=1
yes
Is NOPUT K = NCOMP1 (K2+K)
no
K=K+1
Is NOPUT K = NCOMP2 (K2+K)
no
yes
IFLAG =-1
yes
Is K<JB?
no
Print circuit output both before and after the clock pulse
Is IFLAG = −, 0, + ?
−
0,+
Set jst=1+NSEQ, K2=NSEQ
mcomb=lsi
C
yes
Is NDEL < NTIME ?
no
Is icx < NTSTAT
yes
icx=icx+1
A
no
Stop

circuit before the clock pulse.

NCOMP2 — a one-dimensional array of NCOMP2(I) variables used to store the output sequence of a fault-free circuit after the clock pulse.

IFLAG — an integer variable which is set to 1 when a fault appears and to -1 when the fault is detected, otherwise it is equal to 0.

As an illustration it is assumed that in the circuit of Fig.4.12 gate element 11 gets stuck-at-1 intermittently and this happens with a probability of 0.04. If the fault is to be detected with a probability of 0.8 then it can be found from Fig.5.4a that an iteration sequence has to be cycled for 40 units of time at a test state of the circuit. Assuming the circuit is initially at state 3 (Q_1Q_2=11), the initialising sequence is applied to take it to the first test state i.e. state 4 (Fig.5.9); then the iteration sequence is applied. The result is shown in Fig.5.11.

As a second illustration, gate element 15 is assumed to be stuck-at-0 intermittently with a probability of 0.06. If the probability of detection desired is 0.8 then the circuit has to be tested for 30 units of time at a test state (Fig.5.4b). With the assumed initial state of 4 (Q_1Q_2=10) the iteration sequence 0 (Fig.5.9) is repeatedly applied for 30 units of time but the fault was not detected. Since at the end of the test time, the circuit still remains at state 4, the next iteration sequence 01 is applied, the result is shown in Fig.5.12.

```
PROB. OF FAILURE=0.040
RANDOM SEED=0.490
TIME UNITS=  40
ELEMENT 11 STUCK-AT 2  INTERMITTENT
```

(A)

LOGIC STATE OF CIRCUIT ELEMENTS

```
APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  2
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  3
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2 1 2
 IDEAL OUTPUT 1 2 1
ACTUAL OUTPUT 1 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1 1
ACTUAL OUTPUT 1 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0
```

(A)

(A)

```
APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY  CLOCK PULSE  1
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

       INPUT 2
 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1
ACTUAL OUTPUT 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0
```

(B)

Fig 5.11 Output of program SEQ5

(cont'd)

(B)

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

(C)

(C)

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

APPLY INPUT
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1
APPLY CLOCK PULSE 1
2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 1
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 1
AFTER CLOCK PULSE

FAULTS APPEARED= 0

FAULT APPEARS

APPLY INPUT
2 2 1 1 2 2 1 1 1 2 2 1 1 1 2 2 1 2
APPLY CLOCK PULSE 1
2 2 1 1 2 2 1 1 1 2 2 1 1 1 2 2 1 2

PRESENT STATE 4

INPUT 2
IDEAL OUTPUT 1
ACTUAL OUTPUT 2
BEFORE CLOCK PULSE

IDEAL OUTPUT 1
ACTUAL OUTPUT 2
AFTER CLOCK PULSE

FAULTS APPEARED= 1

FAULT DETECTED

```
PROB. OF FAILURE=0.060
RANDOM SEED=0.600
TIME UNITS=  30
ELEMENT 15 STUCK-AT 1  INTERMITTENT
```

LOGIC STATE OF CIRCUIT ELEMENTS

```
APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  2
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  2
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3
```

(A)

(A)

```
APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  2
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  2
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

APPLY INPUT
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1
APPLY  CLOCK PULSE  2
 2 2 1 1 2 1 1 1 1 2 1 1 1 1 2 2 2 1

PRESENT STATE  4

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0
```

(B)

Fig 5.12 Output of program SEQ5

(cont'd)

(B)

```
APPLY INPUT
 1 2 1 1 2 2 1 2 1 1 2 1 1 2 1 2 1 2
APPLY  CLOCK PULSE  1
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1

PRESENT STATE  3

      FAULT APPEARS

APPLY INPUT
 2 2 2 1 1 1 2 1 1 1 1 2 1 1 1 2 2 1
APPLY  CLOCK PULSE  2
 2 2 2 1 1 1 2 1 1 1 1 2 1 1 1 2 2 1

PRESENT STATE  3

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 2 1
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  1

APPLY INPUT
 1 2 2 1 1 1 1 2 2 1 2 1 2 2 1 2 2 1
APPLY  CLOCK PULSE  1
 1 1 2 2 1 1 1 1 2 2 2 1 2 1 2 2 2 1

PRESENT STATE  2

APPLY INPUT
 2 1 2 2 1 2 2 2 1 1 1 2 1 2 1 1 2 2
APPLY  CLOCK PULSE  2
 2 2 2 1 1 1 2 1 1 2 1 2 1 1 2 2 2 1

PRESENT STATE  3

        INPUT 1 2
 IDEAL OUTPUT 2 1
ACTUAL OUTPUT 1 2
BEFORE CLOCK PULSE

 IDEAL OUTPUT 1 1
ACTUAL OUTPUT 1 1
 AFTER CLOCK PULSE

FAULTS APPEARED=  0

FAULT DETECTED
```

6. PROJECT DISCUSSION AND CONCLUSION

Many research papers have been published on logic circuit testing which are basically the variations of the well-known methods like D-Algorithm and Boolean difference. In this thesis new techniques, based on the idea of fault-folding have been described for the diagnosis of solid and intermittent faults in combinational and sequential circuits. The advantages and disadvantages of these techniques will be highlighted in the following two sections.

6.1 Diagnosis of Solid Faults

6.1.1 Combinational circuits

The D-Algorithm method is perhaps more widely used than any other method for fault-diagnosis in combinational logic networks. However one of its shortcoming is that it may generate redundant tests when applied to logic networks containing monotone function elements; such redundancy is inefficient for automatic test generation on computers (102). Although D-Algorithm does not generate redundant tests for non-monotone function elements e.g. exclusive-OR, parity generators etc. their use is rather limited. The procedure described in this thesis, on the other hand, is very well-suited for test generation of monotonic function elements; by using the functional properties of the logic, it efficiently generates a minimum number of tests for fault-detection.

Consider for example an eight input NAND gate (e.g. Texas Instruments SN7430), a monotone function. Using the D-Algorithm, a test set will be generated after 18 different main D-cube operations, one for the stuck-at-0 and one for the stuck-at-1 faults for each of the eight

input pins and the output pins. However after the application of the method described here, only nine tests result, which are found to be sufficient for the fault-detection. Another example of the generation of redundant tests by D-Algorithm when applied to the network of Fig.6.1, which is composed of monotone functions is shown in Table 6.1. A reduced number of tests can obviously be obtained after deleting the equivalent failure, shown in Table 6.2 but the procedure developed in this thesis takes equivalent failures into account during the process of test generation itself; hence it automatically produces a reduced test set e.g. test set in Table 6.2 for the network of Fig.6.1.

The test-generating algorithm has been computer-programmed so that the fault-detecting test set can be derived directly from a given circuit structure. The computer time necessary to find a test set depends more on the number of inputs of the network than the number of gates in it. This is due to the use of a trial and error technique in selecting a control input combination which together with the static inputs result in a test for an indistinguishable fault-set. This trial and error technique becomes very time-consuming when the number of control input combinations to be applied are very large; also it leads to a waste of effort if a fault in the set is untestable. It seems that further work is necessary for more efficient selection of control inputs in order to make the fully automatic test generation by this technique really practicable for circuits with a large number of inputs. In a recent paper Akers (103) suggested a technique which may prove useful. A possible solution, in the meantime, would probably be to apply a definite number of control input combinations and if a completely defined test is still not realised for a fault set then to intervene manually to check for possible undetectable faults in the set.

On the credit side the reduction in the number of tests achieved

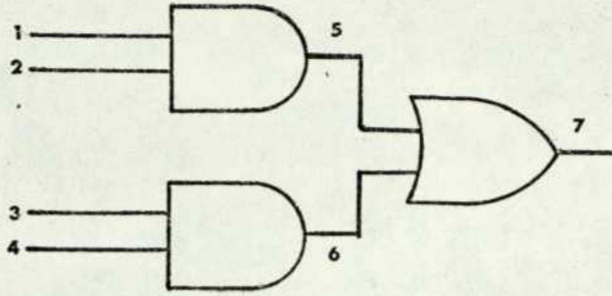


Fig.6.1

	Assumed stuck-at failure	Test input 1 2 3 4	Faults
1	5 s-a-0	1 1 0 0	5 s-a-0,7 s-a-0
2	5 s-a-1	0 0 0 0	5 s-a-1,7 s-a-1
3	6 s-a-0	0 0 1 1	6 s-a-0,7 s-a-0
4	6 s-a-1	0 0 0 0	6 s-a-1,7 s-a-1
5	7 s-a-0	1 1 1 1	7 s-a-0
6	7 s-a-1	0 0 0 0	7 s-a-1

Table 6.1 Test and gate output faults

	Test Input 1 2 3 4	Faults
1	0 0 0 0	5 s-a-1,6 s-a-1,7 s-a-1
2	1 1 0 0	5 s-a-0,7 s-a-0
3	0 0 1 1	6 s-a-0,7 s-a-0

Table 6.2 Reduced test set

is significant (Figure 3.33, section 3.5); in each case only a minimum of tests were generated. When a method based on exclusive-OR operator (104) was used to generate tests for circuit examples 2 and 6 of Figure 3.33, the number of tests generated were 10 and 34 respectively whereas only 5 and 10 tests were found to be sufficient by the method described here, a reduction of 50% and 39% respectively. Since each test in a fault set also locates a fault within an indistinguishable class, no extra effort is necessary for locating faults. A small number of tests means less testing time, this together with the fault-locating capability of each test should increase the cost-saving.

6.1.2 Sequential circuits

An algorithm based on fault-folding technique is presented which successfully detects single and multiple faults in synchronous sequential circuits. This method takes into account both structure and behaviour of a network while generating a test sequence; previous methods used either structure or behaviour. In the computer industry the most widely used technique for test generation of sequential circuits is to build an iterative combinational circuit model of the given sequential circuit and then to employ the D-Algorithm method to generate a test for a logic fault in this iterative circuit. As one might expect, the size of the transformation of the sequential circuit into the iterative circuit has a substantial cost. Also the D-Algorithm does not take any account of the repeated effects of a fault on the internal state of a circuit during the application of the test sequence and therefore sometime fails to find a test for a fault even though one exists (82). For example, it is not possible to generate a test for fault 7 s-a-1 in the circuit of Fig.4.31a (section 4.5.2) by the D-Algorithm method (82); however as has been shown in that section, the fault is detected by the test procedure developed for sequential circuit testing in this thesis.

In conventional checking experiments for synchronous sequential circuits, it is necessary to verify every state transition; input symbols are applied to make the machine go through every state transition and in each case the transition is verified by displaying its response to the distinguishing sequence. If the circuit has not any distinguishing sequence, the resulting experiments become very long to be practically useful. However, the test procedure described here is intended to test the sequential circuit in some selected states only, thus making the general state transition verification procedure irrelevant, consequently it is of little importance whether the circuit has a distinguishing sequence or not. The length of the checking experiment so designed depends mainly on the length of the transfer sequences used to manouvre the circuit from one pre-selected test state to the other. A computer-orientated procedure has been developed in section 4.2.2 for the shortest transfer sequence generation.

The test sequence generated by using the procedure is somewhat more restricted than the conventional checking experiments as far as fault coverage is concerned, but it is more practical. The test procedure has been computerised and for automatic test sequence generation the program requires besides the circuit structure the states at which the circuit has to be tested with the associatedtest values and a homing sequence. The fault-location procedure described for synchronous sequential circuits may be used for identifying indistinguishable failures for a test which consists of the input symbol and state of the circuit at that instant.

A fault-detection test procedure for asynchronous or unclocked sequential networks has also been derived. A synchronous model of the given asynchronous circuit is built up and a test sequence is generated for this synchronous version using the algorithm developed for

synchronous circuits. Primary input values together with the state of the circuit at any instant of time constitute the test input to the asynchronous circuit when a fault is detected it may be localised to an indistinguishable class without any extra effort.

The test generation procedure described here for sequential circuits uses a pure combinational version of the sequential circuit instead of the currently used iterative combinational version, therefore during the computer-aided test sequence generation it would use less core store.

One of the problems encountered was the initialisation of the circuit in the presence of faults. The initialisation or homing sequence takes the circuit under test from an unknown starting state to a known starting state. However, if the homing sequence, generated for a fault-free circuit, is applied to a faulty circuit then the circuit may enter into a state which is different from the state it would have entered if there was no fault in the circuit. With the method described here it is still possible to detect the fault and locate it within a fault equivalent class. But in the presence of certain faults, the homing sequence produces an output which does not correspond to any state in the circuit i.e. it is not possible to conclude what state the circuit has moved into. In that situation one can say that a fault has been detected but it is not possible to give any diagnostic information. Further work is necessary to solve the problem of initialisation in presence of faults.

6.2 Diagnosis of Intermittent Faults

Algorithmic procedures to detect single intermittent faults in combinational and synchronous sequential circuits have been described. These procedures are based on probability theory and involve repeated

application of tests, which are designed for solid fault detection, within a pre-set time period. The time periods are calculated from the Poisson distribution when the probability of failure is known. It has been proposed that if a fault appears with a known probability then it has a certain probability of being detected within the selected time-period; the longer the time-period the better is the chance of detecting the fault. The basic assumption was that an intermittent fault has to occur only once during the test time-period for it to be detected. Although it is always true for an intermittent fault in a combinational logic circuit, since the same test is cycled within the time-period so when the fault occurs it is immediately detected, this is not necessarily applicable to sequential circuits. The iteration sequence used to detect an intermittent fault in sequential circuits consists of a test value applied when the circuit is in a test state and additional input values (transfer sequence) to bring it back to the test state if it moves into a different state after the application of the test value. If a fault occurs during the application of the test input value, then it will be definitely detected provided it is detectable at that state and can be localised as explained in section 4.4. However if the fault occurs during the application of an input value which belongs to the transfer sequence, then it will only be detected if by its presence during the input duration it affects the circuit output response or makes the circuit move to an incorrect state; subsequent application of the other input values lead the circuit to a series of incorrect states which may eventually result in an incorrect response at the output e.g. fault 15 intermittently stuck-at-0 (Fig.5.12). The incorrect state transitions may not always result in an incorrect output response, thereby inhibiting the detection of the fault. Therefore if an intermittent fault appears only during the state transition part of an

iteration sequence, it might not be detected by the procedure described here; however if some means of observing the circuit state is available, obviously this problem will not arise.

The intermittent fault detection procedures are based on the assumption that the probability of malfunction is known. Although it seems a restrictive assumption, the probability could perhaps be estimated from the reliability requirement of the circuit during operation. A circuit, in this context, may be considered to be highly reliable if during its operation the probability of an intermittent fault occurrence is very low. If the network reliability is made directly proportional to a 'goodness factor', G, where

$$G = \frac{1}{\text{Prob. of intermittent fault occurrence}}$$

then a high G ($1 \leq G < \infty$) will mean high reliability; $G \rightarrow \infty$ refers to a fault-free circuit and G = 1 means a solid fault is present. Thus, depending on the reliability desired, the circuit has to be tested to screen out a fault which has a probability of occurrence equal to 1/G.

Another important point is to decide how long a test or a test sequence has to be repeated to increase the probability of detection. As Breuer (33) indicated, it is costly to detect intermittent faults and the cost increases with time. So there has to be trade-off between probability of detection and cost.

6.3 Conclusion

The main objectives of this project, to investigate the problem of diagnosing solid and intermittent faults in logic circuits, have been successfully attained. New methods have been developed to diagnose solid faults in combinational and sequential circuits; these methods

have also been extended to intermittent faults. The results obtained have been compared, wherever possible, with those obtained by the well established methods and the relative advantages have been pointed out.

Certain practical limitations were encountered during the implementation of the methods. Firstly, these methods have been applied only to relatively simple circuits; this is mainly due to the fact that circuits used in industry are generally not made available. Secondly a high level language, FORTRAN IV, was used to write simulation programmes. Two important measures of a good simulator are the speed of simulation (compilation and execution) and the use of a minimum amount of storage. The importance of the second objective is two fold; first minimization of storage requirements will enable the simulator to handle larger circuits and second, its use will not be limited to large computing facilities. FORTRAN IV is a procedure-oriented language i.e. a user can write programmes without requiring him (or her) to know the exact procedures the computer uses to compute. It requires a special programme called translator to change the source programme to an object programme written in machine language. Consequently there is a loss of efficiency since the host computer must translate the source programme and may need to allocate memory for the source programme and translator as well as the object programme. This constraint may require more memory space and often longer execution time than an assembly language programme, however FORTRAN IV programmes are considerably more machine independent and easier to update. Besides in a research project as this where proving the validity of the new techniques are of more concern than making them immediately practicable, the relative advantages of assembly language may justifiably be sacrificed for the more acceptable FORTRAN IV and the small increase in execution time and storage.

The procedures described for testing logic circuits are based on the fault-folding technique which can be used only for structural testing of circuits containing AND, OR, NAND and NOR gates. The test generation procedure for sequential circuits combines the structural testing of the output circuit with the functional testing of the feedback part. Therefore even if EXCLUSIVE-OR gates are present in the feedback part of a circuit, EX-OR function can be realized by using other types of gates.

In sequential circuit testing the fault-folding was carried out to find all the faults that can possibly be detected under a particular state. Since the number of states in a synchronous sequential circuit can be enormous, the problem was how to find a selected number of states that would test cover most of the faults in the circuit.

The computer programme developed for automatic test generation of combinational circuits can handle circuits having up to 120 elements with a maximum of 15 primary inputs. This programme was designed primarily to handle the size of circuits available to the author; by suitable modifications of array sizes the programme can be made to generate tests for much larger circuits.

From the presently available information it is not possible to say precisely by what factor the mill time will increase with the increase of control input combinations. The problem, of course, is that although the first few control inputs will detect many faults in a fault-set, detecting every fault in a set may take a very long time. In the author's opinion the mill time will depend on (a) the total number of control input combinations applied to detect each fault in a set (b) the number of elements in a fault-set and (c) the number of fault-sets considered during the test generation.

The diagnostic procedure developed in this thesis assumes access only to the primary input and output of a circuit. Therefore if additional observable points are available there will be a corresponding increase in diagnostic resolution. For example if the state variables of sequential circuits are observable, then the testing of sequential circuits by the method described here would not be significantly more difficult than for combinational networks; indeed the basic philosophy of the test procedures described in this thesis was to bridge the current gap between combinational and sequential circuit testing. But inclusion of extra test points is contrary to the layout and packaging objectives of LSI. Therefore this instant solution is not practicable, but this does illustrate that constraints may be placed on the actual logic circuit design in order to make the circuit 'easily testable'. In a recent paper, Bennetts et al (105) surveyed and summarised the progress made so far in the theory and practice of testable logic design. It seems one almost inevitable consequence of improving testability would be to use additional logic in implementing a design. On the other extreme, various strategies are also available for fault-tolerant design which again is accomplished by additional logic. Since LSI implementation makes redundant logic economic to use, the possible ways of using redundancy, internal to a LSI circuit, to simplify fault diagnosis should be examined. It is possible, however, by using far-sighted design techniques to lay out logic boards for easy testing, fault isolation and troubleshooting; various suggestions for the improvements in the design of complex printed circuit boards for testability may be found in (106) (107) (108).

The use of medium- and large-scale integrated circuits has made a large impact on the way digital systems are designed and manufactured; in the past the effort was on a minimum gate design, a

much more valid goal today is to minimize the number of IC packages (109). When the MSI component is introduced the currently used structural testing procedure begins to indicate that unnecessary effort is being expended for isolation of one faulty gate among a few dozen gates imbedded within the MSI chip; the fault can only be corrected by replacing the entire chip. This becomes even more apparent for the case of LSI chips. In some cases, the internal logic or circuit structure of a package may not even be known so that gate-level test generation and fault simulation are impossible. This increase in size and complexity of the smallest replaceable module in digital system gives a different complexion to the fault-diagnosis problem. It seem future work should be directed towards generation of diagnostic tests at the physical package or module level; a module level test generation procedure for combinational circuits has been described by Batni et al (110).

A limitation of the present-day fault simulators is that they deal only with the "stuck-at" faults, but these are not the only type of faults that can occur in real circuits. One fault that occurs in LSI circuits is the 'delay' fault; this is introduced during the manufacturing process (3). All the gates may function logically but the propagation through some of them may be three times longer than the normal value. Clearly there is a need for the development of better fault models for LSI technologies.

Although research work has been continuing in the area of fault-diagnosis, it is limited to the detection and location of solid faults; the generation of test procedures for diagnosing intermittent faults is still being neglected. As far as the author is aware only two research papers (34), (35) have been published on intermittent fault detection in the last two and a half years. The reason for the lesser interest on the diagnosis of intermittent faults is that hardly any information is

available on the mechanism of this type of failure. It has been shown in this thesis that provided the probability of intermittent fault occurrence is known, the techniques for solid fault-detection may be applied to the intermittent fault detection as well. It seems semiconductor manufacturers can give a lead in this direction by providing failure statistics with their supplied products.

For the future it seems inevitable that with the ever growing complexity of integrated logic circuits where todays LSI is tomorrows MSI, ever increasing attention must be paid to methods such as have been described, for rendering the location and diagnosis of faults a more efficient process.

APPENDIX A. Random events in time : Poisson distribution

The Poisson distribution characterizes several random processes which are frequently encountered such as shot noise in thermionic valves, transmission of telegraph signals etc. (111) (112). It is an example of a discrete probability distribution since it is defined only for discrete values of random variables.

Let an event occur randomly and it is assumed that the probability of the event occurring during a given interval is statistically independent of the number of times it occurred previously and that the probability is proportional to the time interval Δt, provided Δt is small. Thus the probability of the event occurring once in Δt is given by

$$P(1,\Delta t) = \lambda\Delta t \qquad \text{..... (1)}$$

where λ is a constant to be determined.

Since Δt is small, the probability of the event occurring more than once in Δt is assumed to be negligible. Thus for small Δt the following approximate relation is obtained

$$P(0,\Delta t) + P(1,\Delta t) = 1. \qquad \text{..... (2)}$$

The probability of the event not occurring during an interval of length $(t+\Delta t)$ is given by

$$P(0,t+\Delta t) = P(0,t)P(0,\Delta t). \qquad \text{..... (3)}$$

This follows from the previous assumption that the occurrence of the event during the interval Δt is independent of the number of times it occurred during t, and consequently the expression for statistical independence is valid. From equations (1) and (2),

$$P(0,\Delta t) = 1 - \lambda\Delta t$$

and equation (3) becomes

$$\frac{P(0,t+\Delta t) - P(0,t)}{\Delta t} = - \lambda P(0,t). \qquad \ldots\ldots (4)$$

The left hand side of the equation can be recognised as the definition of $\frac{dP(0,t)}{dt}$ as $\Delta t \rightarrow 0$; hence

$$\frac{dP(0,t)}{dt} = - \lambda P(0,t).$$

This first-order differential equation has the solution

$$P(0,t) = e^{-\mu t}$$

with the initial condition

$$P(0,0) = \lim_{\Delta t \to 0} P(0,\Delta t) = 1.$$

The last result follows from equations (1) and (2). Thus the probability of the event not occurring during time t is given by $e^{-\mu t}$.

Next the probability of the event occurring exactly x times during an interval of length $(t+\Delta t)$ will be determined. For small Δt, the event must occur once or not at all in Δt; therefore

$$P(x,t+\Delta t) = P(x-1,t)P(1,\Delta t) + P(x,t)P(0,\Delta t) \ldots.(5)$$

Using the results obtained previously for $P(1,\Delta t)$ and $P(0,\Delta t)$ equation (5) becomes

$$\frac{P(x,t+\Delta t)-P(x,t)}{\Delta t} + \lambda P(x,t) = \lambda P(x-1,t). \qquad \ldots\ldots (6)$$

As $\Delta t \rightarrow 0$, this gives the differential equation

$$\frac{dP(x,t)}{dt} + \lambda P(x,t) = \lambda P(x-1,t) \qquad \ldots\ldots (7)$$

as a recursion equation relating $P(x,t)$ to $P(x-1,t)$. Since

$P(x,0) = 0$, the solution to this first order linear differential equation is,

$$P(x,t) = \lambda e^{-\lambda t} \int_0^t e^{\lambda t} P(x-1,t)dt. \qquad ..... (8)$$

Equation (8) enables one to determine $P(x,t)$ from $P(x-1,t)$ by the following continuous process. Assuming $x=1$, $P(1,t)$ can be obtained from equation (8).

$$P(1,t) = \lambda e^{-\lambda t} \int_0^t e^{\lambda t} . e^{-\lambda t} dt$$

$$= (\lambda t)e^{-\lambda t}. \qquad ..... (9)$$

From this result, $P(2,t)$ can be determined and then $P(3,t)$ and so on. The final general result, namely, the probability that the event will occur exactly x number of times during time t is

$$P(x,t) = \frac{(\lambda t)^x}{x!} e^{-\lambda t}, \; x = 0,1,2 \cdots \qquad ..... (10)$$

which is the Poisson probability distribution.

The constant λ can be evaluated from the average number of times the event occurred during the time interval t. Since the possible number of times the event can occur during the interval ranges from 0 to ∞, it is found that

$$Av.x = K = \sum_{x=0}^{\infty} xP(x,t) = e^{-\lambda t} \sum_{k=0}^{\infty} \frac{x(\lambda t)^x}{x!}, \; x \geqslant 0.$$

Since the $x=0$ term is zero, the equation becomes

$$K = \lambda t . e^{-\lambda t} \sum_{x=1}^{\infty} \frac{(\lambda t)^{x-1}}{(x-1)!} .$$

But

$$\sum_{x=1}^{\infty} \frac{(\lambda t)^{x-1}}{(x-1)!} = 1 + \lambda t + \frac{(\lambda t)^2}{2!} + \ldots\ldots = e^{\lambda t}$$

therefore,

$$K = \lambda t . e^{-\lambda t} . e^{\lambda t} = \lambda t$$

REFERENCES

1. CASE, P. W. "Evolution of design automation"
   Computer, pp.21-22, May/June, 1972

2. RUSO, R. L. "Design automation"
   pp.18-20, ibid

3. FALK, H. "Design for production"
   IEEE Spectrum, pp.48-53, Oct., 1975

4. GORDON, G. and EFRON, R. "General purpose digital simulation and example of its applications"
   IBM Syst.Jour., Vol.3, pp.22-24, 1964

5. DIMSDALE, B. and MARKOWITZ, H. M. "A description of SIMSCRIPT language"
   IBM Syst.Jour., Vol.3, pp.57-67, 1964

6. DULEY, J. R. and DIETMEYER, D. L. "Translation of a DDL digital system specification to Boolean equations"
   IEEE Trans. on Comput., Vol.C-18, pp.305-313, April, 1969

7. FRIEDMAN, T. D. and YANG, S. C. "Methods used in an automated logic design generator (ALERT)"
   ibid, pp.593-613, July, 1969

8. BREUER, M. A. (Ed.) Design automation of digital systems
   Prentice-Hall, 1972

9. PENNEY, WILLIAM M. and LAN, LILLIAN (Ed.) MOS integrated circuits
   Van Nostrand Reinhod Company, 1972

10. SUSSKIND, A. K. "Diagnosis for logic networks"
    IEEE Spectrum, pp.40-47, Oct., 1973

11. MATTREA, LUCINDA "Component reliability"
    Electronics International, Vol.48
    Part 1, No.20, pp.91-98, Oct., 1975
    Part 2, No.22, pp.87-94, Oct., 1975

12. AVIZIENIS, A. "Design of fault-tolerant computers"
    Proc FJCC, Vol.31, pp.733-743, 1967

13. YEN, Y. T. "Intermittent failure problems of four-phase MOS circuits"
    IEEE J. Solid-state Circuits, Vol.SC-3, pp.107-110, June, 1969

14. CHANG, H. Y., MANNING, E., METZE, F. Fault diagnosis of digital systems
    Wiley Interscience, 1970

15. McCLUSKEY, E. J. "Test and diagnosis procedures for digital networks"
Computer, pp.17-20, Jan./Feb., 1971

16. SHORT, R. A. "The attainment of reliable digital system through the use of redundancy - a survey"
IEEE Comput. group news, Vol.2, pp.2-17, March, 1968

17. VON NEUMANN, J. "Probabilistic logics and the synthesis of reliable organism from unreliable components"
Automata Studies, from Annals of Mathematical Studies, No.34, Princeton Univ. Press, pp.43-99, 1956

18. MATHUR, F. P. and AVIZIENIS, A. "Reliability analysis and architecture of a hybrid redundant digital system : generalised triple modular redundancy with self-repair"
SJCC, Vol.36, pp.375-383, May, 1970

19. TYRON, J. G. "Quadded logic" in Redundancy techniques for computing systems
R.H.Wilcox and W.C.Mann, Eds., Spartan, pp.205-228, 1962

20. PIERCE, W. H. Failure-tolerant computer design
New York : Academic Press, 1965

21. KALASCHKA, T. F. "Two contributions to redundancy theory"
Proc. Eighth Annual Symp. on Switching and Automata Theory, pp.175-183, 1967

22. FREEMAN, H. A. and METZE, G. "Fault-tolerant computers using "Dotted Logic" redundancy techniques"
IEEE Trans. on Comput., Vol.C-21, No.8, pp.867-871, Aug., 1972

23. AVIZIENIS, A., GILLEY, G., MATHUR, F., RENNELS, D., ROHR, J. and RUBIN, D. "The STAR (self-testing and repairing) Computer: An investigation of the theory and practice of fault-tolerant computer design"
IEEE Trans. on Comput., Vol.C-20, No.11, pp.1312-1321, Nov., 1971

24. DOWNING, R. W., NOWAK, J. S. and TUOMENOKSA, L. S. "No.1 ESS Maintenance Plan"
BSTJ, Vol.43, pp.1961-2019, 1964

25. ARMSTRONG, D. B. "A general method of applying error-correcting to synchronous digital systems"
BSTJ, Vol.40, pp.572-598, 1961

26. RAY-CHAUDHURI, D. "On the construction of minimally reliable system design"
ibid pp.595-611

27. HSIAO, M. Y. and TOU, J. T. "Application of error-correcting codes in computer reliability studies"
IEEE Trans. on Reliability, Vol.R-18, No.3, pp.108-118, Aug., 1969

28. KEINER, W. L. "Functional testing - a user looks at logic simulation"
Proc. 10th Design Automation Workshop, Portland, Oregon, pp.151-158, June, 1973

29. MEI, C. Y. M. "Bridging and stuck-at faults"
Int.Symp. on Fault-tolerant Computing, pp.91-94, June, 1973

30. FRIEDMAN, A. D. "Diagnosis of short-faults in combinational circuits"
ibid, pp.95-99

31. BENNETTS, R. G. and LEWIN, D. W. "Fault diagnosis of digital systems- a review"
Computer, Vol.14, pp.199-206, 1971

32. BALL, M. and HARDIE, F. "Effects and detection of intermittent failures in digital system"
Proc. FJCC, Vol.35, pp.329-335, 1969

33. BREUER, M. A. "Testing of intermittent faults in digital circuits"
IEEE Trans. on Comput., Vol.C-22, No.3, pp.241-246, March 1973

34. KAMAL, S. and PAGE, C. V. "Intermittent faults: A model and a detection procedure"
IEEE Trans. on Comput., Vol.C-23, No.7, pp.713-719, July 1974

35. KAMAL, S. "An approach to the diagnosis of intermittent faults"
IEEE Trans. on Comput., Vol.C-24, No.5, pp.461-467, May 1975

36. PARKER, K. P. and McCLUSKEY, E. J. "Analysis of logic circuits with faults using input signal probabilities"
IEEE Trans. on Comput., Vol.C-24, No.5, pp.573-578, May 1975

37. McCLURE, R. M. "A programming language for simulating digital systems"
Jour.ACM, Vol.12, pp.14-22, Jan., 1965

38. McKINNEY, J. G. "Synchronous logic simulation"
Proc. SHARE-ACM Design Automation Workshop, pp.19-22, 1967

39. HARDIE, F. H. and SUHOCKI, R. J. "Design and use of fault-simulation for Saturn computer design"
IEEE Trans. on Comput., Vol.EC-16, pp.412-429, August, 1967

40. HAYS, G. G. "Computer-aided design: simulation of digital design logic"
IEEE Trans. on Comput., Vol.C-18, No.1, pp.1-10, Jan., 1969

41. ULRICH, E. G. "Time sequenced logical simulation based in circuit delay of active network paths"
Proc. ACM 20th Nat.Conf., pp.437-448, 1965

42. SCHLAEPPI, H. P. "A formal language for describing machine logic, timing and sequences - LOTIS"
IEEE Trans. on Comput., Vol.EC-13, No.8, pp.439-448, August, 1964

43. DULEY, J. R. and DIETMEYER, D. L. "A digital system design language (DDL)"
IEEE Trans. on Comput., Vol.C-17, No.9, pp.850-860, Sept., 1968

44. CHU, Y. <u>Introduction to computer organisation</u>
Prentice-Hall, New Jersey, 1970

45. SZYGENDA, S. A. "TEGAS 2 - Anatomy of a general purpose test generation and simulation system for digital logic"
Proc. 9th Design Automation Workshop, Dallas, pp.116-127, June, 1972

46. SZYGENDA, S., ROUSE, D. and THOMPSON, E. "A model and implementation of a universal time delay simulator for large digital networks"
Proc. SJCC, pp.207-216, 1970

47. WILLIAMS, T. "Logic design"
Digital Design, Part 1, pp.118-121, April, 1975
Part 2, pp.58-65, May, 1975

48. BREUER, M. A. and HARRISON, R. L. "Procedures for eliminating static and dynamic hazards in test generation"
IEEE Trans. on Comput., Vol.C-23, No.10, pp.1069-1077, Oct., 1974

49. LUDLOW, M. P. "The diagnosis of solid and intermittent faults in electronic logic circuits"
PhD thesis, 1973, The City University

50. FRIEDMAN, A. D. "Fault detection in redundant circuits"
IEEE Trans. on Comput., Vol.EC-16, pp.99-100, Feb., 1967

51. SU, Y. H. S. and CHO, Y. C. "A new approach to the fault-location of combinational circuits"
IEEE Trans. on Comput., Vol.C-21, No.1, pp.21-30, Jan., 1972

52. FRIEDMAN, A. D. and MENON, P. R. <u>Fault-detection in digital circuits</u>
Prentice-Hall, New Jersey, 1971

53. ARMSTRONG, D. B. "On finding a nearly minimal set of fault-detection tests for combinational logic nets" IEEE Trans. Elect. Comput., Vol.EC-15, pp.66-73, Feb., 1966

54. SCHNEIDER, P. R. "On the necessity to examine D-chains in diagnostic test generation - An example" IBM Jour. of R. and D., Vol.11(1), p.114, 1967

55. ROTH, J. P. "Diagnosis of automata failures : A calculus and a method" IBM Jour. of R. and D., Vol.10, pp.278-291, July 1966

56. ROTH, J. P., BOURICIUS, W. G. and SCHNEIDER, P. R. "Programmed algorithms to compute tests to detect and distinguish between failures in logical circuits" IEEE Trans.Comput., Vol.C-16, pp.567-580, Oct., 1967

57. POAGE, J. F. "Derivation of optimum tests to detect faults in combinational circuits" Proc. Symp. on Mathematical theory of Automata, pp.483-528, 1963

58. SELLERS, F. F., HSIAO, M. Y. and BEARNSON, C. L. "Analyzing error with Boolean difference" IEEE Trans. on Comput., Vol.C-17, pp.676-683, July, 1968

59. ACKERS, S. B. "On a theory of Boolean functions" Jour.Soc. of Ind. Appl. Math., Vol.7, pp.487-498, Dec., 1959

60. McCLUSKEY, E. J. and CLEGG, F. W. "Fault equivalence in combinational logic networks" IEEE Trans.Comput., Vol.C-20, No.11, pp.1286-1293, Nov., 1971

61. SCHERTZ, D. R. and METZE, G. "A new representation for faults in combinational digital circuits" IEEE Trans.Comput., Vol.C-21, No.8, pp.858-866, August, 1972

62. TO, KLIN "Fault-folding for irredundant and redundant combinational circuits" IEEE Trans.Comput., Vol.C-22, No.11, pp.1008-1015, Nov., 1973

63. HAYES, J. P. "A NAND model for fault diagnosis in combinational logic networks" IEEE Trans.Comput., Vol.C-20, No.12, pp.1496-1505, Dec., 1971

64. DIEPHUIS, R. J. "Fault analysis for combiational logic networks" PhD dissertation, Dept. of Electrical Engineering, MIT, Cambridge, Sept., 1969

65. GAULT, J. W. "The application of fault indistinguishability in combinational networks"
Univ. of Iowa, Ames, Themis Project Tech., Rep.13, CFSTIAD 692 420, July, 1969

66. HAYES, J. P. "A study of digital network structure and its relation to fault diagnosis"
Co-ordinated Science Lab., Univ. of Illinois, Urbana-champaign, Rep.R-467, CFSTIAD 707 691, May, 1970

67. SCHERTZ, D. R. and METZE, G. "On the design of multiple fault diagnosable networks"
IEEE Trans.Comput., Vol.C-20, pp.1361-1364, Nov., 1971

68. HAYES, J. P. "The fan-out structure of switching functions"
Jour.ACM, Vol.22, No.4, pp.551-571, Oct., 1975

69. KOHAVI, I. and KOHAVI, Z. "Detection of multiple faults in combinational logic networks"
IEEE Trans.Comput., Vol.C-21, No.6, pp.556-568, June, 1972

70. LEWIN, D. Theory and design of digital computers
Nelson, 1972

71. AMAR, V. and CONDULMARI, N. "Diagnosis of large combinational networks"
IEEE Trans. on Elect.Comput., Vol.EC-16, pp.675-680, 1967

72. PREISS, R. J. 'Fault test generation', Ch.VII, Design automation of digital systems (8)

73. SESHU, S. and FREEMAN, D. N. "The diagnosis of asynchronous sequential switching systems"
IEEE Trans. on Elect. Comput., Vol.EC-11, pp.459-465, August 1962

74. POAGE, J. F. and McCLUSKEY, E. J. "Derivation of optimum test sequences for sequential machines"
5th Annual Symp. on Switching Theory and Logical Design, pp.121-132, 1964

75. KUBO, H. "A procedure for generating test sequences to detect sequential circuit failures"
NEC, Res. Develop., No.12, 1968

76. PUTZOLOU, G. R. and ROTH, J. P. "A heuristic algorithm for the testing of asynchronous circuits"
IEEE Trans. on Comput., Vol.C-20, pp.639-646, June, 1971

77. FUNATSU, S., WAKATSUKI, N. and ARIMA, T. "Test generation systems in Japan"
Proc. 12th Design Automation Conference, Boston, pp.114-122, June, 1975

78. CHANG, S. J., SU, Y. H. and BREUER, M. A. "Detection and location of multiple stuck-type failures in synchronous sequential networks"
IEEE Computer Group Repository, R-72-223

79. BOSSEN, D. C. and HONG, S. J. "Cause effect analysis for multiple fault detection in combinational networks"
IEEE Trans. on Comput., Vol.C-20, pp.1252-1257, Nov., 1971

80. HSIAO, M. Y. and CHIA, D. K. "Boolean differences for fault detection in asynchronous sequential machines"
IEEE Trans. on Comput., Vol.C-20, pp.1356-1361, Nov., 1971

81. ARIMA, T. and TSUBOYA, M. "A new heuristic test generation algorithm for sequential circuits"
Proc.11th Design Automation Workshop, Colorado, pp.169-176, June, 1974

82. MUTH, P. "A nine-valued circuit model for test generation"
IEEE Trans. on Comput., Vol.C-25, pp.630-636, June, 1976

83. MOORE, E. F. "Gedanken-experiments on sequential machines"
Automata Studies, Princeton Univ. Press, Princeton, New Jersey, pp.129-153, 1956

84. GILL, A. Introduction to the theory of finite state machine
McGraw-Hill, New York, 1972

85. HENNIE, F. C. "Fault-detecting experiments for sequential circuits"
Proc.5th Annual Symp. on Switching Theory and Logical Design, pp.95-110, Nov., 1964

86. KIME, C. R. "An organisation for checking experiments on sequential circuits"
IEEE Trans. on Comput., Vol.EC-15, pp.113-115, Feb., 1966

87. GÖNENC, G. "A method for the design of fault-detection experiments"
IEEE Trans. on Comput., Vol.C-22, pp.397-399, April, 1973

88. FRIEDMAN, A. D. and MENON, P. R. "Restricted checking sequences for sequential machines"
IEEE Trans. on Comput., Vol.C-22, pp.397-399, April, 1973

89. THAYSE, A. "Testing of asynchronous sequential switching circuits"
Phillips Research Repts. 27, pp.99-106, 1972

90. MEALEY, G. H. "A method for synthesizing sequential circuits" Bell Sys. Tech. Jour., Vol.34, pp.1045-1085, Sept., 1955

91. KOHAVI, Z. Switching and finite automata theory McGraw-Hill, 1970

92. HIBBARD, T. N. "Least upper bounds on minimal terminal state experiments for the classes of sequential machines" Jour. ACM, Vol.8, pp.601-612, Oct., 1961

93. KOHAVI, I. "Fault-diagnosis of logic circuits" Proc. of the 10th Annual Symp. on Switching and Automata Theory, pp.166-173, Oct., 1969

94. BREUER, M. A. "The effects of races, delays and delay faults on test generation" IEEE Trans. on Comput., Vol.C-23, pp.1078-1092, Oct., 1974

95. RUTMAN, R. A. "Fault-detection test generation for sequential logic by heuristic tree search" IEEE Computer Group Repository, R-72-187

96. SHIVA, S. G. and TROY NAGLE, H. "Let a computer design memory circuits" Electronic Design, Vol.22, No.23, pp.122-127, Nov., 1974

97. EICHELBERGER, E. B. "Hazard detection in combinational and sequential switching circuits" IBM Jour. R. and D., Vol.9, pp.90-99, 1965

98. ASHKINAZY, A. "Fault-detection experiments in asynchronous sequential machines" Proc.11th Annual Symp. on Switching and Automata Theory, pp.88-96, 1970

99. SU, Y. H. S. "Fault diagnosis in logic networks" Computer Design, pp.87-92, Jan., 1974

100. ROTENBERG, A. "A new pseudo-random number generator" Jour.ACM, Vol.7, No.1, pp.75-77, Jan., 1960

101 KIRCH, A. M. Introduction to statistics with FORTRAN Holt, Rinehart and Winston, Inc., 1973

101a. HOYT, J. P. Introduction to probability theory International Textbook Company, 1967

102. KOGA, Y. and HIRATA, F. "Fault locating test generation for combinational logic circuits" Int.Symp. on Fault-tolerant Computing, pp.131-136, 1972

103. AKERS, S. B. "A logic system for fault test generation" IEEE Trans. on Comput., Vol.C-25, pp.620-630, June, 1976

104. BENNETTS, R. G. "A realistic approach to detection test generation for combinational logic circuits" The Computer Journal, Vol.15, No.3, pp.238-246, 1972

105. BENNETTS, R. G. and SCOTT, R. V. "Recent developments in the theory and practice of testable logic design" Jour. IERE, Vol.45, No.11, pp.667-679, Nov., 1975

106. BOSWELL, F. R. "Designing testability into complex logic boards" Electronics International, Vol.45, No.17, pp.116-119, August, 1972

107. SCHNEIDER, D. "Designing logic boards for automatic testing" Electronics International, Vol.47, No.15, pp.100-104, July, 1974

108. BLUNDEN, D. F., BOYCE, A. H. and LAWSON, D. J. "Some aspects of testing logic circuits" Digital Process, Vol.1, No.2, pp.171-176, Summer, 1975

109. LEWIN, D. "Outstanding problems in logic design" Jour. IERE, Vol.44, No.1, pp.9-17, Jan., 1974

110. BATNI, R. and KIME, C. R. "A module-level testing approach for combinational networks" IEEE Trans. on Comput., Vol.C-25, No.6, pp.594-604, June, 1976

111. THOMAS, J. B. <u>An introduction to statistical communication theory</u> John Wiley & Sons, 1969

112. PANTER, P. F. <u>Modulation, noise and spectral analysis</u> McGraw-Hill, 1965

ACKNOWLEDGEMENT

I would like to express my gratitude to my supervisor, Dr. James Missen, for his invaluable advice and illuminating suggestions during the course of the project. Without his help and encouragement this work would not have been accomplished.

I am grateful to the Science Research Council for an 'instant' award and particularly to Mrs. B. [redacted] for her patient typing and correcting of the manuscript.