Fault-tolerant digital machine design.

MITCHELL, R.B.

1978

ROBERT GORDON'S INSTITUTE OF TECHNOLOGY, ABERDEEN

SCHOOL OF ELECTRONIC AND ELECTRICAL ENGINEERING

FAULT-TOLERANT DIGITAL MACHINE DESIGN

by

ROBERT B MITCHELL

This thesis is presented as a requirement for a C N A A Master of Philosophy Degree.

ROBERT B MITCHELL
February 1978

DECLARATION

I hereby declare that this thesis, composed by myself, is a record of work carried out by myself, has not been accepted in any previous application for a degree, and that all sources of information have been acknowledged.

Robert B Mitchell

ACKNOWLEDGEMENTS

I am indebted to my supervisor Dr. N. D. Deans for his invaluable assistance and encouragement throughout this study.

I would also like to express my gratitude to the technical staff at R.G.I.T. for their constructive comments and Mrs. G. Mitchell for her patience and co-operation in typing this thesis.

CONTENTS Page No.

CONTENTS (contd.) Page No.

ABSTRACT.

This thesis describes the work of the author towards an M. Phil. degree in the field of Fault-Tolerant Digital Machine Design.

As a preview, fail-safe machine design is discussed in detail, and various new design techniques are presented.

The fundamentals of fault-tolerant digital machine design are presented, along with various design techniques.

Both hardware construction and computer simulation programs have been used liberally throughout this study.

CHAPTER 1. INTRODUCTION

One of the prime requirements of a computing system is the ability to operate correctly over a sufficiently long period of time. Therefore, certain measures must be taken, either in the initial design or in the subsequent testing of the system, in order to satisfy this requirement. In the past, computers were used largely in an off-line, batch-processing mode, and the consequences of undetected hardware malfunctions were relatively minor. However, because of the increasing use of computers in on-line, real-time applications such as the control of nuclear reactors, spacecraft trajectories and military equipment, notably missile-guidance systems, incorrect computer operation in any of these applications can be potentially disastrous. Furthermore, the increasing size and complexity of digital computers have made it more and more difficult to ensure correct machine operation.

There are various failures which may occur within a digital machine, although this study shall be concerned only with logical faults. These produce some changes in the logical behaviour of the machine. Thus, component failures which affect voltages, currents, shapes of pulses or delays, but do not alter the logical function realised by a particular circuit, will not be considered. Also included in this category are failures of power supply, external input signals, and clock signals.

A fault in a digital circuit is a physical defect of one or more components, which can cause the circuit to malfunction. Ageing or manufacturing defects can cause a component to gradually deteriorate, giving rise to marginal faults. Noise and overly close tolerances can cause intermittent faults, which are timevarying, being present in some intervals of time and absent in others. Many faults that are originally intermittent eventually become _solid_, which implies that the malfunction is permanent until repairs are made.

Throughout this study, only _solid logical_ faults will be considered. The majority of solid faults which occur in digital circuits create either stuck-at-high or stuck-at-low conditions. The basic T.T.L. logic gate is illustrated in Figure 1(a). A simplified version, representing a NAND gate as shown in Figure 1(b), illustrates some of the stuck-faults that can occur in this type of gate.

Mode 1 represents a permanently open base connection, while mode 2 represents a permanently open collector connection. Under these faults, the transistor output Q would appear to be stuck-at-high. On the other hand, mode 3 represents a short from collector to emitter, therefore Q would appear to be permanently stuck-at-low.

Any of these failure modes can seriously upset the functional capabilities of a digital circuit, hence the need for circuits which can automatically detect a fault as soon as it occurs, or more important in this study, circuits which can continue to operate correctly even although a fault has occurred.

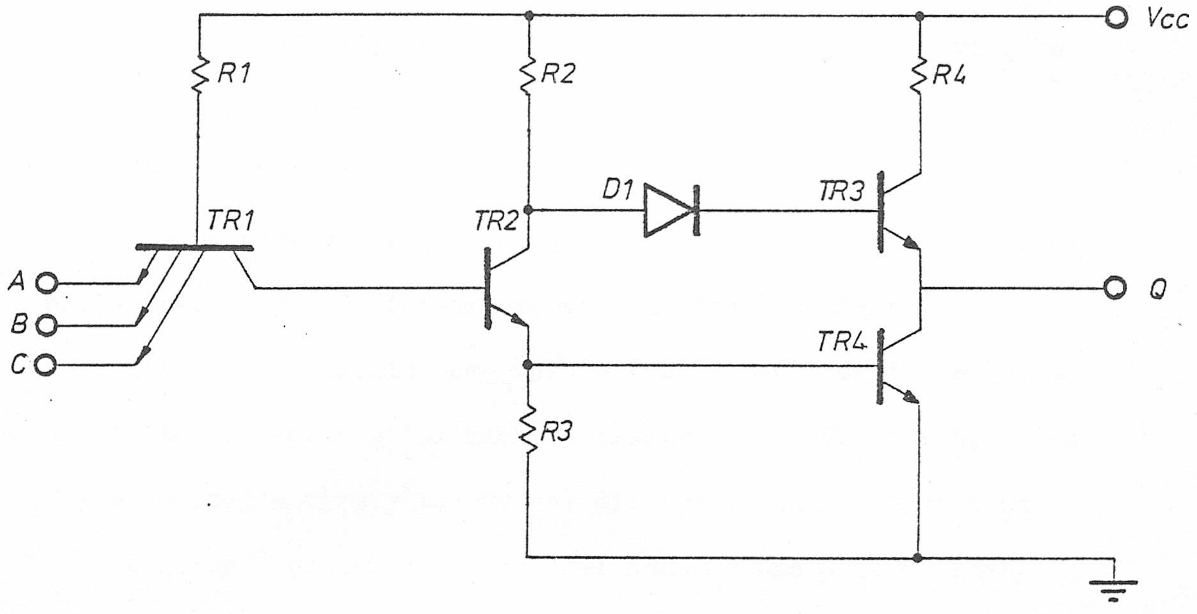
Improvements in the behaviour of digital circuitry under fault conditions can be achieved by:

(a) Fail-Safe Design.

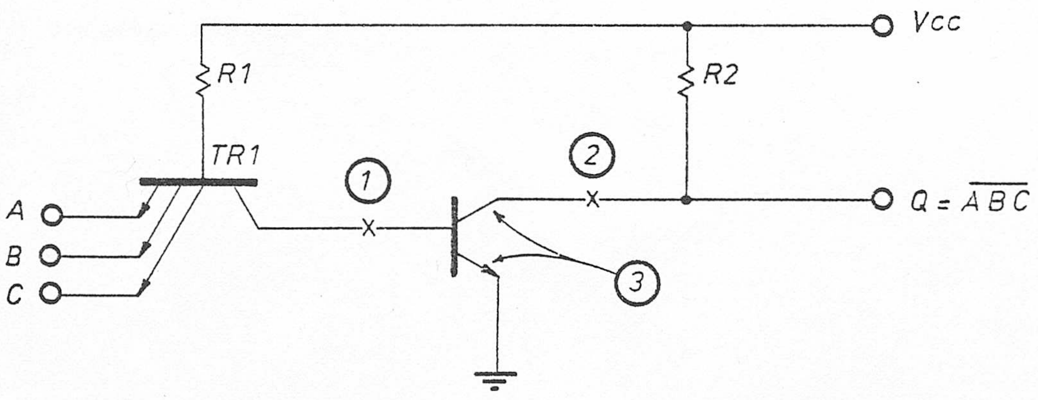
(b) Fault-Tolerant Design.

(c) Easily-testable Circuit Design

Fail-safe and fault-tolerant design are similar in the respect that they involve special design techniques, whereas diagnostic



(a) basic T. T. L. gate



(b) simplified version showing failure modes

Figure 1

① base open

② collector open

③ collector - emitter short

testing involves the application of test sequences to conventionally designed circuitry. There is an abundance of literature on the subject of diagnostic testing, therefore it will not be considered in this study. [1] However, the design and operation of fail-safe and fault-tolerant circuitry will be investigated in detail.

Chapter 2 deals with the various mathematic definitions and theorems necessary for theddesign of fail-safe digital circuitry, while Chapter 3 puts forward several design techniques.

Chapter 4 presents some theoretical aspects on the subject of fault-tolerant digital machine design, while Chapter 5, 6 and 7 are devoted entirely to several different design techniques.

Chapter 8 presents the overall conclusions and outlines possible topics of future research. The various computer programs used throughout this study are listed in the Appendices.

Note that, throughout this study, all design techniques apply to synchronous, sequential circuits, unless otherwise stated, since these provide most of the computational power in any modern computer system.

CHAPTER 2. FUNDAMENTALS OF FAIL-SAFE DIGITAL MACHINE DESIGN

Fail-safe digital circuits are designed in such a way that, if a logical fault occurs within the system, the output values always adopt a known "safe" state. This means that the extent of damage is much less than if the system fails with any other output state. Therefore, a logical system is said to be "fail-safe" if, in the event of failures, its output is either error-free or assumes a safe value.

As a practical example, consider a traffic controller with two light signals, red and green. The green signal denotes "safe state" or "go" and the red signal denotes "dangerous state" or "stop". Then the controller should show the red signal when the traffic control system fails, regardless of the actual situation on the road. If the failed traffic controller shows the green light, while the actual situation on the road is in the dangerous state, a fatal accident could occur.

In order to understand fail-safe circuit design requirements, some basic definitions and theorems are needed. These are listed below.

2.1. Preliminary Mathematical Definitions

Definition 1: A Boolean function f of n-variables is monotonic increasing if, and only if, $x \geq y$ implies $f(x) \geq f(y)$, where $x=(x_1,x_2, \ldots\ldots, x_n)$, $y=(y_1,y_2, \ldots\ldots, y_n)$, and $x \geq y$ means $x_i \geq y_i$ for $i=1,2,\ldots,n$. Similarly, a monotonic decreasing function is defined to be one for which $x \geq y$ implies $f(x) \leq f(y)$.

Definition 2: A Boolean function which is monotonic increasing with respect to some variables and monotonic decreasing with respect to the remaining variables is called a unate function. Thus, $f_1 = \bar{x}y+\bar{x}z$ is unate while $f_2 = xy+xz$ is monotonic increasing and $f_3 = \overline{xy}+\overline{xz}$ is monotonic decreasing.

Definition 3: If in a system the loss caused by a faulty 1 output is much greater than that caused by a faulty 0 output, then the system is said to be 0-fail-safe. In a similar manner, a 1-fail-safe system can be defined.

Definition 4: A logical component which, when it fails, always fails with a 0(or 1) output is said to be S_0-asymmetric (or S_1-asymmetric). Such components are called asymmetrical components.

Definition 5: A realisation of a sequential machine is said to be output - fail - safe, if, and only if, no failure can cause unsafe errors at the output terminals.

Definition 6: If the realisation of a machine, in addition to being output-fail-safe, goes on to a predetermined set F of states in the event of a failure, then it is said to be F-state fail-safe.

F-state fail-safe machines are more desirable from an error indication point of view since, in the event of a failure, the machine enters a known state F and therefore, an error-detecting circuit can be easily designed.

2.2 Criteria for State Assignment

In order to produce permissible state assignments for any type of fail-safe digital machine, certain basic theorems must be upheld. Formal proofs of these theorems may be found in the indicated references.

Theorem 1: A sequential circuit is output-fail-safe if, and only if, its state functions as well as its output functions are monotonic increasing with respect to the state variables, when all the logical components used in the realisation fail asymmetrically. [3]

The following theorem gives the necessary and sufficient condition for a state assignment to satisfy the requirements of theorem 1.

Theorem 2: The next state and the output functions of any sequential machine become monotonic with respect to the state variables if, and only if, the binary vectors used for the state assignment are not pairwise comparable, under the ordering relation of Definition 1. [2]

Theorem 3: A state assignment which uses a set of unordered code vectors will result in an F-state fail-safe sequential circuit if the state functions are all realised either in sum-of-products or in product-of-sums form. [3]

From the above three theorems, it can be seen that only certain code vectors can be chosen as the state assignments of a particular sequential machine, if the system is to become fail-safe. The vectors in any assignment must be pairwise incomparable; this simply means that a Hamming distance $\geqslant 2$ must exist between all the vectors.

The term "monotonic increasing" has been used frequently in this discussion. Since this is a rather abstract concept, it is worthwhile investigating further. Definition 1 gives a strict mathematical explanation; however, it is much simpler to deal in graphical terms.

2.3 Spatial Representation of a Boolean Quantity

Consider the diagram shown in Figure 2.1. Let X be an n-dimensional vector. Each of the n components can take two values 0 or 1, so that X has 2^n distinct possibilities. Now consider, in n-dimensional space, the points whose coordinate values are 0 or 1. To each possible X there corresponds one and only one of these points. The points form the vertices of a hypercube.

The concept of monotonic increasing functions may be depicted quite easily using the hypercube. Two examples of Boolean expressions are shown in Figure 2.2.

The black circles represent points which satisfy their respective functions; for example, in Figure 2.2(a), 101 satisfies $x_1+x_2x_3 = 1$ but 001 does not. Figure 2.2(a) is an example of a monotonic increasing function because as one moves up the hypercube (following the interconnecting lines from the point 000), once a function point has been reached, all the coordinate points above this point are all function points. Since, in Figure 2.2(b) the point 000 satisfies the function, this is not the case. Therefore, $x_1+\bar{x}_1\bar{x}_2\bar{x}_3 = 1$ is not a monotonic increasing function.

The hypercube can also be used to represent pairwise incomparable binary vectors as shown in Figure 2.3.. It can be seen that groups of pairwise incomparable vectors lie on the same level of a hypercube as shown by the dashed lines. Investigation of four-dimensional and five-dimensional hypercubes produced the same results. This led to the formulation of a theorem.

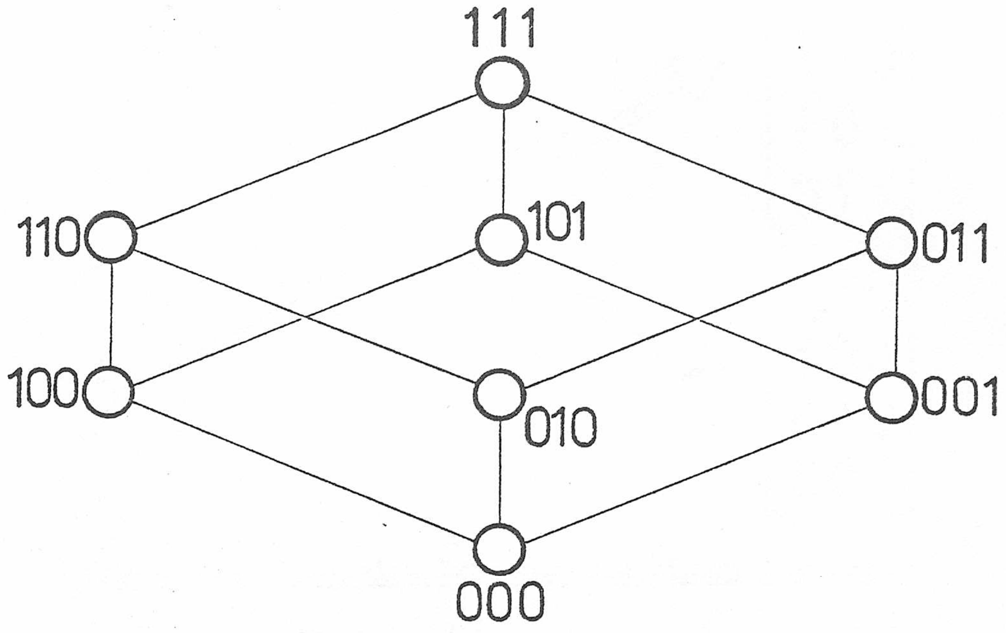


Figure 2.1 Hypercube

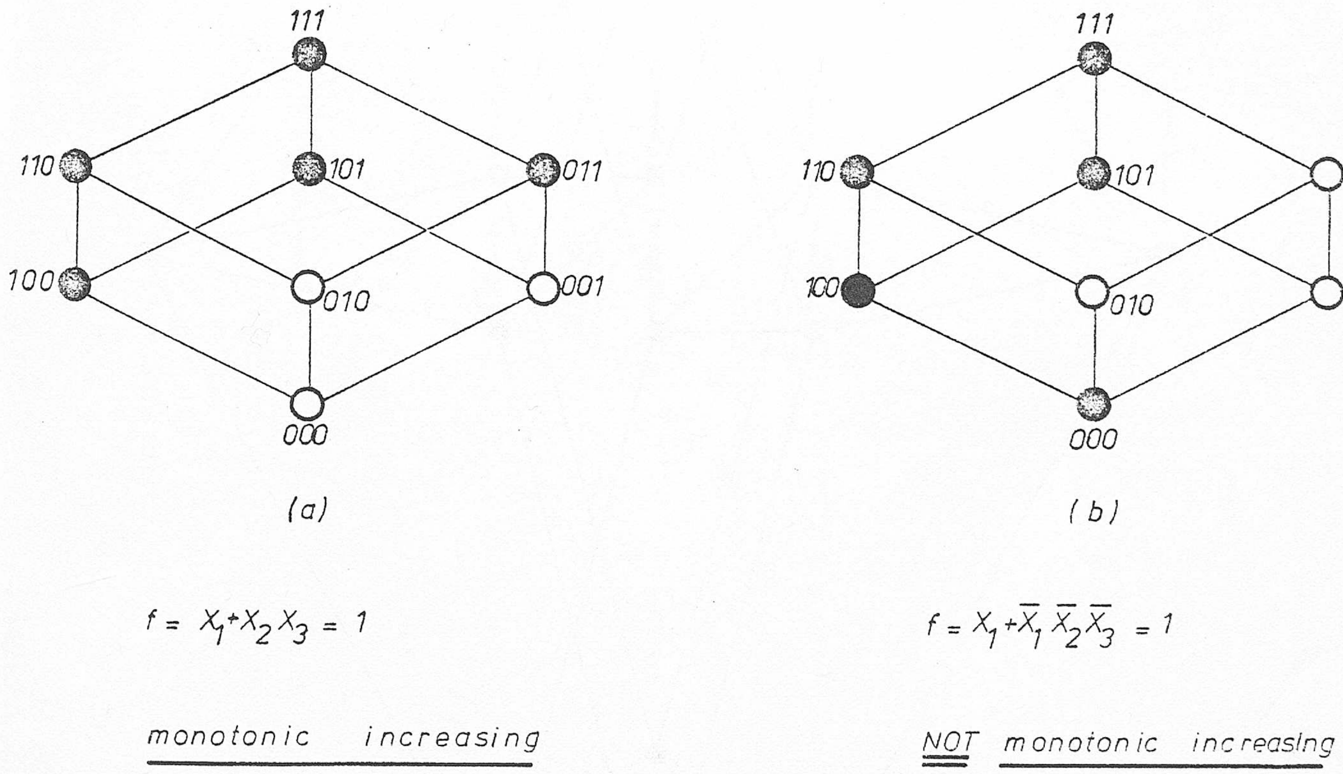
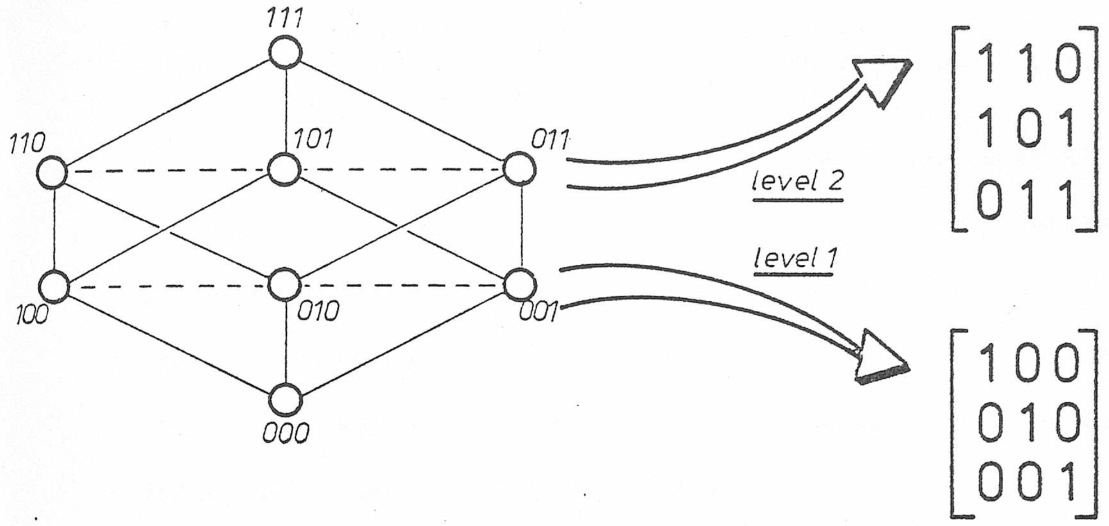


Figure 2.2 Black Circle Representation of Boolean Functions



$$\begin{bmatrix}1&1&0\\1&0&1\\0&1&1\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$$

Figure 2.3 Pairwise Incomparable Binary Vectors

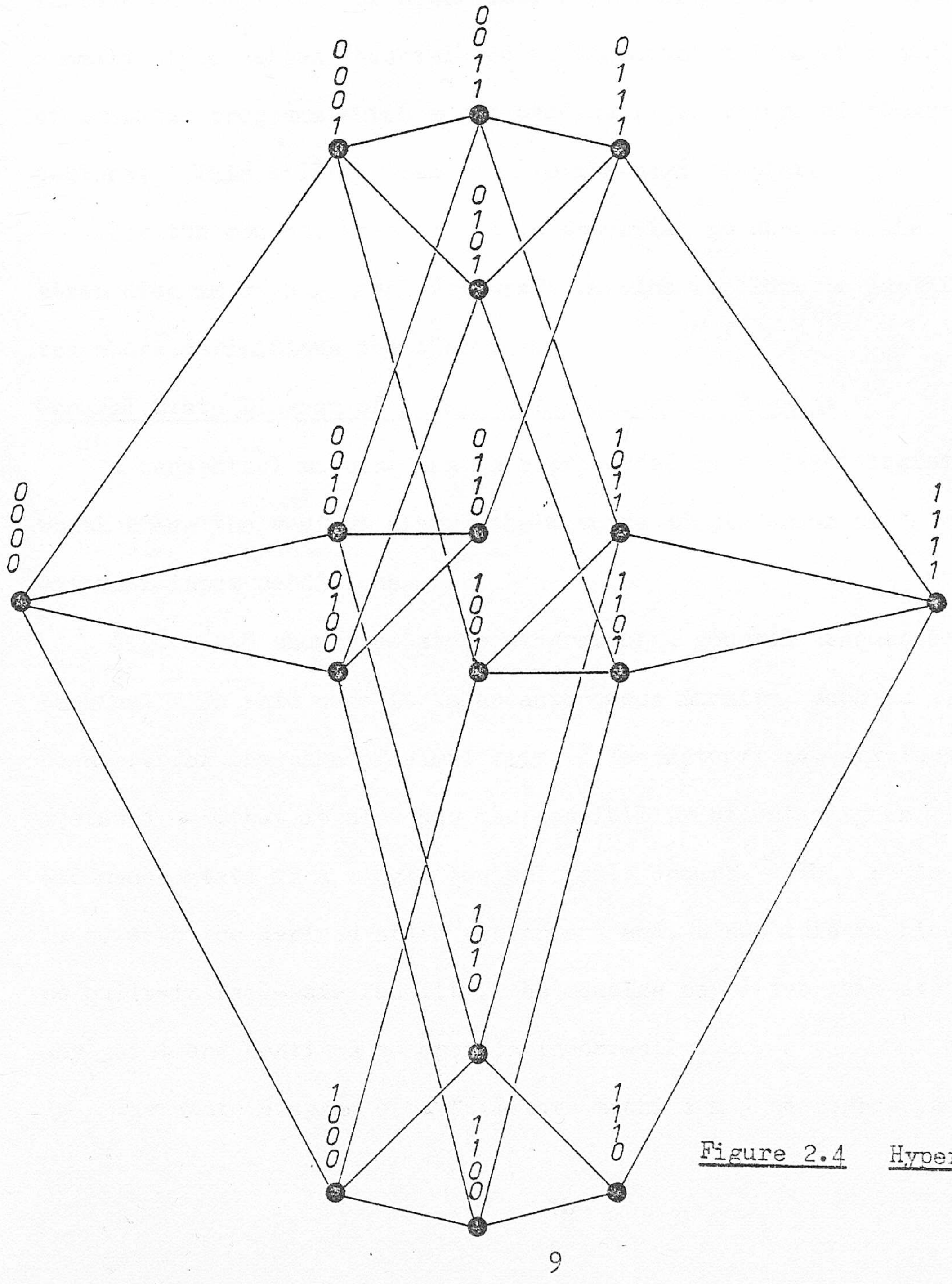


Figure 2.4 Hypercube

Theorem: The fact that a group of binary vectors lies on the same level of a hypercube implies that the vectors are pairwise incomparable with each other.

It would seen reasonable to assume, therefore, that all state assignments used in the design of fail-safe circuitry consist of binary vectors which lie on the same level of a hypercube. However, the theorem above is peculiar in the respect that its converse is not true. The example of Berger coding, which will be investigated later, will clearly illustrate this.

Figure 2.4 illustrates a four-dimensional hypercube. The increasing complexity of hypercubes beyond four variables and the anomaly of the above theorem, led to the construction of a series of computer programs which would produce legal groups of binary vectors. This will be discussed in the next chapter.

For the moment, however, it is essential to consider the state diagram of a general fail-safe machine in order to justify the above definitions and theorems.

2.4 General State Diagram of a Fail-Safe Sequential Machine

A sequential machine can be represented by a state diagram, which shows the various states the machine adopts under certain external-input conditions.

Figure 2.5 shows the state diagram of a general sequential machine. In this case it is an autonomous machine, such as a counter, for the sake of simplicity. The network has six legal states 1 → 6, but it also has the possibility of entering an erroneous state if a single logical fault occurs. This state is outwith the derived state assignment and, since this machine has no built-in fail-safe facility, the machine may leave this state at any point and continue to operate incorrectly.

The state diagram of a fail-safe machine may be represented

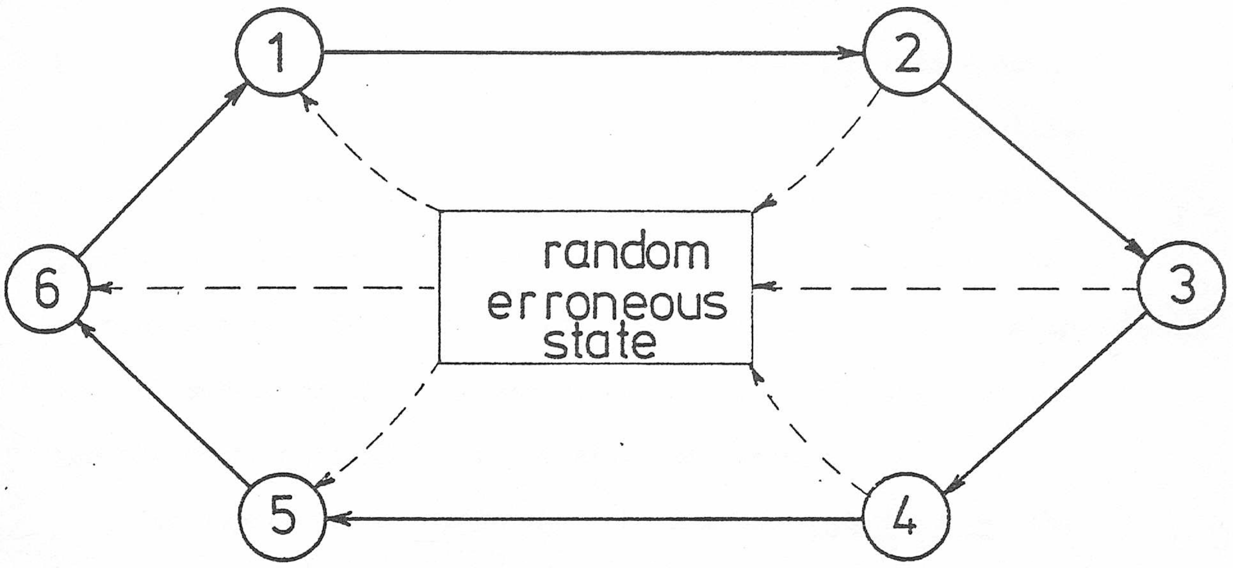


Figure 2.5 General State Diagram

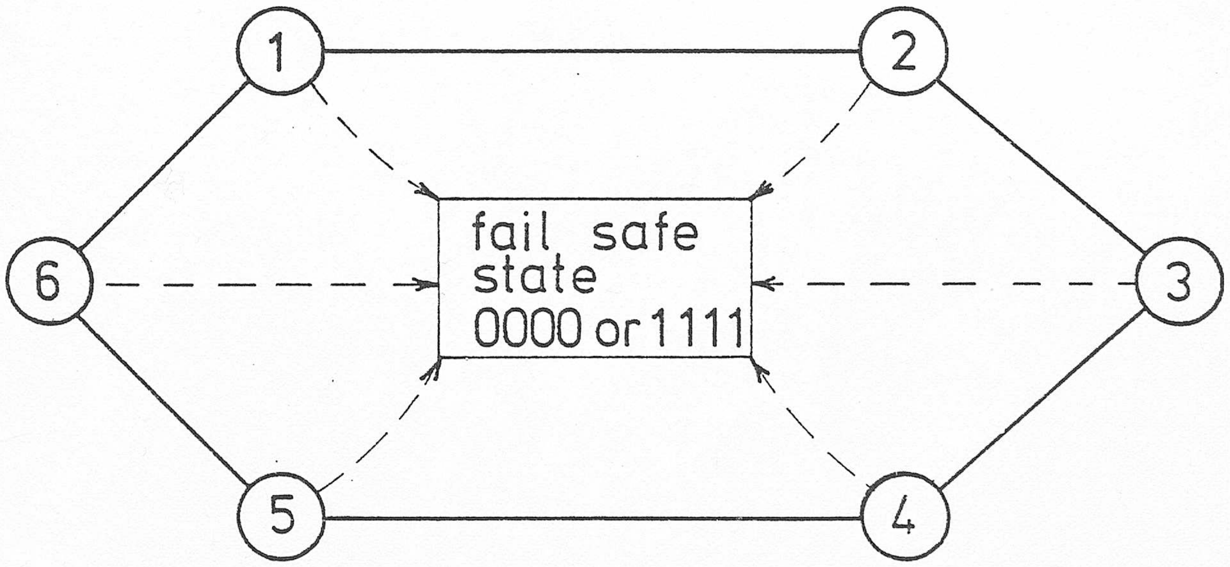


Figure 2.6 Fail-Safe State Diagram

in a similar fashion, as shown in Figure 2.6. This machine also has six legal states, but, of course, the individual states may be different, since they must satisfy the conditions laid down earlier. The error-free operation of this machine is identical to the machine of Figure 2.5, except that, instead of entering a purely random erroneous state when a logical fault occurs, the machine enters a fail-safe state, the F-state, which is usually the all-zeroes or all-ones state, depending on the type of fault and the combinational logic structure of the machine.

The important factor is that the machine cannot leave the F-state until the fault has been repaired, hence this system never functions incorrectly.

How this type of state diagram is implemented to produce a fail-safe digital machine will be dealt with in the next chapter.

CHAPTER 3. FAIL-SAFE DESIGN TECHNIQUES.

Several fail-safe design techniques are investigated in detail in this chapter. The first is a straight forward method using the transition table of the desired machine and is also outlined in reference [2]. The second is a technique which was constructed after thorough investigation of the Karnaugh Map minimisation method used in conventional digital machine design. The application of these techniques to the design of autonomous sequential circuits is also presented, along with the use and advantages of NAND synthesis in fail-safe design. Finally, computational methods of state assignment selection are outlined and presented in flow chart form along with selected program results.

3.1 Transition Table Technique

This technique is best illustrated by example. The three binary vectors

$$\begin{bmatrix} 011 \\ 101 \\ 110 \end{bmatrix}$$

are known to be pairwise incomparable (Figure 2.3), so these may be chosen as the state assignment of a simple synchronous, sequential machine. The actual design requirement was chosen to be a circuit which retains its state when an external 0 is applied and cycles continuously through the three states on the application of an external 1. The state transition table is shown in Figure 3.1.

D-type flip-flops are used as memory elements, the control data being derived from the standard characteristic equation. The control functions can be derived by any minimisation technique, or intuitively from the table. They are:

I/P	present state			next state			controls		
x	y_1	y_2	y_3	y_1	y_2	y_3	D_1	D_2	D_3
0	0	1	1	0	1	1	0	1	1
0	1	0	1	1	0	1	1	0	1
0	1	1	0	1	1	0	1	1	0
1	0	1	1	1	0	1	1	0	1
1	1	0	1	1	1	0	1	1	0
1	1	1	0	0	1	1	0	1	1

Figure 3.1 Conventional State Transition Table

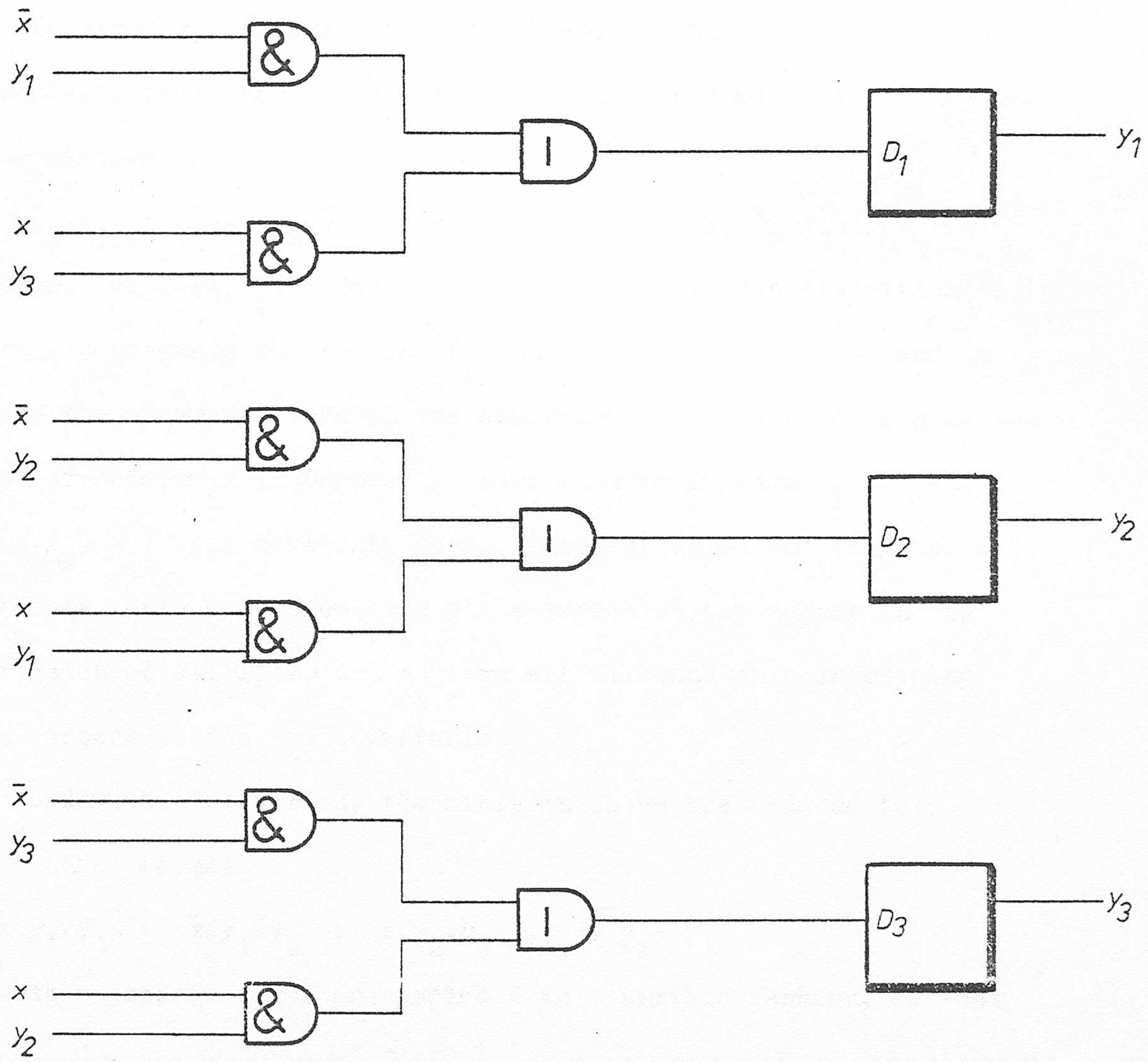


Figure 3.2 Circuit Representation

$$D_1 = \bar{x}.y_1 + x.y_3$$
$$D_2 = \bar{x}.y_2 + x.y_1$$
$$D_3 = \bar{x}.y_3 + x.y_2$$

The circuit is realised using AND and OR logic gates as illustrated in Figure 3.2. Note that, in practice, the complemented external input signals would be obtained via simple inverter gates, but are shown already complemented for the sake of clarity.

This is the conventional design approach for most types of sequential circuits. However, to produce a fail-safe version of the above design, a slightly different technique is employed.

Consider the column D_1 in the transition table of Figure 3.1. It contains four ones corresponding to the minterms of the function. These minterms are:

$\bar{x}.y_1.\bar{y}_2.y_3$; $\bar{x}.y_1.y_2.\bar{y}_3$; $x.\bar{y}_1.y_2.y_3$; $x.y_1.\bar{y}_2.y_3$

However, let $D_i(q,I_p)$ denote the control data in the transition table where q represents the row and I_p the column of the table, and $(y_1,\ldots,y_n)$ denote the binary vectors in the assignment. Now, according to the proof of Theorem 2 in Chapter 2, each minterm in each D_i for which $D_i\,(q,I_p) = 1$ will have only those state variables for which $y_i = 1$. Therefore, only uncomplemented state variables can appear in the expression of all D_i's, and so they are all monotonic increasing with respect to the state variables.

Under this criterion, the minterms above are reduced to the product terms:

$\bar{x}.y_1.y_3$; $\bar{x}.y_1.y_2$; $x.y_2.y_3$; $x.\ y_1.\ y_3$

The other product terms are produced in a similar fashion, so that the control functions of the now fail-safe circuit become:

$$D_1 = (y_1 \cdot y_3 + y_1 \cdot y_2)\bar{x} + (y_2 \cdot y_3 + y_1 \cdot y_3)\ x$$
$$D_2 = (y_2 \cdot y_3 + y_1 \cdot y_2)\bar{x} + (y_1 \cdot y_3 + y_1 \cdot y_2)\ x$$
$$D_3 = (y_2 \cdot y_3 + y_1 \cdot y_3)\bar{x} + (y_2 \cdot y_3 + y_1 \cdot y_2)\ x$$

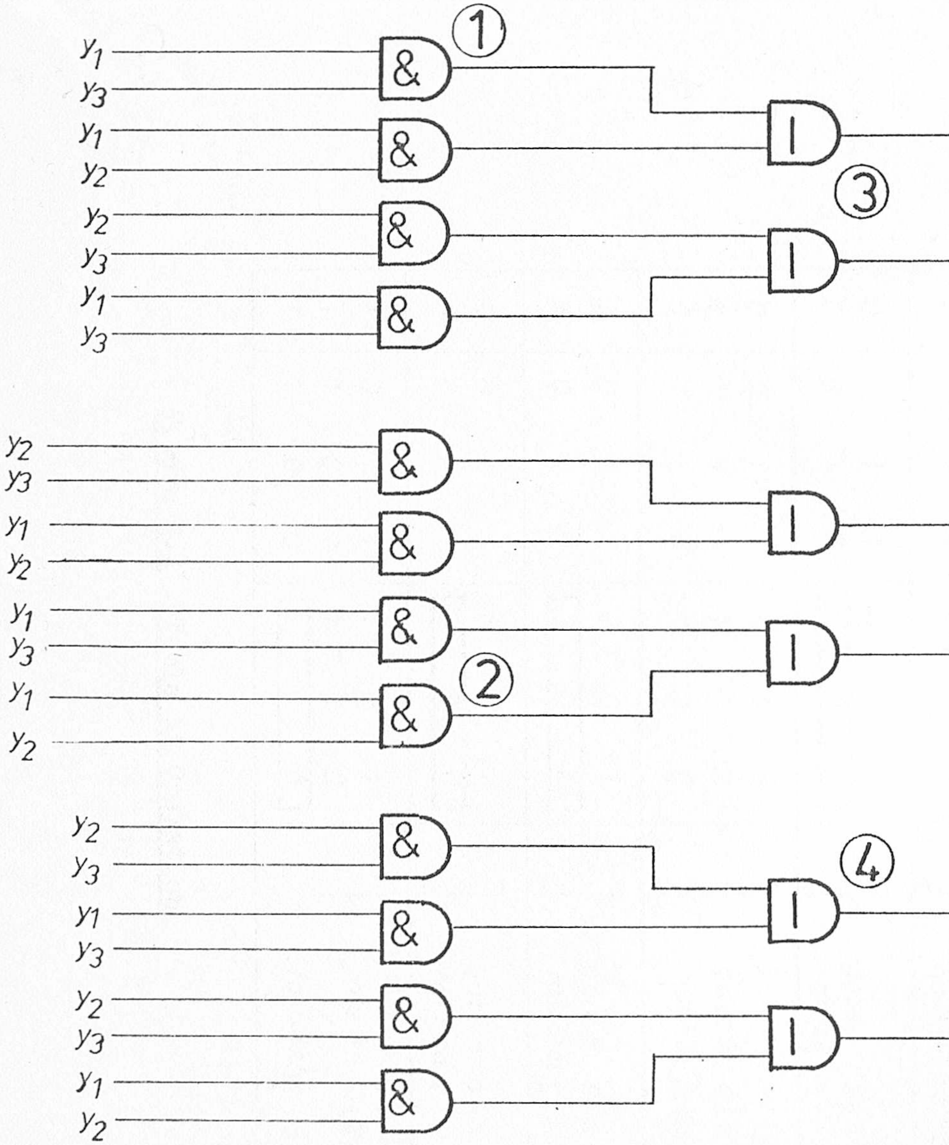
This circuit was constructed using AND and OR logic gates as shown in Figure 3.3. The circuit was tested for fail-safe operation and the results tabulated as illustrated in Figure 3.4.

Certain conclusions may be drawn from the operation table of this circuit.

(a) The most important point is the fact that fail-safe operation has been achieved, since the machine enters the F-state (000) when a single stuck-at-0 fault occurs.

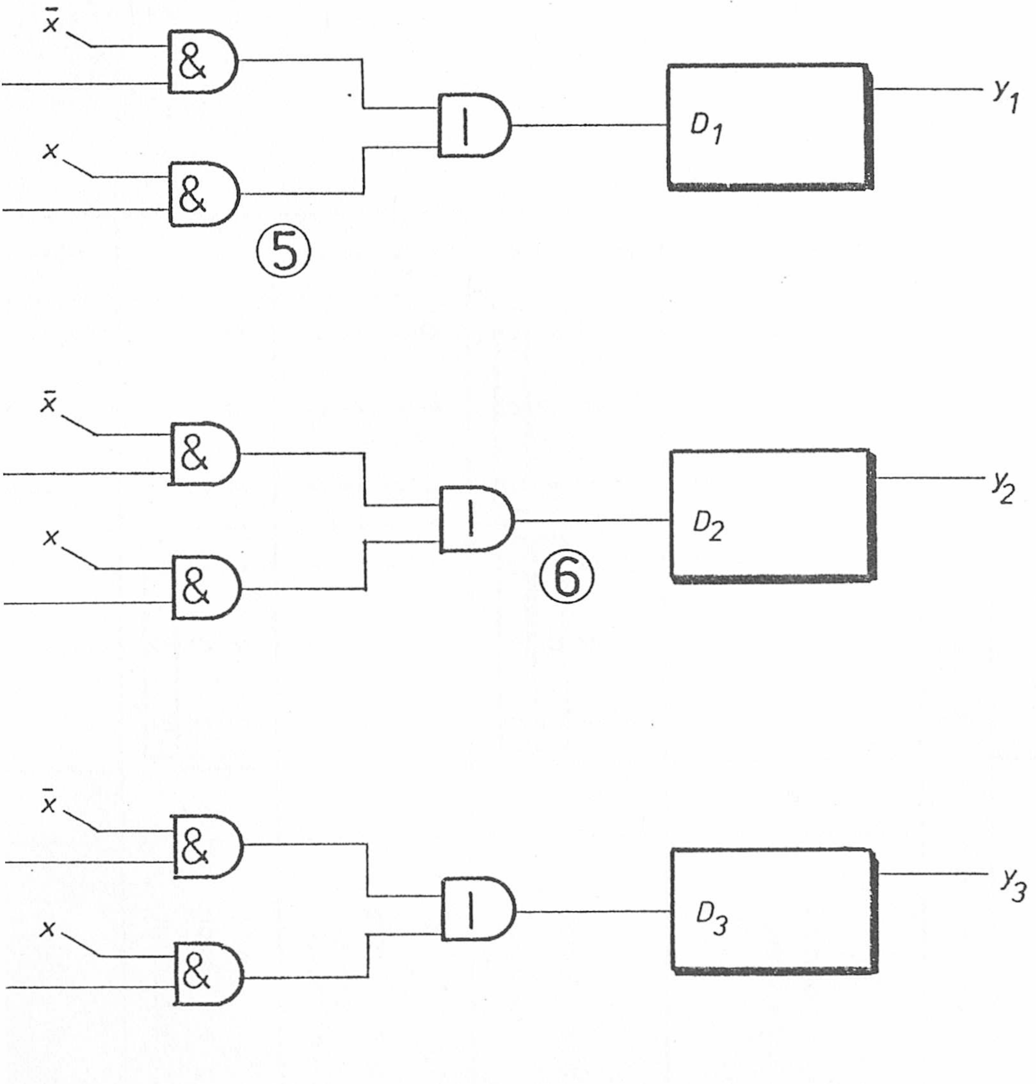
(b) However, the machine may continue to operate correctly under a stuck-at-0 fault, depending on whether or not the particular failed gate is required to generate a control function. This occurs because the circuit is constructed in such a way that a network of logic gates may be enabled or disenabled depending on the value of the external input X. For example, if gate 2 in Figure 3.3 has its output stuck-at-0, and X has the value 0, then gate 2 is never used, in this condition, to generate the control function D_2, therefore the circuit continues to operate correctly. Only when X becomes 1 will the circuit enter a fail-safe state.

(c) When this circuit was tested, all the logic gates were, in turn, given a simulated stuck-at-0 fault by disconnecting the output and grounding the input of the following gate. However, since the combinational



y_1
y_3
y_1
y_2
y_2
y_3
y_1
y_3
y_2
y_3
y_1
y_2
y_1
y_3
y_1
y_2
y_2
y_3
y_1
y_3
y_2
y_3
y_1
y_2
&
1
①
②
③
④

Figure 3.3



Circuit Realisation using Transition Table Technique

input	present state			next state			
x	y_1	y_2	y_3	Y_1	Y_2	Y_3	comments
0	0	1	1	0	1	1	normal operation
0	1	0	1	1	0	1	
0	1	1	0	1	1	0	
1	0	1	1	1	0	1	
1	1	0	1	1	1	0	
1	1	1	0	0	1	1	
0	0	1	1	0	1	1	gate 1 s-a-0 fail-safe
0	1	0	1	0	0	1	
0	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
0	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	0	1	1	1	0	1	gate 1 s-a-0 normal operation
1	1	0	1	1	1	0	
1	1	1	0	0	1	1	
1	0	1	1	1	0	1	
1	1	0	1	1	1	0	gate 2 s-a-0 fail-safe
1	1	1	0	0	0	1	
1	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	0	1	1	1	0	1	gate 3 s-a-0 normal operation
1	1	0	1	1	1	0	
1	1	1	0	0	1	1	
0	0	1	1	0	1	0	gate 4 s-a-0 fail-safe
0	0	1	0	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
0	0	1	1	0	1	1	gate 5 s-a-0 normal operation
0	1	0	1	1	0	1	
0	1	1	0	1	1	0	
1	0	1	1	0	0	1	gate 5 s-a-0 fail-safe
1	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
0	0	1	1	0	0	1	gate 6 s-a-0 fail-safe
0	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	0	1	1	1	0	1	gate 6 s-a-0 fail-safe
1	1	0	1	1	0	0	
1	1	0	0	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	

Figure 3.4 Table of Operation

logic networks feeding the flip-flops are identical, it is sufficient to test only a few selected gates as shown in Figure 3.3.

(d) To produce a fail-safe representation of Figure 3.2, a 200% increase in the number of gates used is required in this particular case.

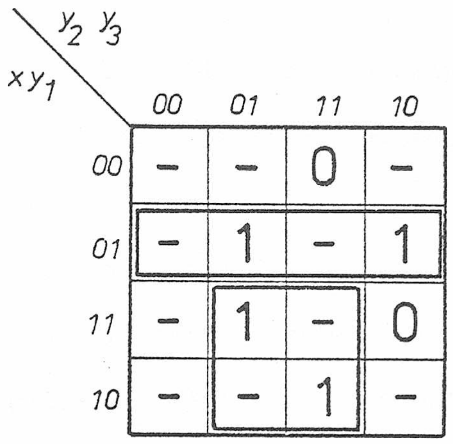
This transition table technique has proved satisfactory in the design of fail-safe sequential circuitry. The modified Karnaugh-Mapping technique will now be investigated in detail.

3.2. Modified Karnaugh-Mapping Technique

A Karnaugh-map is a graphical method of minimising Boolean functions.. The minterms (or maxterms) of a function are arranged in such a way that adjacent terms may be combined in groups of two, four, eight or sixteen, thereby reducing the number of final terms which describe the function [4] .

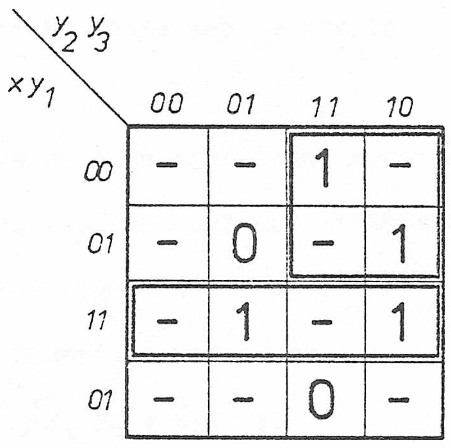
Figure 3.5 shows the general form of the Karnaugh-map applied to the transition table of Figure 3.1. The minterms and maxterms are entered in the appropriate place on the map, while the remainder of the positions are denoted by a slash, indicating a "don't-care" state. These permit a minimal set of control functions.

In a similar fashion, the Karnaugh-map may be utilised in a specific way to produce the control functions for a fail-safe sequential machine. In this case, however, only certain groupings of adjacent terms may be chosen to ensure that the functions obtained are all monotonic increasing. There is also a restriction on the number of "don't-care" states permitted in the Karnaugh-map itself. (As will be seen in Section 3.4, this restriction applies



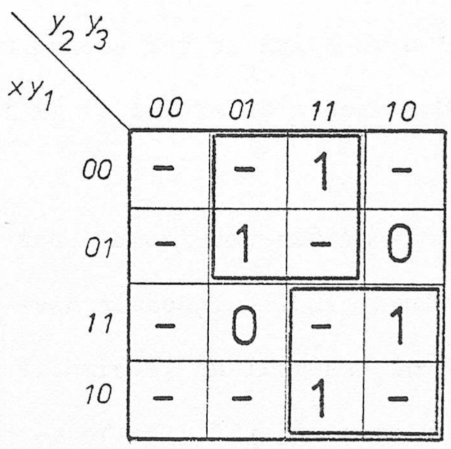
xy_1 \ y_2y_3	00	01	11	10
00	–	–	0	–
01	–	1	–	1
11	–	1	–	0
10	–	–	1	–

$$D_1 = \bar{x}y_1 + xy_3$$



xy_1 \ y_2y_3	00	01	11	10
00	–	–	1	–
01	–	0	–	1
11	–	1	–	1
01	–	–	0	–

$$D_2 = \bar{x}y_2 + xy_1$$



xy_1 \ y_2y_3	00	01	11	10
00	–	–	1	–
01	–	1	–	0
11	–	0	–	1
10	–	–	1	–

$$D_3 = \bar{x}y_3 + xy_2$$

Figure 3.5 Karnaugh-Map Minimisation

only to sequential circuits with external inputs, and not to autonomous circuits.) For a four-variable Karnaugh-map, the legal "don't care" states are:

(a) the all-ones position (1111)

(b) the X111 position, where X represents the external input with value 0. Note that the external input X may take up any position, (for example 1X11, 11X1, 111X).

These "don't-care" positions and the permitted term groupings for fail-safe design are illustrated in Figure 3.6. Note that two and three variable maps may be constructed in a similar manner.

As an example of this technique, consider Figure 3.7, which illustrates a modified Karnaugh-map minimisation of the original transition table of Figure 3.1.

The control functions now become:

$$D_1 = y_1 \cdot y_3 + \bar{x} \cdot y_1 \cdot y_2 + x \cdot y_2 \cdot y_3$$
$$D_2 = y_1 \cdot y_2 + \bar{x} \cdot y_2 \cdot y_3 + x \cdot y_1 \cdot y_3$$
$$D_3 = y_2 \cdot y_3 + \bar{x} \cdot y_1 \cdot y_3 + x \cdot y_1 \cdot y_2$$

The circuit is shown in Figure 3.8 while a table of operation under logic failures is illustrated in Figure 3.9.

Certain conclusions may be drawn from this table.

(a) The circuit is fail-safe to stuck-at-0 logic faults.

(b) As before, the circuit may continue to operate correctly, even although a logic fault has occurred.

(c) Using this technique, an increase in the number of logic gates of only 133.3% is required, compared with 200% for the previous technique.

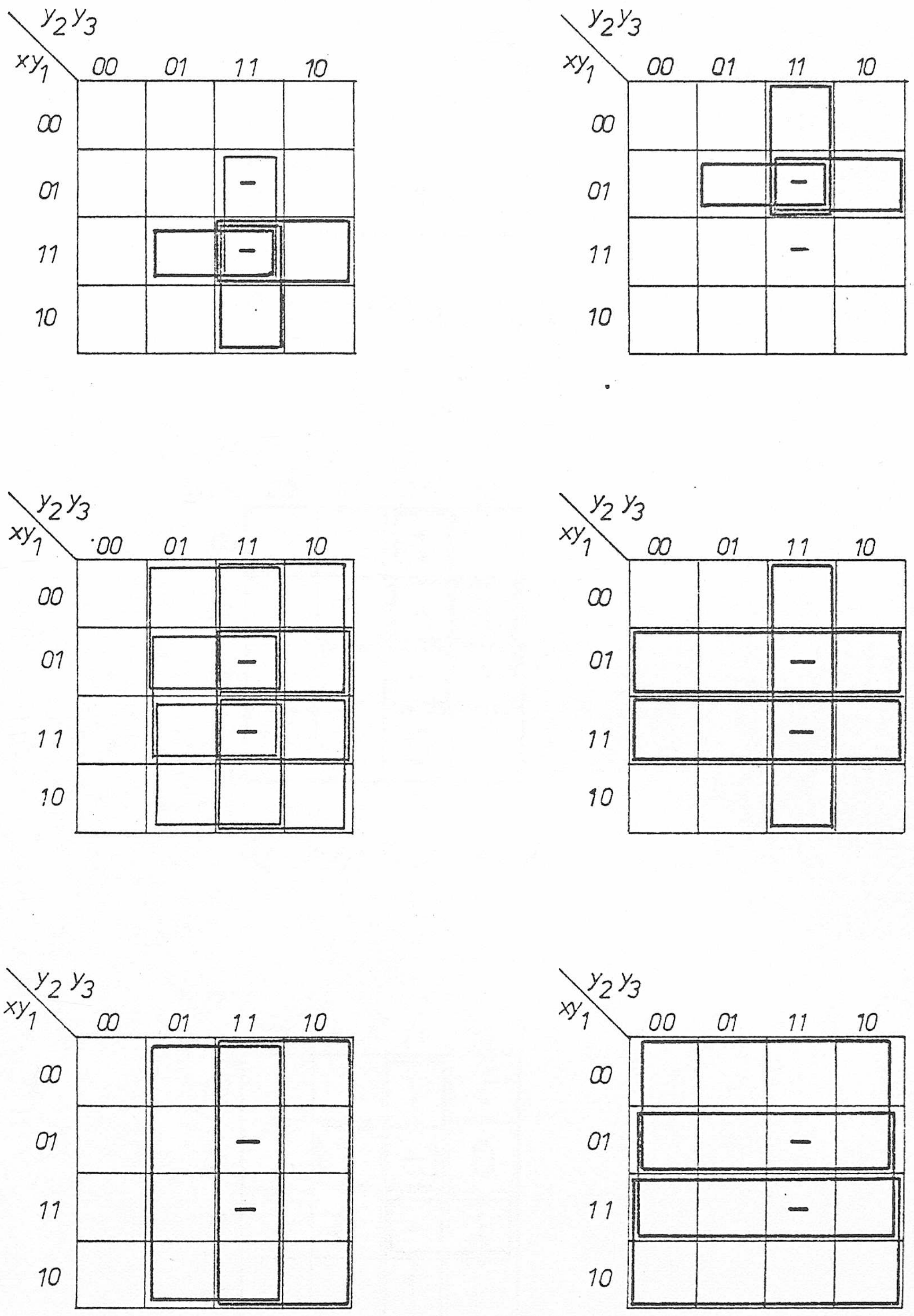
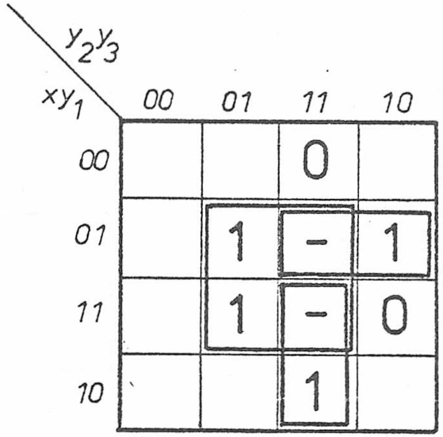
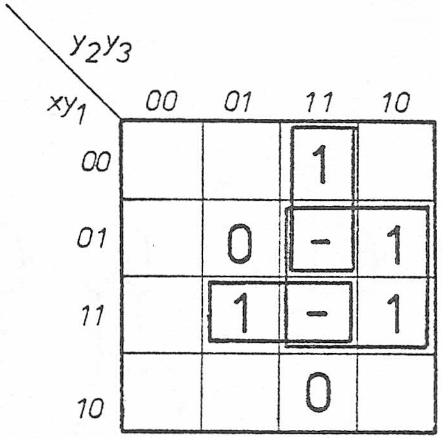


Figure 3.6 Term Groupings for Fail-Safe Design



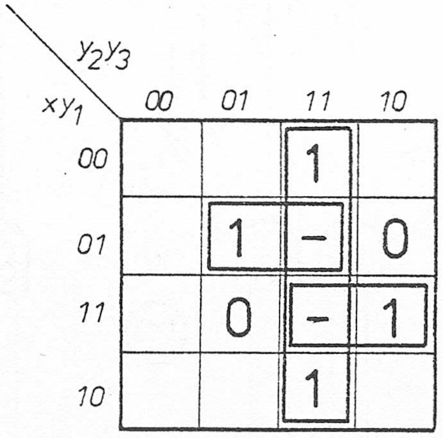
xy_1 \ y_2y_3	00	01	11	10
00			0	
01		1	–	1
11		1	–	0
10			1	

$$D_1 = y_1y_3 + \overline{x}y_1y_2 + xy_2y_3$$



xy_1 \ y_2y_3	00	01	11	10
00			1	
01		0	–	1
11		1	–	1
10			0	

$D_2 = y_1 y_2 + \bar{x} y_2 y_3 + x y_1 y_3$



$D_3 = y_2 y_3 + \bar{x} y_1 y_3 + x y_1 y_2$

Figure 3.7 Modified Karnaugh-Map

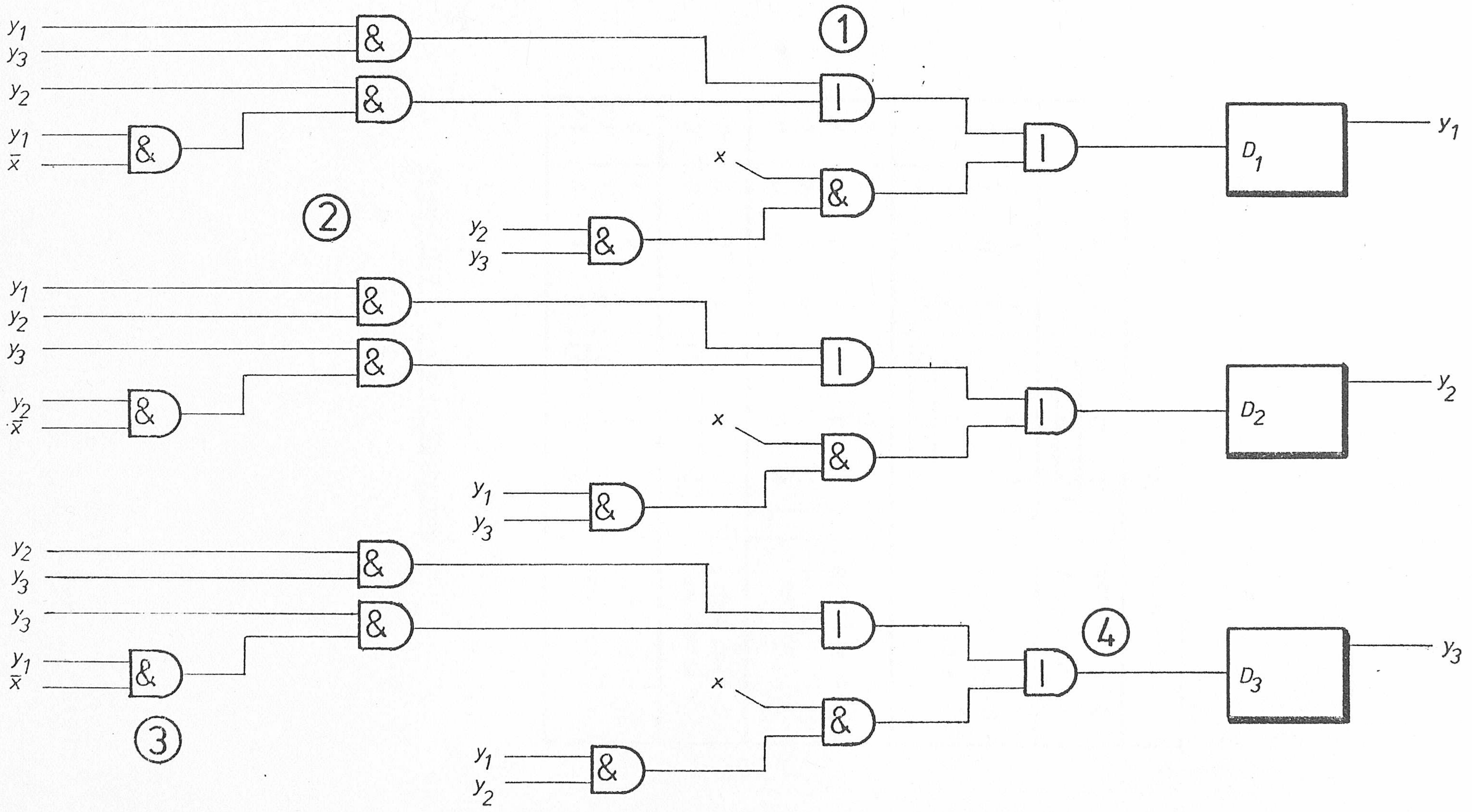


Figure 3.8 Fail-Safe Representation using Karnaugh-Map Technique

input	present state			next state			
x	y_1	y_2	y_3	y_1	y_2	y_3	comments
0	1	0	1	0	0	1	gate 1 s-a-0 fail - safe
0	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	0	1	1	1	0	1	gate 2 s-a-0 fail - safe
1	1	0	1	1	1	0	
1	1	1	0	0	0	1	
1	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	0	1	1	1	0	1	gate3 s-a-0 normal operation
1	1	0	1	1	1	0	
1	1	1	0	0	1	1	
1	0	1	1	1	0	1	
0	0	1	1	0	1	0	gate 4 s-a-0 fail - safe
0	0	1	0	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	1	0	1	1	1	0	gate 4 s-a-0 fail-safe
1	1	1	0	0	1	0	
1	0	1	0	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	

Figure 3.9 Table of Operation

The main advantage of this modified Kernaugh-map technique over the transition table method is the fact that, in most cases, fewer logic gates are needed to produce a fail-safe version of a particular sequential machine.

The use of NAND synthesis in fail-safe design will now be discussed.

3.3 The Use of NAND Synthesis

This is a technique whereby circuits, using conventional AND and OR logic gates, may be translated into equivalent circuits using only NAND gates [5]. This is probably of greatest importance in the manufacture of microcircuits using large scale integration (L.S.I.).

This technique is illustrated purely by example.

The circuit shown in Figure 3.8 may be translated quite easily into a NAND logic circuit as shown in Figure 3.10. The most important point to note in this procedure is the fact that 3-input NAND gates are readily available in TTL package form whereas AND and OR gates are usually produced only in 2-input versions. Therefore the two-level AND and OR networks shown in Figure 3.8 may be replaced by a one-level NAND gate as shown in Figure 3.10. The table of operation shown in Figure 3.11 illustrates that:

(a) the circuit may continue to operate correctly under a stuck-at-0 fault.

(b) the circuit is fail-safe to stuck-at-0 logic faults. However, it is clearly seen that sometimes the F-state is 000 and sometimes it is 111. This was found to be characteristic of

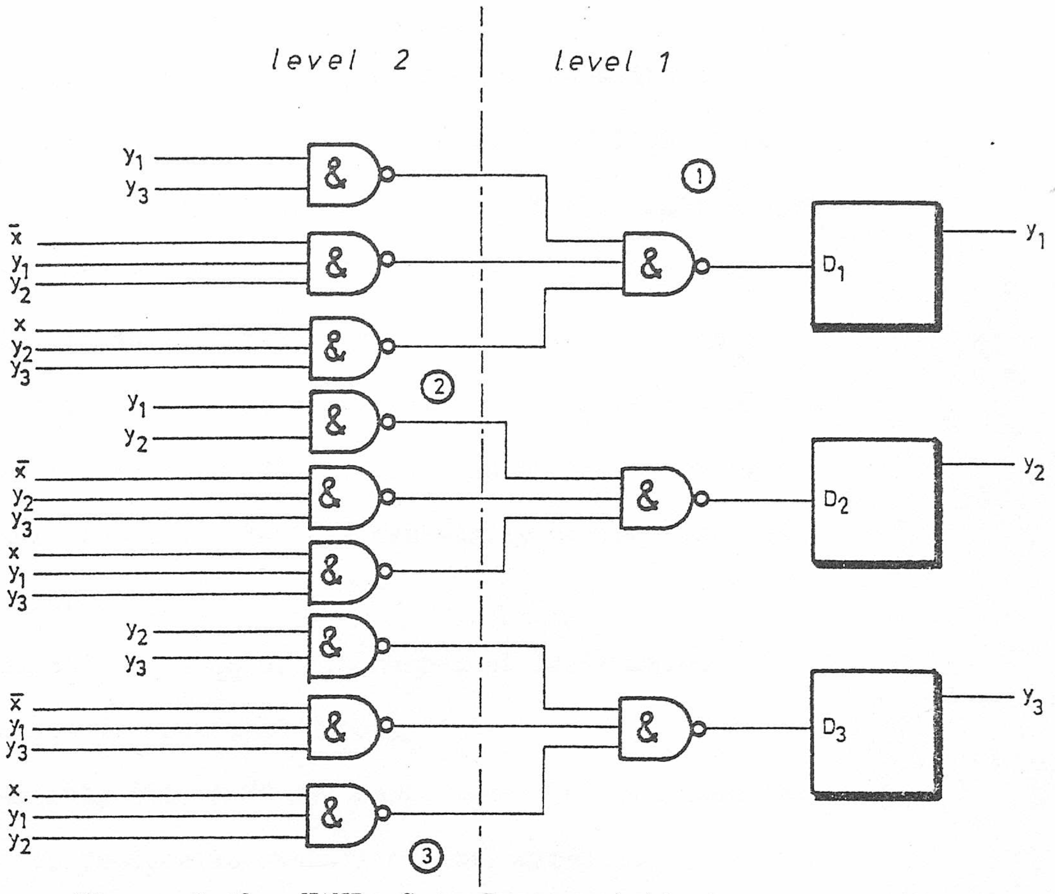


Figure 3.10 NAND Gate Representation

Figure 3.11 Table of Operation

input	present state			next state			
x	y_1	y_2	y_3	Y_1	Y_2	Y_3	comments
0	1	0	1	0	0	1	gate 1 s-a-0 fail-safe
0	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
1	1	1	0	0	1	1	gate 1 s-a-0 fail-safe
1	0	1	1	0	0	1	
1	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{0}$	
0	1	1	0	1	1	0	gate2 s-a-0 normal operation
0	1	1	0	1	1	0	
0	1	0	1	1	1	1	gate 2 s-a-0 fail-safe
0	1	1	1	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	
1	1	1	0	0	1	1	gate2 s-a-0 fail-safe
1	0	1	1	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	
1	0	1	1	1	0	1	gate3 s-a-0 fail-safe
1	1	0	1	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	

fail-safe circuits constructed using NAND logic; if a stuck-at-0 fault occurs in an "odd" level (see Figure 3.10), the F-state is 000; if the fault occurs in an "even" level, the F-state is 111. This is obviously an advantage when repairing a fault in a more complex circuit since attention can be restricted to either even or odd levels. An error-detecting circuit for a 111 F-state can easily be incorporated as before.

(c) using this technique, the number of logic gates required is increased by only 33%.

In general, this NAND synthesis technique can usually be applied to produce an overall minimal circuit.

The design of autonomous sequential circuits using the Karnaugh-map method will now be discussed.

3.4 Autonomous Circuit Design

An autonomous circuit is a sequential machine with no externally-applied input signals. The circuit is controlled solely by the sequence of clock pulses applied to the memory elements in the circuit. Fail-safe autonomous circuitry can be designed using the transition table technique, exactly as before. However, the Karnaugh-map method is slightly different when applied to autonomous circuits, since there is no restriction on the number of "don't-care" states appearing in the Karnaugh-map; however, there is a restriction on the number of term groupings, since all the variables in the Karnaugh-map are state variables. (Beforehand, the map contained the input variable x, which is not classed as a state variable and may appear in complemented form in the final control functions, without violating the condition that all the functions must be monotonic increasing).

The legal term groupings are illustrated in Figure 3.12.

An example will demonstrate this technique. Figure 3.13 illustrates the transition table, conventional control functions and circuit diagram of an autonomous sequential machine which simply cycles through four states continuously. Note that, for the purpose of comparison, the state assignment vectors are pairwise incomparable with each other. However, when the modified Karnaugh-map minimisation technique is used, as shown in Figure 3.14, a fail-safe circuit is obtained. This is clearly shown by the table of operation illustrated in Figure 3.15.

This technique is quite a simple one to use. Indeed, beginning with a known state assignment, any of the above design procedures is a relatively straightforward task. However, problems can arise when choosing a particular state assignment. The state vectors must, first of all, satisfy the original design requirement; but they must also satisfy a specific ordering relation, as seen earlier. More often than not, these requirements are contradictory, and producing a state assignment which satisfies both conditions can be a very tedious task indeed. Therefore, a computer program was constructed to alleviate this problem.

3.5 Computational Methods of State Assignment Selection

As mentioned earler in Section 2.3, it seemed reasonable to assume that all feasible groups of pairwise incomparable binary vectors lie on the same level of a hypercube, whether this cube be of 3, 4 or even 10 variables.

Therefore, a computer program was written in BASIC to compute these groups of vectors. The flowchart is shown in Figure 3.16, while the actual program is illustrated in Appendix 1.

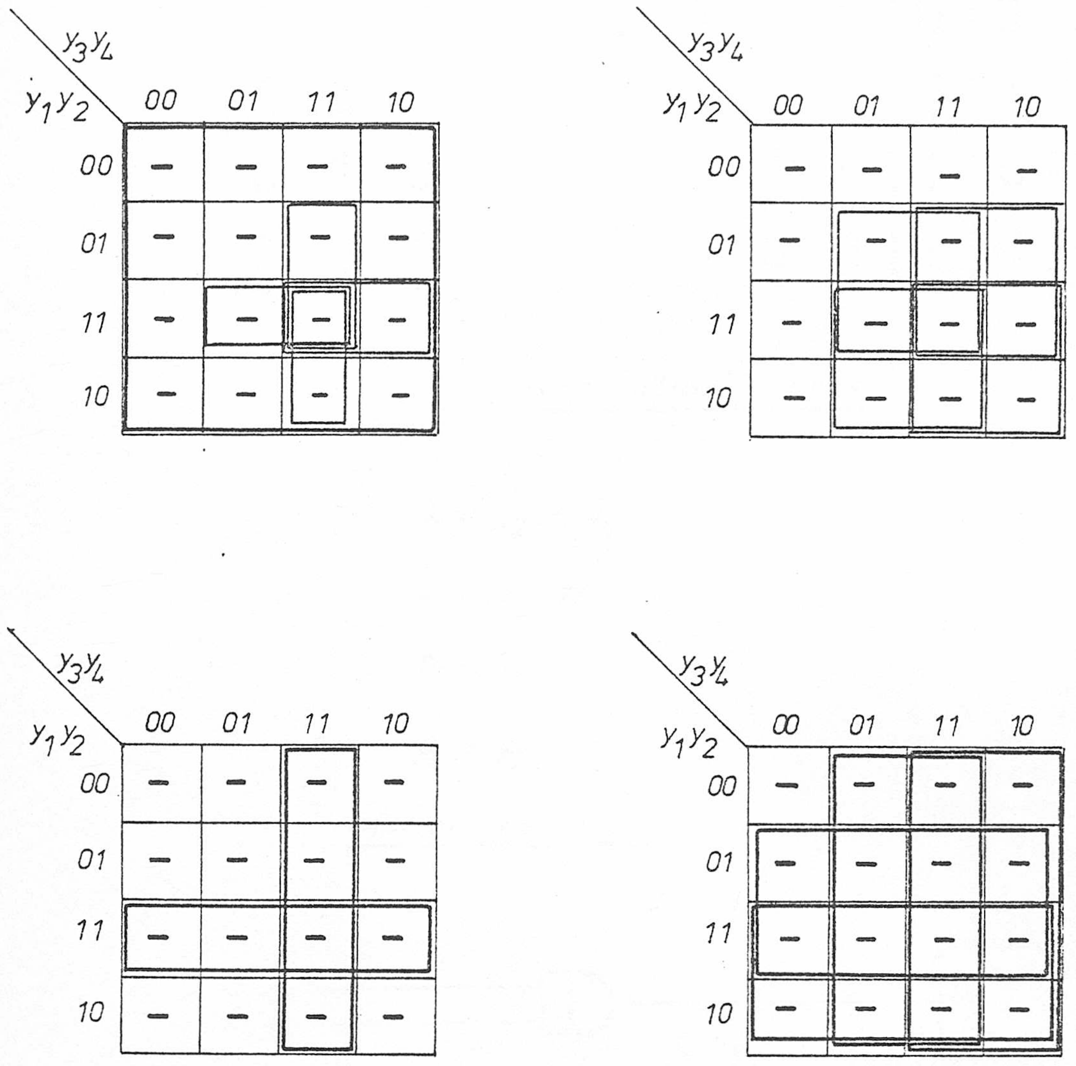


Figure 3.12 Term Groupings for Autonomous Design

present state				next state				controls			
y_1	y_2	y_3	y_4	Y_1	Y_2	Y_3	Y_4	D_1	D_2	D_3	D_4
1	0	0	1	0	0	1	1	0	0	1	1
0	0	1	1	0	1	0	1	0	1	0	1
0	1	0	1	1	0	1	0	1	0	1	0
1	0	1	0	1	0	0	1	1	0	0	1

<u>transition table</u>

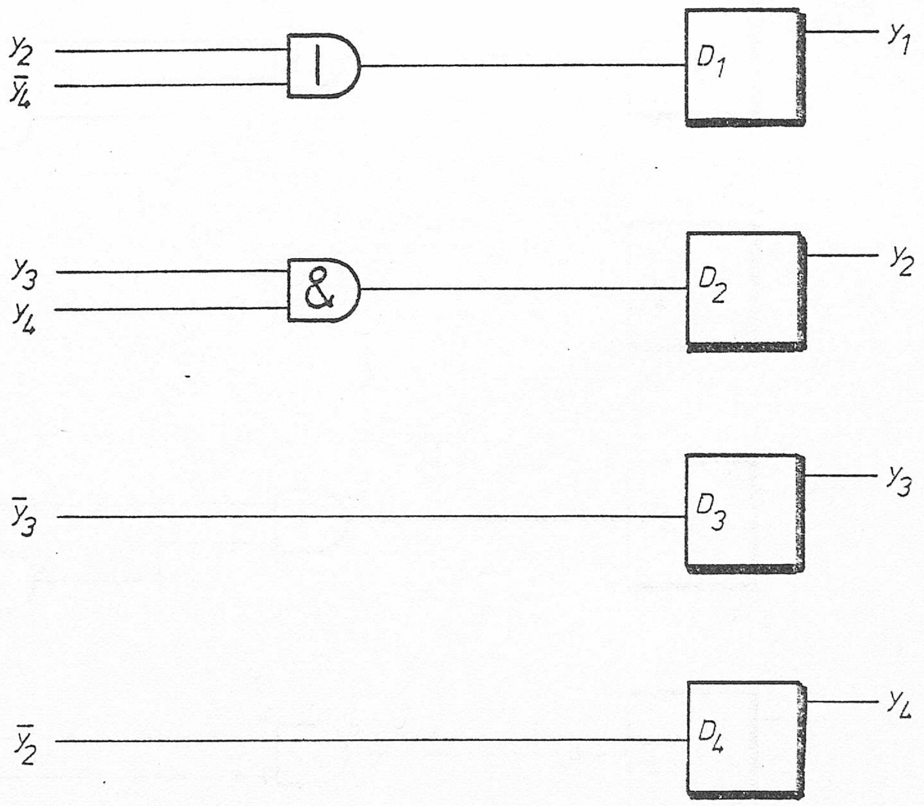
$$D_1 = y_2 + \bar{y}_4$$

$$D_2 = y_3 \cdot y_4$$

$$D_3 = \bar{y}_3$$

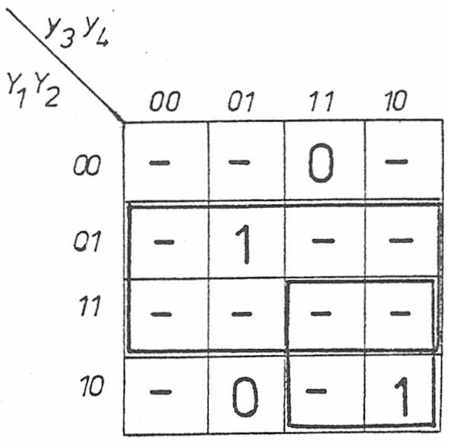
$$D_4 = \bar{y}_2$$

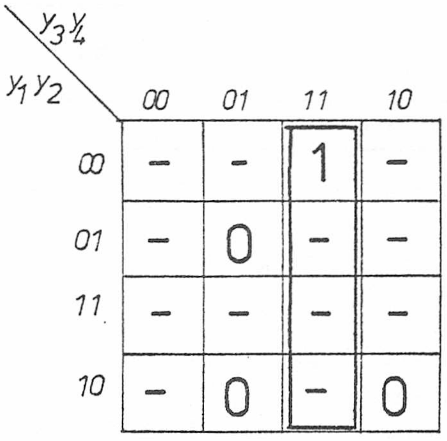
<u>control equations</u>



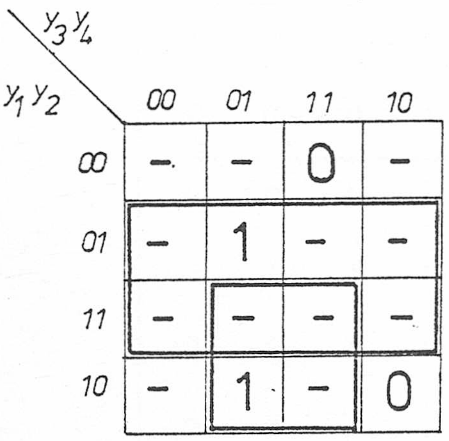
<u>circuit</u>

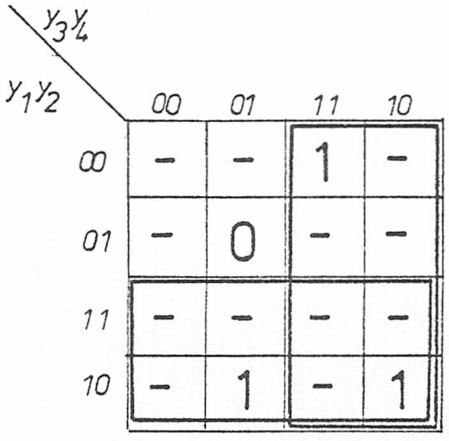
Figure 3.13 Conventional Autonomous Design





$D_1 = y_2 + y_1 y_3$ $D_2 = y_3 y_4$





$D_3 = y_2 + y_1 y_4$ $D_4 = y_1 + y_3$

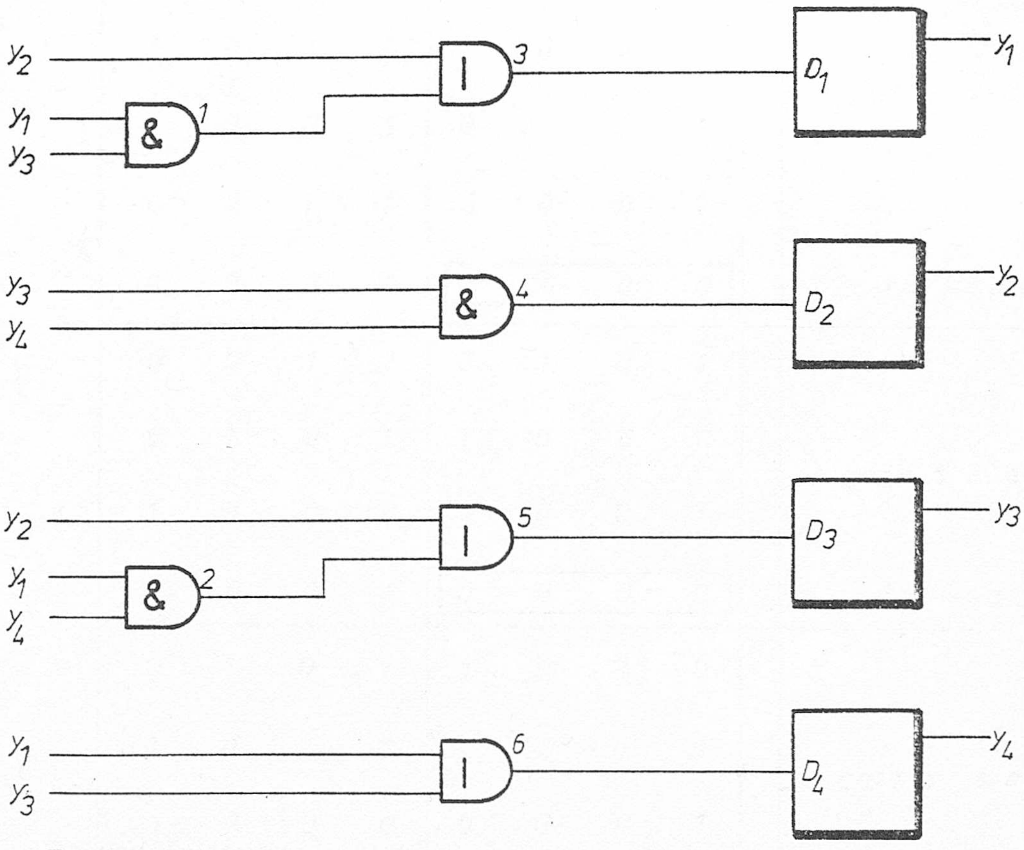


Figure 3.14 Fail-Safe Autonomous Design

Figure 3.15 Operation Table

present state				next state				
y_1	y_2	y_3	y_4	Y_1	Y_2	Y_3	Y_4	comments
1	0	0	1	0	0	1	1	
0	0	1	1	0	1	0	1	
0	1	0	1	1	0	1	0	gate 1 s-a-0
1	0	1	0	0	0	0	1	
0	0	0	1	0	0	0	0	fail-safe
1	0	0	1	0	0	0	1	gate 2 s-a-0
0	0	0	1	0	0	0	0	fail-safe
0	0	1	1	0	1	0	1	
0	1	0	1	0	0	1	0	gate 3 s-a-0
0	0	1	0	0	0	0	1	
0	0	0	1	0	0	0	0	fail-safe
0	1	0	1	1	0	1	0	
1	0	1	0	1	0	0	1	gate 4 s-a-0
1	0	0	1	0	0	1	1	
0	0	1	1	0	0	0	1	
0	0	0	1	0	0	0	0	fail-safe
0	0	1	1	0	1	0	1	
0	1	0	1	1	0	0	0	gate 5 s-a-0
1	0	0	0	0	0	0	1	
0	0	0	1	0	0	0	0	fail-safe
0	1	0	1	1	0	1	0	
1	0	1	0	1	0	0	0	gate 6 s-a-0
1	0	0	0	0	0	0	1	
0	0	0	1	0	0	0	0	fail-safe

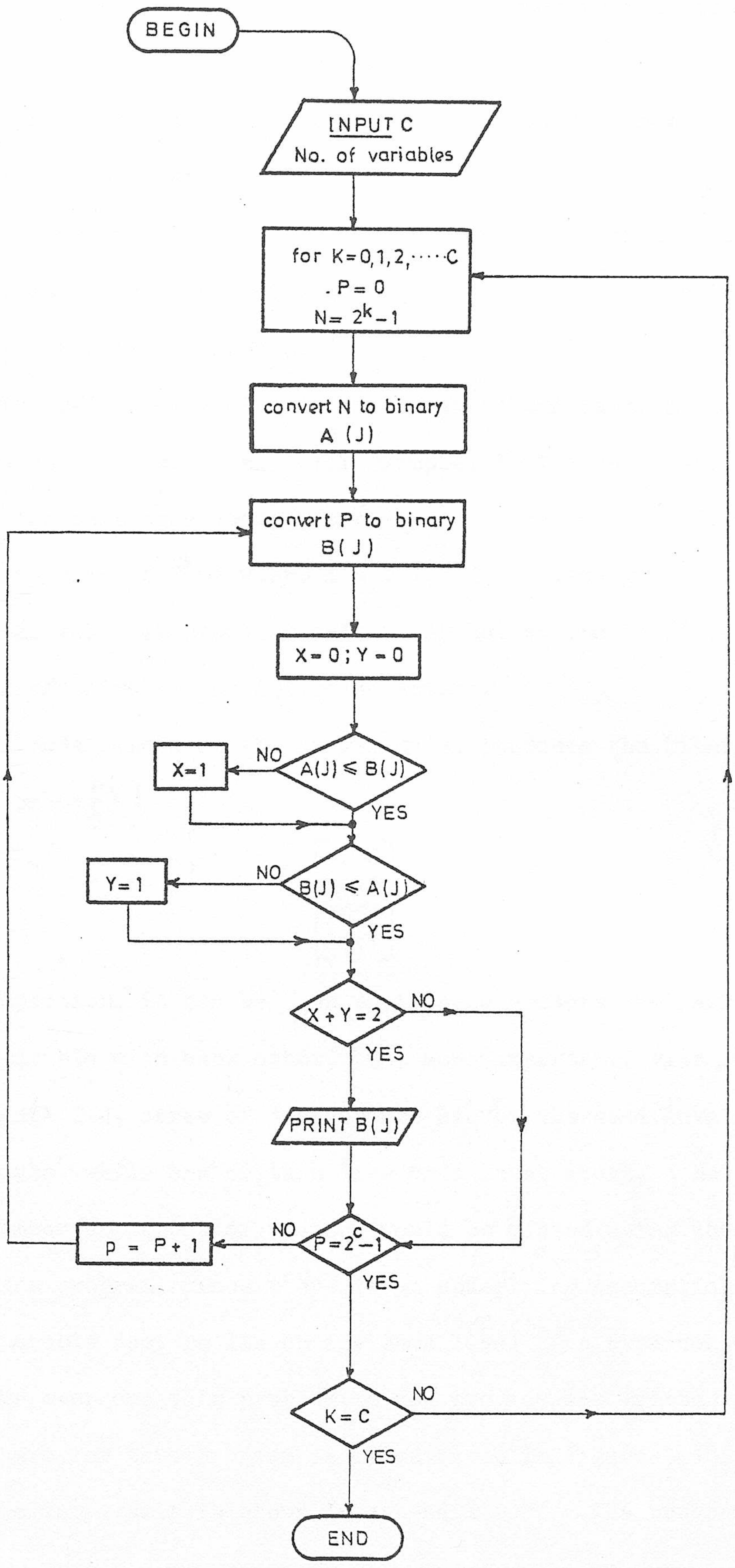


Figure 3.16 Flowchart

The results for 3, 4 and 5 variables are shown in Figure 3.17, the vector groups asterisked being those used in the earlier sections on fail-safe design.

The operation of the program is straightforward:

A reference binary vector is chosen from each level of a hypercube and all other vectors compared with them. If a vector is found to be pairwise incomparable with the reference vector, then it is printed.

The reference vector is chosen to be the least decimal equivalent in each level; for example, in the 3-variable case, the reference vectors correspond to decimal 0,1,3 and 7 a progression of 2^k-1 where $k = 0,1,2,3$. Although this program performs well, it produces only a sub-set of the total possible groups of pairwise incomparable vectors.

A code, known as the Berger code, produces the binary vectors shown below[3]

$$\begin{bmatrix} 1100 \\ 1010 \\ 1001 \\ 0111 \end{bmatrix}$$

By inspection, it can be seen that these vectors are pairwise incomparable with each other, but, more important, with reference to Figure 2.4, three of the vectors lie on the same level of a hypercube, while one of them lies on a level above. However, this important group of vectors would be missed using the above computer program, since there is an underlying assumption that incomparable vectors lie on the same level of a hypercube.

To overcome this problem, a new program was written. The flowchart for this program is illustrated in Figure 3.18, while the program itself is shown in Appendix 2. The program is

```
                PAIRWISE INCOMPARABLE BINARY NUMBERS

HOW MANY VARIABLES? 3

Ø   Ø   Ø

Ø   Ø   1
Ø   1   Ø
1   Ø   Ø

Ø   1   1
1   Ø   1   *
1   1   Ø

1   1   1


HOW MANY VARIABLES? 4

Ø   Ø   Ø   Ø

Ø   Ø   Ø   1
Ø   Ø   1   Ø
Ø   1   Ø   Ø
1   Ø   Ø   Ø

Ø   Ø   1   1
Ø   1   Ø   1
Ø   1   1   Ø   *
1   Ø   Ø   1
1   Ø   1   Ø
1   1   Ø   Ø

Ø   1   1   1
1   Ø   1   1
1   1   Ø   1
1   1   1   Ø

1   1   1   1


HOW MANY VARIABLES? 5

Ø   Ø   Ø   Ø   Ø

Ø   Ø   Ø   Ø   1
Ø   Ø   Ø   1   Ø
Ø   Ø   1   Ø   Ø
Ø   1   Ø   Ø   Ø
1   Ø   Ø   Ø   Ø

Ø   Ø   Ø   1   1
Ø   Ø   1   Ø   1
Ø   Ø   1   1   Ø
Ø   1   Ø   Ø   1
Ø   1   Ø   1   Ø
Ø   1   1   Ø   Ø
1   Ø   Ø   Ø   1
1   Ø   Ø   1   Ø
1   Ø   1   Ø   Ø
1   1   Ø   Ø   Ø

Ø   Ø   1   1   1
Ø   1   Ø   1   1
Ø   1   1   Ø   1
Ø   1   1   1   Ø
1   Ø   Ø   1   1
1   Ø   1   Ø   1
1   Ø   1   1   Ø
1   1   Ø   Ø   1
1   1   Ø   1   Ø
1   1   1   Ø   Ø

Ø   1   1   1   1
1   Ø   1   1   1
1   1   Ø   1   1
1   1   1   Ø   1
1   1   1   1   Ø

1   1   1   1   1
```

Figure 3.17 Results

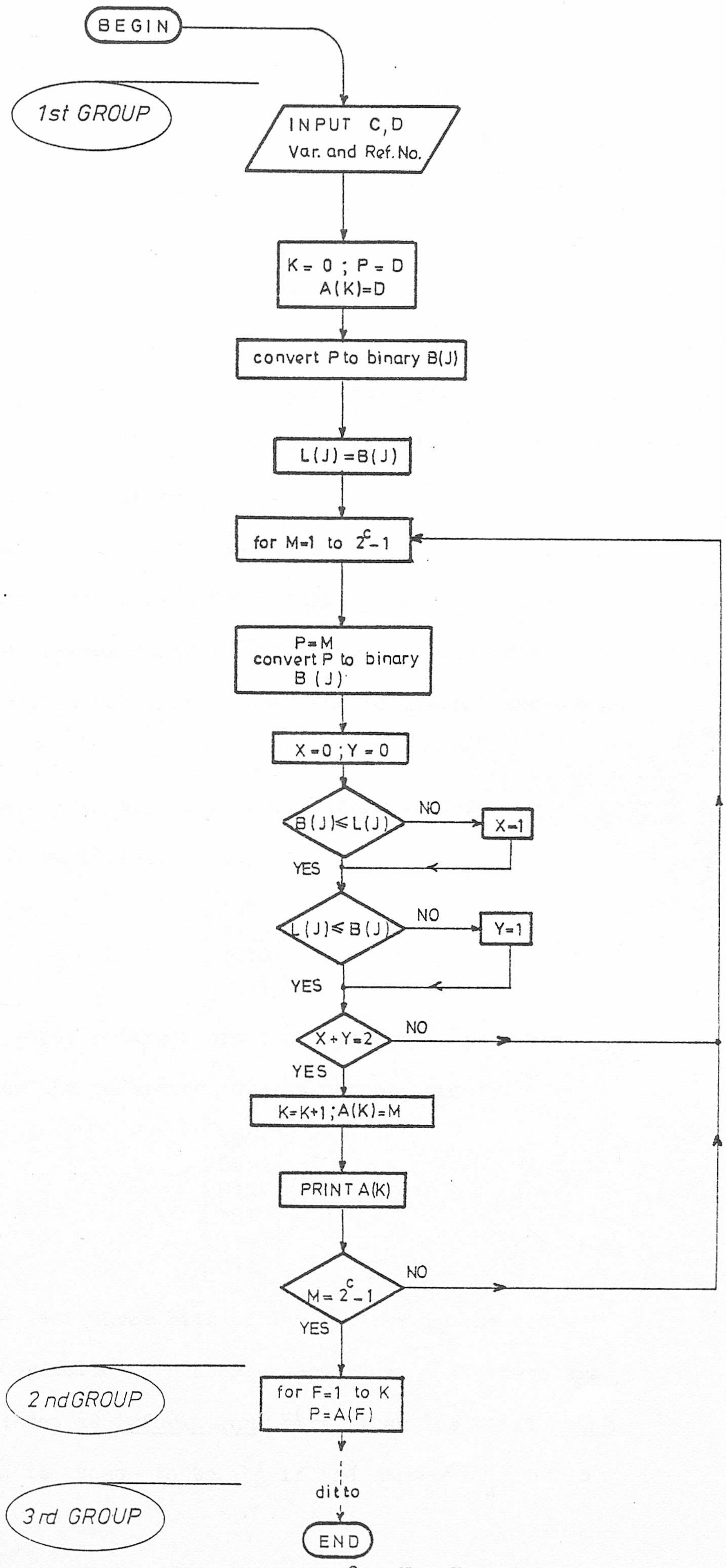


Figure 3.18 New Flowchart

constructed in such a way that, once a reference vector has been chosen, it is compared with all other possible vectors and the ones which are incomparable are printed in the first group. Each of the vectors in the first group is then compared with all the other vectors in the first group and those which are incomparable are printed in the second group, and so on. The format of the program results is illustrated in Figure 3.19 for the case of Berger code. Three examples of actual computer printout are shown in Figure 3.20. Note that the vectors are printed in decimal form for speed and convenience.

Obviously this program may be extended to deal with much larger binary vectors by simply adding more "groups" to the program. However, the program shown is sufficient to illustrate the technique.

After extensive program runs, the following conclusions were made:

(a) There are only four legal Berger vectors for four variables, namely:

$$\begin{bmatrix} 1001 \\ 1100 \\ 1010 \\ 0111 \end{bmatrix}$$

(b) However, a larger group of vectors is possible when the reference 1001 is chosen, namely:

$$\begin{bmatrix} 1001 \\ 0011 \\ 0101 \\ 1010 \\ 1100 \\ 0110 \end{bmatrix}$$

(c) The last three bits of the vectors in the above group follow a binary progression. If these are classed as <u>information</u> bits, then the first <u>check</u> bit is chosen to be a <u>1</u> if the number of ones in

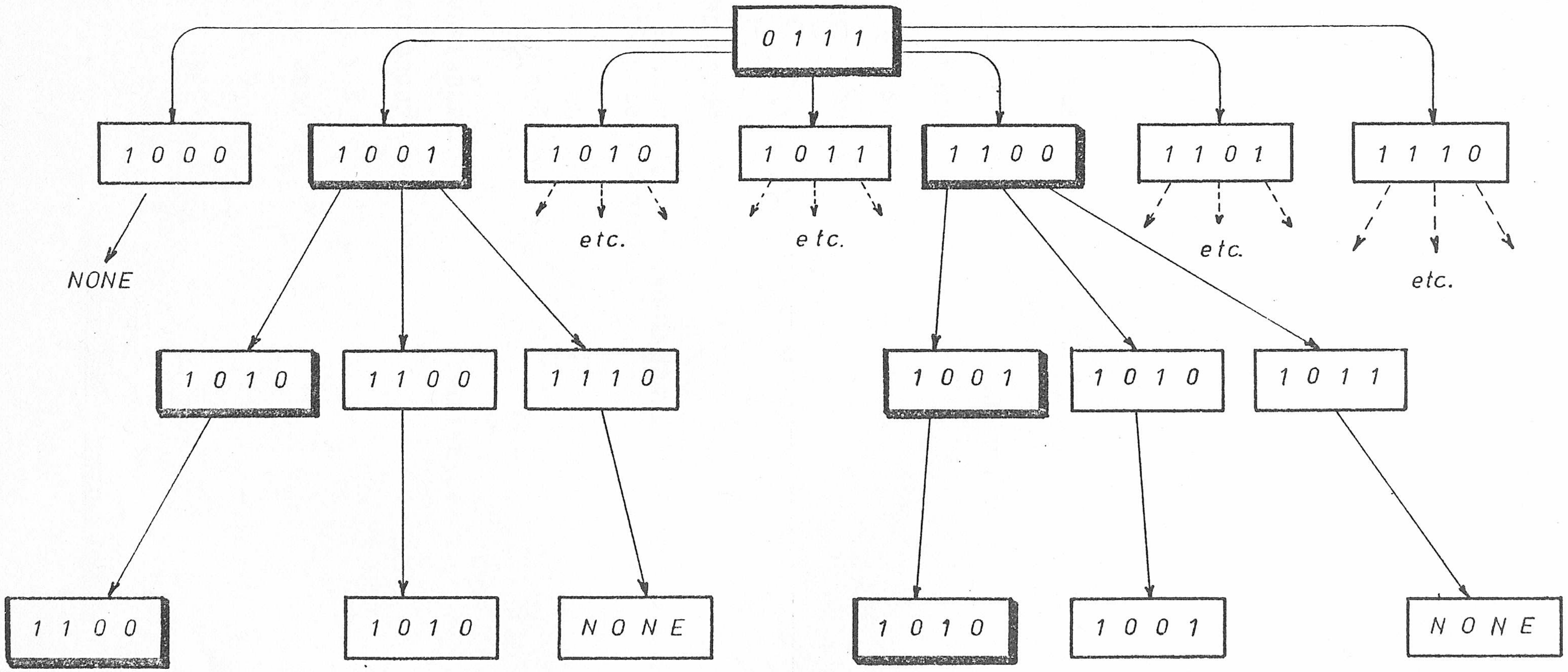


Figure 3.19 Format of Computer Program Results

```
VAR AND REF NO.? 4,7

 4 VARIABLES                   REFERENCE= 7

FIRST GROUP

 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14 \

SECOND GROUP

NONE\ 10  12  14 \ 9  12  13 \ 12  13  14 \ 9  10  11 \
10  11  14 \ 9  11  13 \

THIRD GROUP

 12 \ 10 \NONE\ 12 \ 9 \NONE\NONE\ 14 \ 13 \ 10 \ 9 \NONE\NONE\
14 \ 11 \NONE\ 13 \ 11 \


VAR AND REF NO.? 4,1

 4 VARIABLES                   REFERENCE= 1

FIRST GROUP

 2 \ 4 \ 6 \ 8 \ 10 \ 12 \ 14 \

SECOND GROUP

 4  8  12 \ 2  8  10 \ 8  10 12 \ 2  4  6 \ 4  6  12 \ 2  6  10 \
NONE\

THIRD GROUP

 8 \ 4 \NONE\ 8 \ 2 \NONE\NONE\ 12 \ 10 \ 4 \ 2 \NONE\NONE\ 12 \
6 \NONE\ 10 \ 6 \NONE\


VAR AND REF NO.? 3,2

 3 VARIABLES                   REFERENCE= 2

FIRST GROUP

 1 \ 4 \ 5 \

SECOND GROUP

 4 \ 1 \NONE\

THIRD GROUP

NONE\NONE\NONE\
```

Figure 3.20   Program Results

the information bits is odd. The check bit is a 0 if the number of ones in the information bits is even. Thus, a very useful type of code, known as a 2-out-of-4 parity-check code, is produced.

(d) Various other codes may be produced in the same way by careful study of the program results.

(e) Every feasible group of pairwise incomparable binary vectors may be produced using this computer program.

This computer analysis concludes the study of fail-safe digital machine design. Fundamental definitions are theorems have been presented along with various design techniques and illustrated by specific design examples. Throughout the study, relevant conclusions have been drawn and compared.

The following chapters are devoted to fault-tolerant digital machine design.

CHAPTER 4. AN INTRODUCTION TO FAULT-TOLERANT DIGITAL SYSTEMS

Fail-safe digital systems are quite acceptable from an error-detection point of view, since it is assumed that the system can be shut-down for a certain length of time for maintenance purposes. It is also assumed that maintenance is possible. However, in applications where continuous operation is essential for a specified length of time, fault-tolerant design techniques become very important. Without exception, fault-tolerant design techniques involve a certain amount of circuit redundancy, the specific techniques to be used depending upon whether or not repair is possible and also on the required duration of reliable operation.

Fault-tolerant design techniques can be divided into two classes. In the first, called fault-masking, the effects of any fault are masked by additional circuitry. This circuitry is an integral part of the system and no switching is involved, thus error-correction is instantaneous.

In the second class are schemes which detect and locate any fault in the system and replace the faulty unit by switching in a spare unit. These systems are called self-checking systems.

4.1 Fault-Masking Techniques

Fault-masking techniques are useful when the system is required to operate reliably over a relatively short period of time and repair is impossible. Over the past few years, this has been the basis for various fault-tolerant design techniques. These include triple-modular-redundancy, quadded logic, radial logic and the use of error-correcting codes.

(a) Triple-Modular-Redundancy (TMR)

This is perhaps the oldest form of fault-masking, in which a complete system is produced in triplicate. The three system outputs, which in fault-free operation are identical, are fed into a majority-logic gate as shown in Figure 4.1. This type of gate produces an output corresponding to the majority of the inputs. Therefore, if a perfect majority gate is assumed, the system illustrated in Figure 4.1 will never fail unless two or more units fail. It can be shown that the mean time before failure (MTBF) of this redundant system is less than that of the irredundant system [6]

. However, for small values of t, the time period, the probability of survival of the redundant system is greater than that of the irredundant system. Such systems are useful when a high reliability is required over a short period of time.

(b) Multiplexing [7]

The method of multiplexing is similar to the above method except that the original system is divided into subsystems and each subsystem is triplicated as illustrated in Figure 4.2. A fault in any element in a subsystem, including the majority gates, will be masked by this system. At the output, it is necessary to select the proper output from among the three outputs either by a fault-free circuit or an observer.

(c) Quadded Logic [8]

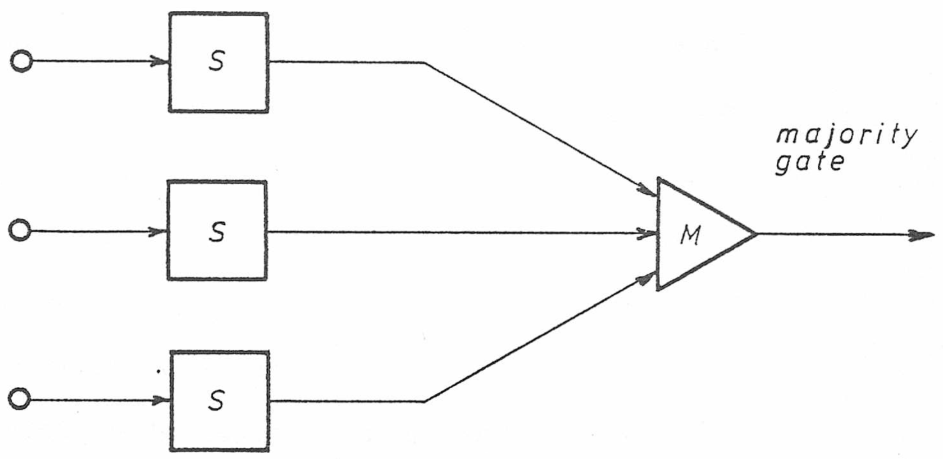


Figure 4.1 Triple-Modular-Redundancy

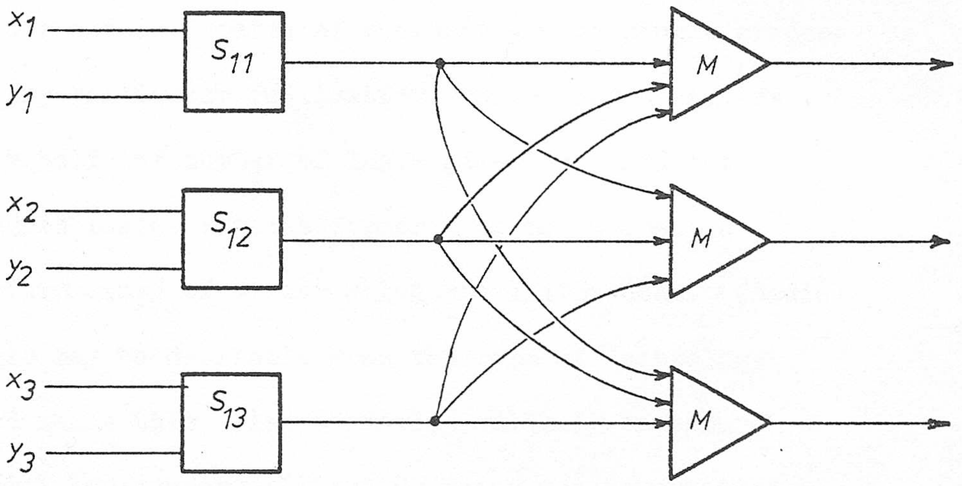


Figure 4.2 Multiplexing

As the name of this technique implies, logic elements appear in quadruplicate. Any fault which appears is corrected at the next level. Thus all single faults, except in the last two stages, are masked. Most multiple faults are also masked unless they appear in circuit elements which are close together. Any circuit containing AND, OR and NOT logic gates can be quadded. Similarly, quadded circuits can be designed using NAND and NOR gates (Jensen, 1963). Sequential circuits can be synthesised also, since flip-flops can be realised by treating them as circuits formed by interconnecting simple logic gates, but containing feedback.

(d) Radial Logic [9]

Radial logic makes use of the fault-masking properties of the NOR (or NAND) gate with duplicated inputs and is capable of correcting most single errors. In any particular realisation, radial logic requires only half the number of logic gates required for quadded logic, but the former does not correct a certain class of errors which the latter does. Radial logic may be desirable when the type of technology used makes this class of faults unlikely to occur. Radial logic using AND and OR gates can be obtained as a simple extension of the NOR realisation, but certain classes of faults still cannot be masked.

(e) Error-Correcting Codes [10]

A method, whereby error-correcting codes are used to obtain reliable digital systems, was proposed by

Armstrong in 1961. The method is actually a generalisation of the triplication and voting procedure discussed earlier. The technique is applicable to both combination and sequential circuits. Since this technique forms the basis for the fault-tolerant digital counter presented in the next chapter, it is discussed later in some detail.

4.2 Self-Checking Systems

So far, digital systems operating under two different sets of conditions have been discussed.

In the first, the fail-safe system, interruptions are tolerable and repair is possible, since only error-detection takes place and an F-state is reached. In this case, a system which is relatively easy to test is desirable so as to minimise the time required for maintenance.

In the second class, repair is impossible but the system is required to operate with high reliability for a relatively short period of time. The fault-masking techniques discussed in the preceding section are ideally suited for this application.

A third type of environment is one in which interruptions in the operation are intolerable but repair is possible. In order to operate under these conditions, the system should be self-checking. It should be able to detect any fault within itself, identify the faulty subsystem and switch it out of the system. This should be done in such a manner that the system can continue to operate with the remaining units, while the faulty unit is repaired.

(a) Electronic Switching System (ESS) [11]

The No. 1 Electronic Switching System (ESS) used in the Bell System for telephone switching is a highly reliable system, one of whose reliability objectives is that the system operation should not be interrupted for more than two hours over its 40-year life. In addition to the use of long-life components and conservative circuit design, this high degree of reliability is attained by duplicating the vital parts of the system so as to retain an operational system in the presence of component failures. Circuits and programs are provided to determine the faulty unit and switch it out of operation. Diagnostic programs and maintenance dictionaries are provided to locate the faulty package in the failed unit, leading to rapid repair.

(b) Self-Testing and Repair (STAR) Computer [12]

This is an experimental computer which was designed and constructed primarily for research and evaluation of self-repair techniques. Its performance characteristics are meant to be suitable for the guidance and control of unmanned interplanetary spacecraft. The computer is required to operate reliably over a period of several years. Temporary malfunctions may be tolerated provided they are detected and the computations repeated. Time is also available for switching out faulty units and switching in spares. The STAR computer has a fixed configuration of subsystems, with spares provided for each subsystem. Spares are permanently connected to the system through information buses, but are left unpowered. Replacement of a faulty subsystem by a spare is effected by turning off the power to the former and powering the latter.

This concludes the introduction and background to fault-tolerant digital systems. The following chapters deal with various approaches and techniques in the design of these systems. As mentioned earlier, the next chapter deals with a specific type of fault-masking technique, namely, the use of error-correcting codes, and its application to the design of a fault-tolerant digital counter.

CHAPTER 5. FAULT-TOLERANT DIGITAL COUNTER DESIGN

In this chapter, the use of error-correcting codes, or, more specifically, parity-checking codes, is discussed in detail. The design of a single-fault-tolerant digital counter using this technique is illustrated by a particular example using a modified first-order Reed-Muller parity-check matrix. The chapter is concluded by a computer-aided design study of fault-tolerant counters.

5.1 Parity-Check Codes and their Uses

Consider an m-input, n-output combinational circuit which can be designed so that it produces the correct outputs even in the presence of a single fault. If there is no shared logic between the shared outputs, then K check bits could be added and an error-correcting code used. If shared logic is allowed, then a single fault may affect several outputs and the code should be capable of correcting all errors that may result from a single fault.

A more efficient technique, suggested by Armstrong is to break the given m-input, n-output circuit into r sub-units. There may be shared logic between outputs within the same sub-unit, but no shared logic between sub-units. Errors produced by a single faulty sub-unit can be corrected by adding q p-output sub-units as shown in Figure 5.1. The outputs Z_{ij}, $i=r+1,\ldots,r+q$ serve as check bits for Z_{kj}, $k=1\ldots r$, and $j=1\ldots p$, in a single-error-correcting code.

In applying error-correcting techniques to sequential circuits, it is necessary to perform error-correction on the outputs as well as the state variables, otherwise a fault in a sub-unit, whose outputs are state variables, may be fed back

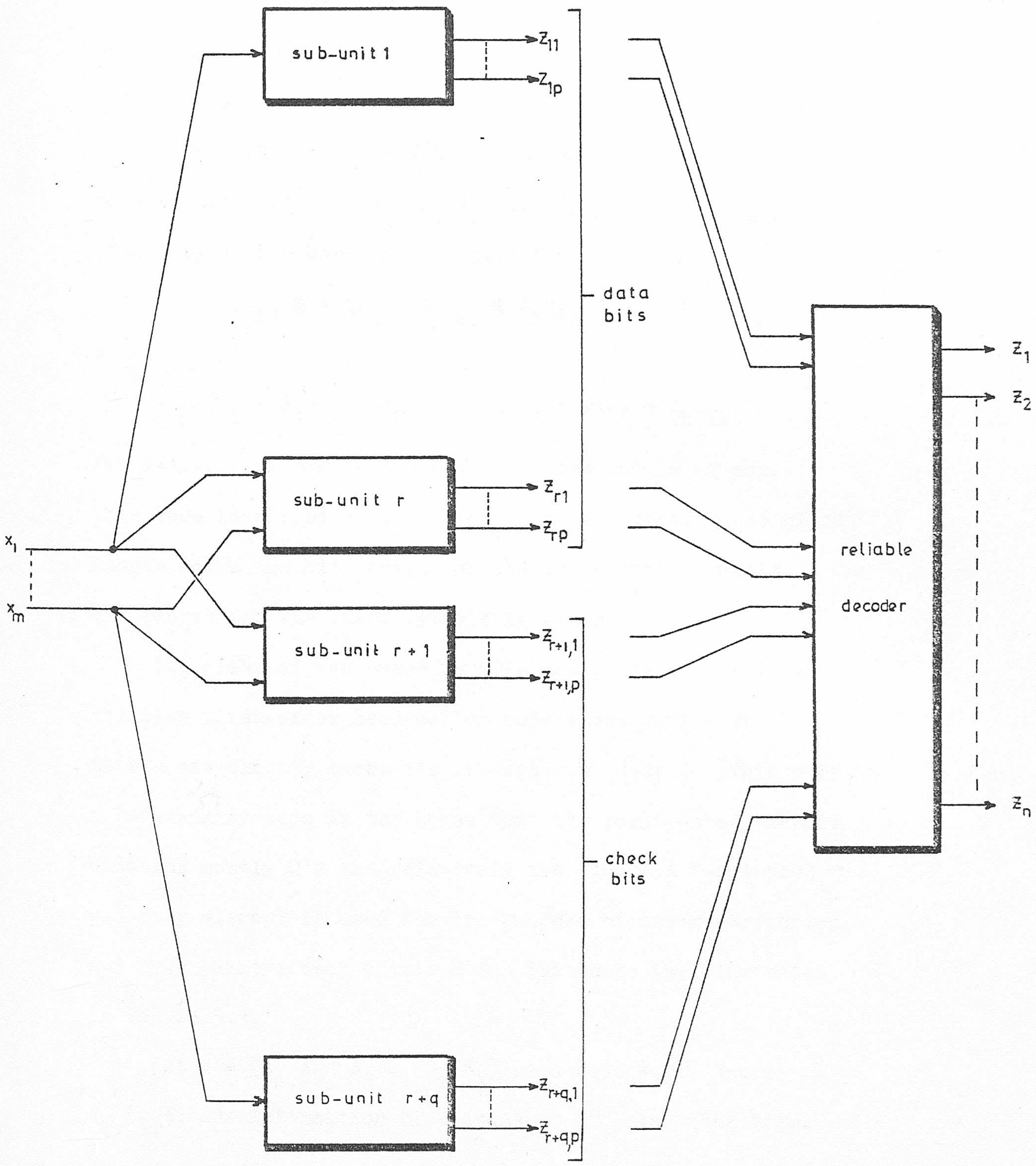


Figure 5.1 Method of Sub-Units

resulting in errors in more than one sub-unit at a later time.

A parity-check code is characterised by its parity-check matrix. A parity-check matrix H of n columns and n-k rows for any binary error-correcting code can be expressed in general in a reduced-echelon form as shown in Figure 5.2 [13]

I_{n-k} is an n-k identity matrix and Q is an n-k by k matrix with binary element q_{ij}.

The corresponding code space V consists of all elements v such that $vH^T = 0$, where H^T is the transpose of matrix H. More specifically, if $v = A_1, A_2, \ldots, A_k, B_1, B_2, \ldots B_{n-k})$, then v is a code word if and only if

$$A_1 q_{i1} \oplus A_2 q_{i2} \oplus A_3 q_{i3} \oplus A_k q_{ik} \oplus B_1 = 0$$

or

$$B_i = A_1 q_{i1} \oplus A_2 q_{i2} \oplus A_3 q_{i3} \oplus \ldots \oplus A_k q_{ik}$$

for i=1,2,...., n-k, where $\oplus$ denotes the modulo -2 sum. This code is called an (n,k) code, where n denotes the block length and k the bit length for the information symbols. The bit length for the check symbols is given by n-k.

The class of code used in this design technique is a modified first-order Reed-Muller code whose parity-check matrix has exactly three 1's in each row [14] . This code is a low-density code in the sense that its parity-check matrix contains mostly 0's and relatively few 1's. A 2-out-of-3 majority element is used for the purpose of error correction. A 3 by 6 parity-check matrix H for this code is illustrated in Figure 5.3.

Let $v = (A_1, A_2, A_3, B_1, B_2, B_3)$ be a code word, where A_1, A_2, A_3 are information bits and B_1, B_2, B_3 are check bits. Then

$$H = \begin{bmatrix} q_{11} & q_{12} & q_{13} & \cdots & q_{1k} & 1 & 0 & 0 & \cdots & 0 \\ q_{21} & q_{22} & q_{23} & \cdots & q_{2k} & 0 & 1 & 0 & \cdots & 0 \\ q_{31} & q_{32} & q_{33} & \cdots & q_{3k} & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & & \vdots & \vdots & \vdots & & \\ q_{n-k1} & q_{n-k2} & q_{n-k3} & & q_{n-kk} & 0 & 0 & 0 & & 1 \end{bmatrix}$$

$$= \left[Q, I_{n-k} \right]$$

Figure 5.2 General Form of Parity-Check Matrix

$$H = \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Figure 5.3 Reed-Muller Code Parity-Check Matrix

$vH^T = 0$ and from the given matrix H, a set of parity-check equations can be derived, thus:

$$B_1 = A_1 \oplus A_2$$
$$B_2 = A_2 \oplus A_3$$
$$B_3 = A_3 \oplus A_1 \qquad (1)$$

or

$$A_1 = A_2 \oplus B_1 = A_3 \oplus B_3$$
$$A_2 = A_1 \oplus B_1 = A_3 \oplus B_2$$
$$A_3 = A_1 \oplus B_3 = A_2 \oplus B_2 \qquad (2)$$

Note that each A_i, for $i = 1,2,3$ in the set of (2), can be determined by exactly three <u>independent</u> relationships.

Therefore:-

$$A_1(k) = Ma_{jk}\ (A_1,\ A_2 \oplus B_1,\ A_3 \oplus B_3)$$
$$A_2(k) = Ma_{jk}\ (A_2,\ A_1 \oplus B_1,\ A_3 \oplus B_2)$$
$$A_3(k) = Ma_{jk}\ (A_3,\ A_1 \oplus B_3,\ A_2 \oplus B_2) \qquad (3)$$

where the subscript k denotes the k^{th} physical realisation of the particular majority element. These majority elements of (3) give the correct output if, at most, one of the terms $A_1, A_2, A_3, B_1, B_2, B_3$ has a component fault.

From the set of (1):

$$\bar{A}_1 = \bar{A}_2 \oplus B_1 = \bar{A}_3 \oplus B_3$$
$$\bar{A}_2 = \bar{A}_1 \oplus B_1 = \bar{A}_3 \oplus B_2$$
$$\bar{A}_3 = \bar{A}_1 \oplus B_3 = \bar{A}_2 \oplus B_2 \qquad (4)$$

therefore:

$$\bar{A}_1(k) = Ma_{jk}\ (\bar{A}_1,\ \bar{A}_2 \oplus B_1,\ \bar{A}_3 \oplus B_3)$$
$$\bar{A}_2(k) = Ma_{jk}\ (\bar{A}_2,\ \bar{A}_1 \oplus B_1,\ \bar{A}_3 \oplus B_2)$$
$$\bar{A}_3(k) = Ma_{jk}\ (\bar{A}_3,\ \bar{A}_1 \oplus B_3,\ \bar{A}_2 \oplus B_2) \qquad (5)$$

These results can be applied to the design of a single-fault-tolerant three-stage counter.

5.2 Fault-Tolerant Digital Counter

As an illustration, consider the design of an ordinary three-stage binary counter with information bits A_3, A_2 and A_1 as shown in Figure 5.4. In order to produce a fault-tolerant version of this counter, three auxiliary check stages are required. These check bits B_3, B_2 and B_1 are produced using the equations of (1). The control functions are now required in order to realise the circuit. T-type flip-flops (or J-K flip-flops with J and K tied together) are used as memory elements for reasons discussed later. The characteristic equation of a T-Type flip-flop:

$$\Delta Q(t) = Q(t) \oplus Q(t+\tau)$$

is used to produce the control functions as illustrated in Figure 5.4. Minimisation produces the final control equations as shown. It can be seen from these equations that the variable A_1 is required three times, $\bar{A}_1$ is required twice, A_2 once and $\bar{A}_2$ twice. Therefore, eight majority elements are needed to prevent any "bottleneck" problems. The final circuit diagram is illustrated in Figure 5.5. Note that these majority elements contain not only a 3-input majority logic gate, but also two exclusive - OR gates in order to satisfy the conditions given in equations (3) and (5). This circuit was built and tested and found to be completely fault-tolerant to single logic faults, irrespective of the type of fault or where it occurred.

Throughout this study, it has been stressed that, in order to produce a reliable machine, it is desirable to use the minimum of components. Therefore, T-type flip-flops were employed in the fault-tolerant counter to ensure a minimal circuit. Obviously, the use of J-K flip-flops would have

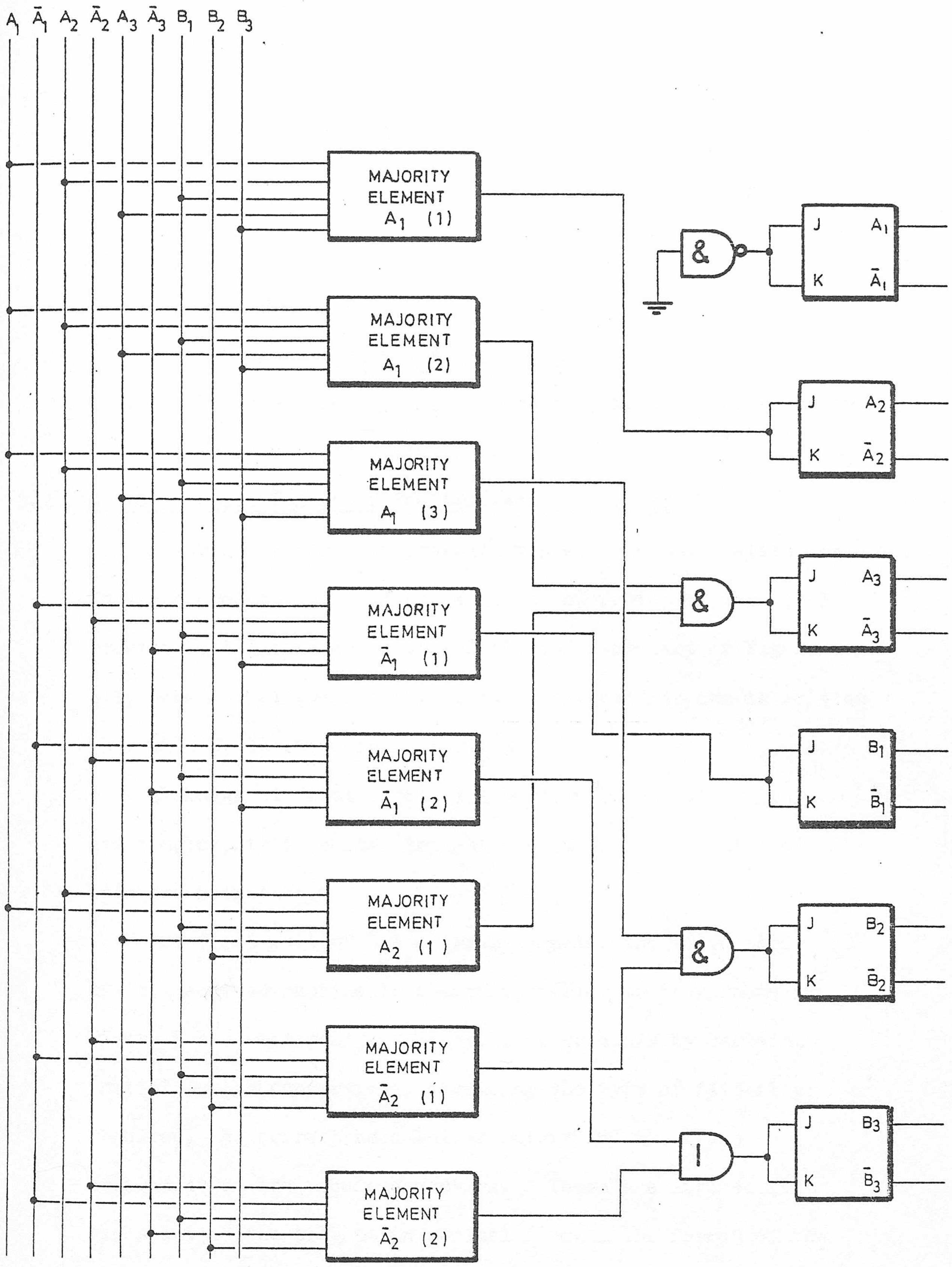
state	information bits			check bits			change operator form					
	A_3	A_2	A_1	B_3	B_2	B_1	A_3	A_2	A_1	B_3	B_2	B_1
S_0	0	0	0	0	0	0	0	0	1	1	0	1
S_1	0	0	1	1	0	1	0	1	1	1	1	0
S_2	0	1	0	0	1	1	0	0	1	1	0	1
S_3	0	1	1	1	1	0	1	1	1	0	0	0
S_4	1	0	0	1	1	0	0	0	1	1	0	1
S_5	1	0	1	0	1	1	0	1	1	1	1	0
S_6	1	1	0	1	0	1	0	0	1	1	0	1
S_7	1	1	1	0	0	0	1	1	1	0	0	0
S_8	0	0	0	0	0	0	0	0	1	1	0	1

$A_1 = 1,$ $\qquad B_1 = \bar{A}_1$

$A_2 = A_1,$ $\qquad B_2 = A_1\bar{A}_2$

$A_3 = A_1A_2,$ $\qquad B_3 = \bar{A}_1+\bar{A}_2$.

Figure 5.4 State Table and Control Functions



A_1 $\bar{A}_1$ A_2 $\bar{A}_2$ A_3 $\bar{A}_3$ B_1 B_2 B_3
MAJORITY ELEMENT A_1 (1)
MAJORITY ELEMENT A_1 (2)
MAJORITY ELEMENT A_1 (3)
MAJORITY ELEMENT $\bar{A}_1$ (1)
MAJORITY ELEMENT $\bar{A}_1$ (2)
MAJORITY ELEMENT A_2 (1)
MAJORITY ELEMENT $\bar{A}_2$ (1)
MAJORITY ELEMENT $\bar{A}_2$ (2)
&
&
&
1
J K A_1 $\bar{A}_1$
J K A_2 $\bar{A}_2$
J K A_3 $\bar{A}_3$
J K B_1 $\bar{B}_1$
J K B_2 $\bar{B}_2$
J K B_3 $\bar{B}_3$

Figure 5.5 Three-Stage Single-Fault-Tolerant Counter

required much more combinational logic, and if D-type had been used, the control equations would have been:

$$A_1 = \bar{A}_1$$

$$A_2 = A_1 \oplus A_2$$

$$A_3 = \bar{A}_1A_3 + \bar{A}_2A_3 + A_1A_2\bar{A}_3$$

$$B_1 = \bar{A}_2$$

$$B_2 = A_1\bar{A}_3 + A_2\bar{A}_3 + \bar{A}_1\bar{A}_2A_3$$

$$B_3 = \bar{A}_1\bar{A}_3 + A_2\bar{A}_3 + A_1\bar{A}_2A_3$$

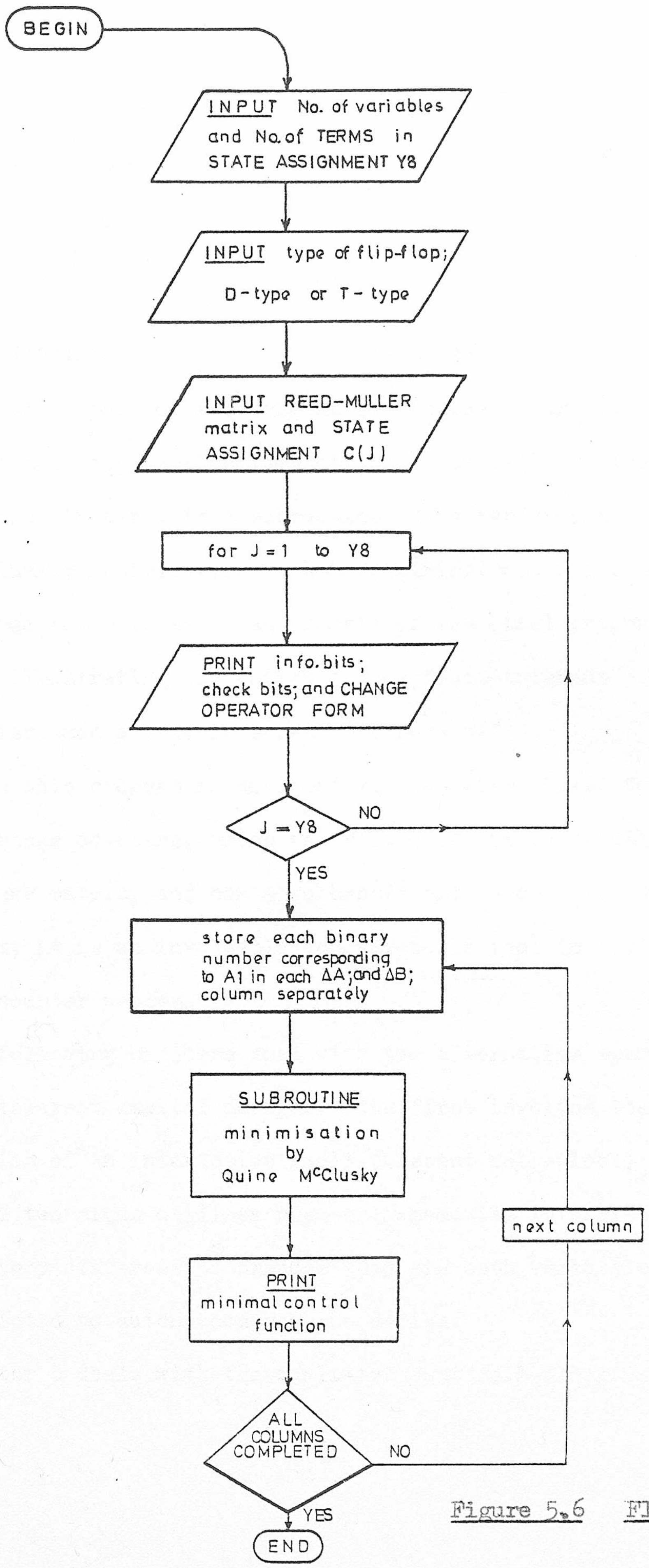
These would have produced a much more complicated and, therefore, a much less reliable circuit.

5.3 Computer-Aided Design of Fault-Tolerant Counters

A series of computer programs was written to simulate the design procedure outlined above. The operation of the final program is illustrated in the simplified flowchart of Figure 5.6; the actual program which, for technical reasons is written in FORTRAN, is shown in Appendix 3.

Although at first sight, the program looks rather long and complex, it is quite straightforward and can be divided into two main parts.

The first part of the program computes the state table of the required machine in a format similar to that shown in Figure 5.4. Naturally, this table is governed by certain initial design constraints, including the type of flip-flop required, the correct Reed-Muller matrix and the state assignment of the required machine. These are inputed at the start of the program in decimal form. The format of the Reed-Muller matrix is inputed as a string of digits, indicating the positions of the 1's in the matrix. For example, the



BEGIN
INPUT No. of variables and No. of TERMS in STATE ASSIGNMENT Y8
INPUT type of flip-flop; D-type or T-type
INPUT REED-MULLER matrix and STATE ASSIGNMENT C(J)
for J = 1 to Y8
PRINT info. bits; check bits; and CHANGE OPERATOR FORM
J = Y8
NO
YES
store each binary number corresponding to A1 in each ΔA; and ΔB; column separately
SUBROUTINE minimisation by Quine McClusky
PRINT minimal control function
next column
ALL COLUMNS COMPLETED
NO
YES
END

Figure 5.6 Flowchart

matrix in Figure 5.3 is represented as 122313. Parity-check matrices may be constructed for four and five variables as shown below.

$$\left[\begin{array}{c|c} 1100 & 1000 \\ 0110 & 0100 \\ 1001 & 0010 \\ 0011 & 0001 \end{array}\right] \qquad \left[\begin{array}{c|c} 10100 & 10000 \\ 01010 & 01000 \\ 00101 & 00100 \\ 10010 & 00010 \\ 01001 & 00001 \end{array}\right]$$

12231434 1324351425

The second part of the program stores the binary numbers in each "change operator" column of the state table. These are then fed, in turn, to a subroutine which performs a Quine-McCluskey minimisation. Hence, minimal control functions are produced and printed. An example of the final program printout, illustrating the design of the fault-tolerant counter discussed above, is shown in Figure 5.7.

Since this program is designed to cope with three, four and five-stage counters, using variations of the Reed-Muller parity-check matrix, and can also handle any conceivable state assignment, it is an invaluable and versatile tool in fault-tolerant counter design.

The following chapters deal with two alternative approaches to fault-tolerant digital design. The first involves the construction of an interactive fault-tolerant cell-block, whilst the second technique utilises read-only memories in digital design. Although very different techniques they are both versatile and are not restricted to autonomous machine design.

Chapter 6 deals with the fault-tolerant cell-block.

```
INFO BITS      CHECK BITS      CHANGE OP FORM
                               DELTA A'S      DELTA B'S
 Ø  Ø  Ø        Ø  Ø  Ø
 Ø  Ø  1        1  Ø  1         Ø  Ø  1        1  Ø  1
 Ø  1  Ø        Ø  1  1         Ø  1  1        1  1  Ø
 Ø  1  1        1  1  Ø         Ø  Ø  1        1  Ø  1
 1  Ø  Ø        1  1  Ø         1  1  1        Ø  Ø  Ø
 1  Ø  1        Ø  1  1         Ø  Ø  1        1  Ø  1
 1  1  Ø        1  Ø  1         Ø  1  1        1  1  Ø
 1  1  1        Ø  Ø  Ø         Ø  Ø  1        1  Ø  1
 Ø  Ø  Ø        Ø  Ø  Ø         1  1  1        Ø  Ø  Ø

DELTA A3=A2A1+A2A1+
DELTA A2=A1+A1+A1+A1+
DELTA A1=1
DELTA B3=A2'+A2'+A1'+A1'+
DELTA B2=A2'A1+A2'A1+
DELTA B1=A1'+A1'+A1'+A1'+

  END
```

Figure 5.7 Results

CHAPTER 6. FAULT-TOLERANT CELL-BLOCK DESIGN

Throughout this study, classical methods have been primarily used in the design of both fail-safe and fault-tolerant sequential circuitry. Classical design generally involves a verbal description of the system function, followed by the construction of a state graph illustrating the various states and transitions required to perform this function. This is usually a straightforward task. However, the subsequent steps in the design procedure are not quite so simple and it generally requires the expertise of the design engineer to produce a reliable system with the minimum of components.

On the other hand, with the advent of microcircuits using large-scale-integration, electronics is rapidly becoming a "black-box" technology in the sense that very complicated circuits are now becoming commercially available in single packages. The piecing-together of these individual units to produce complex systems is now the primary role of the engineer.

It seems good sense, therefore, to design a logic element which, when incorporated in a system of identical elements, is as close as possible to the exact analogue of the state graph, setting and resetting according to the various transitions required by the system. The desirable properties of such an element are that it represents a state on the state graph, connected to other states in one-to-one correspondence with the state graph arrows, and it indicates or "remembers" the state of the system at any time. A circuit built of these elements would also have the advantages that it is already designed once the state graph is designed, and the circuit could be easily understood by anyone who understood its function. [15]

Moreover, if this logic element could be made fault-tolerant, the result would be simplified design coupled with increased reliability. The design of such a fault-tolerant cell-block is discussed in detail in this chapter.

6.1 Initial Cell-Block Development

During the development of a suitable cell-block, several designs were built and tested. These will be investigated in turn.

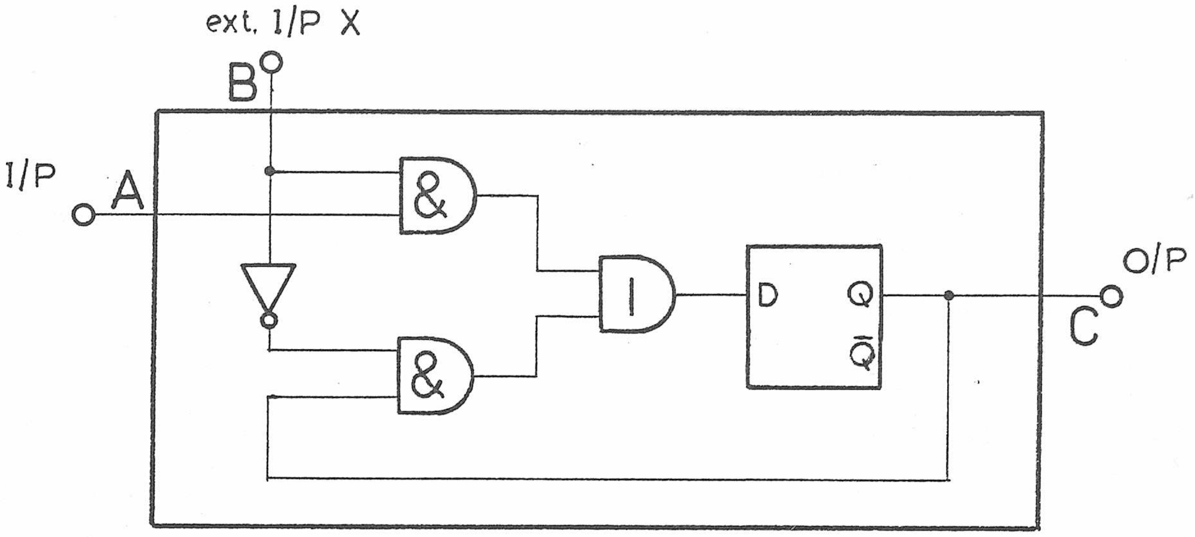
(i) An initial design is illustrated in Figure 6.1(a). If Q is a logical 1 and X is a logical 0, the internal feedback loop and associated combinational logic ensures that the cell remains in a high state. In this condition, the cell is effectively isolated from all other cells in the system. If, however, X becomes a logical 1, the D input becomes a 0, resetting the cell on the following clock pulse, while the Q output enables the following cell-block. A typical state graph, representing a simple ring-counter, is illustrated in Figure 6.1(b) and the circuit implementation is shown in Figure 6.1(c).

Although this cell-block functions correctly it has certain disadvantages:-

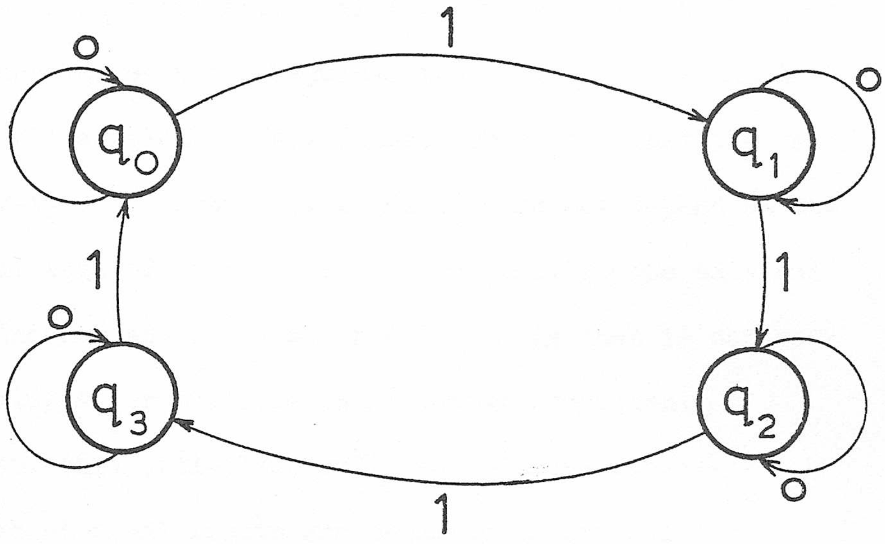
(a) The circuit relies on an incoming logical 0 for resetting.

(b) The cell is limited to single-input, single-output operation.

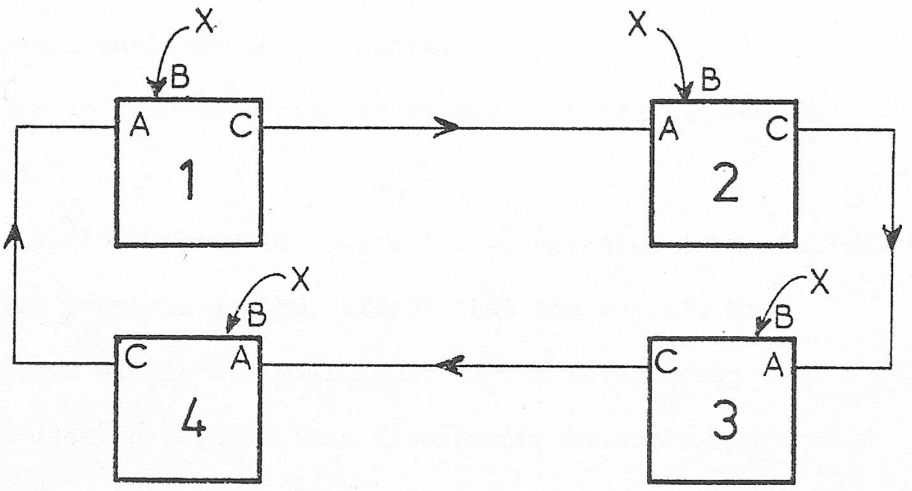
(c) The state-graph analogy is broken, since there is no external feedback loop from output to input representing the 'same-state' condition.



(a) Simple cell-block



(b) Simple state diagram



(c) Circuit implementation

Figure 6.1

After this design attempt, it became elear that the required combinational logic would be more usefully employed at the output of the memory element. This led to a new design.

(ii) The design shown in Figure 6.2(a) uses a J-K-flip-flop as the memory element, since this proves more versatile for resetting purposes. If Q is a logical 1 and both the external inputs are logical 0, the cell remains in the high state, since a logical 0 is fed back to the K-input of the flip-flop. If, however, one of the external inputs is high, the corresponding AND gate is enabled and a logical 1 is fed back to the K-input. On the occurrence of a clock-pulse, the cell is reset and the following cell enabled. This means, therefore, that the cell is effectively self-resetting, since it does not depend on the logic signal applied at the J-input, but only on the external inputs. Another advantage of this design is that it can be easily developed for multiple input-output operation.

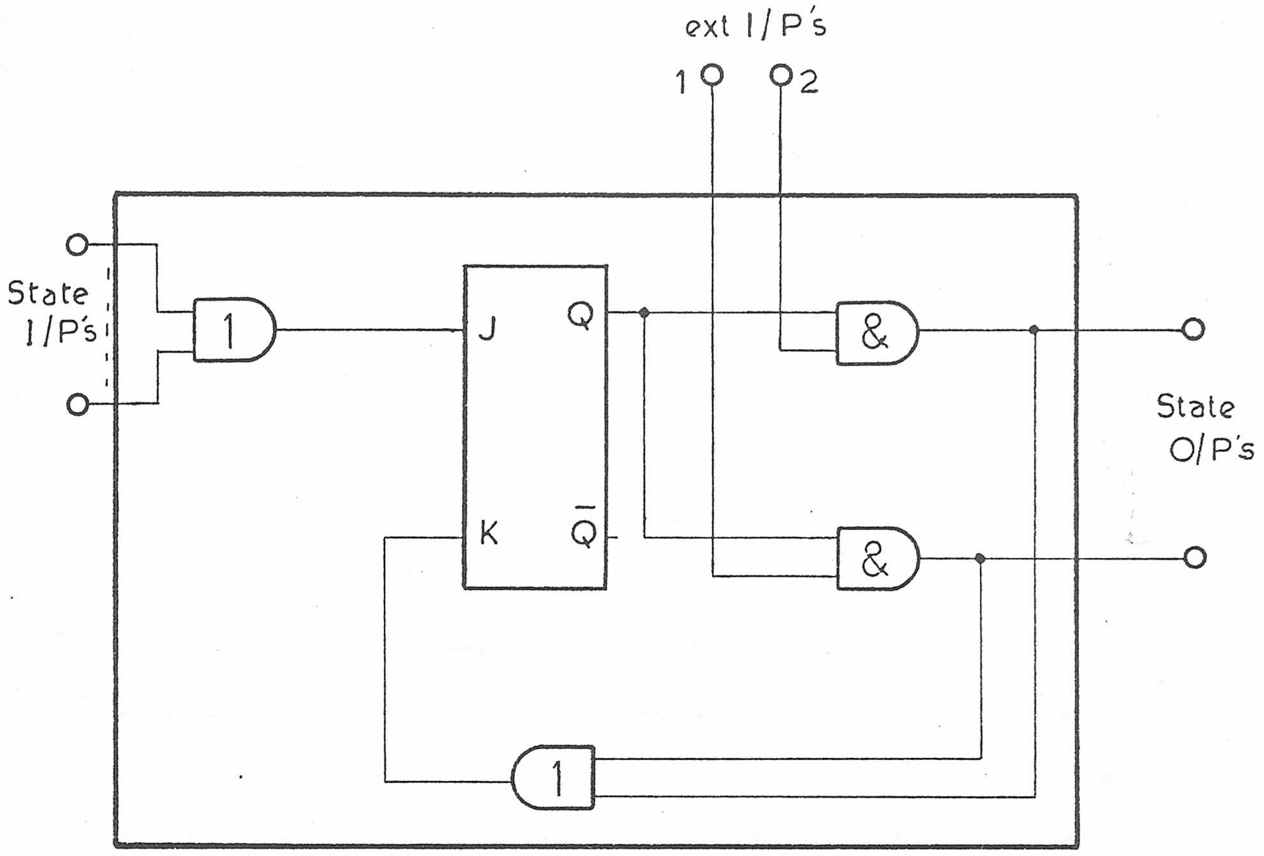
However, this cell-block still has disadvantages:-

(a) Two external inputs are required to control two output signals.

(b) The state-graph analogy is still broken for the reason outlined in (c) above.

In order to overcome these drawbacks, a third cell-block was designed.

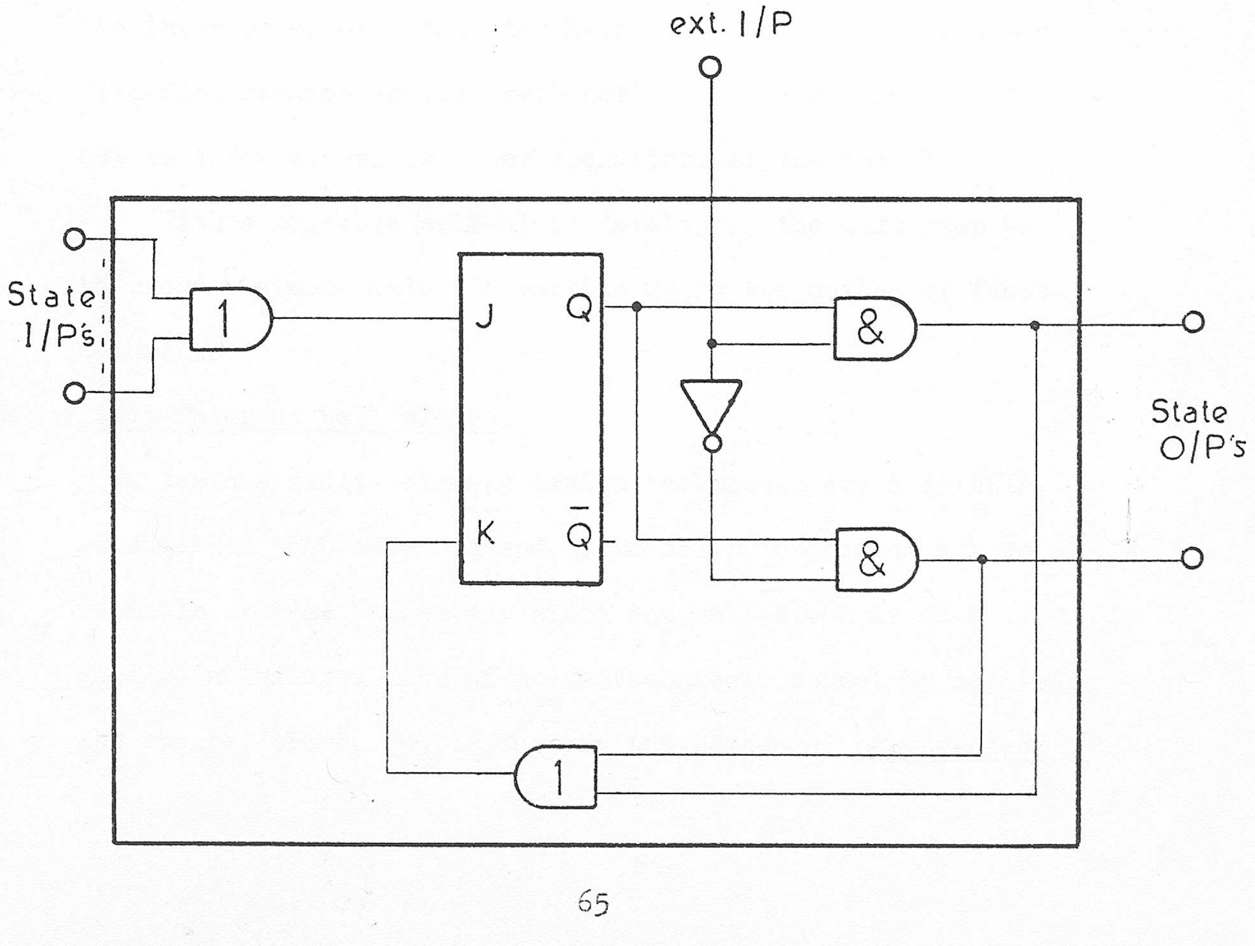
(iii) The cell-block shown in Figure 6.2(b) operates in a similar manner to the previous design, except that the outputs are controlled by a single external input X. Although only two outputs are used throughout this discussion, obviously any number of required outputs could be provided by a simple extension of



(a) 2-input/output cell-block

Figure 6.2

(b) Improved version



this technique. The state graph of Figure 6.3(a) is realised via the circuit shown in Figure 6.3(b). Note, in this case, that the circuit is directly analogous to the graph and the ultimate aim has been achieved.

However, close inspection and testing of this system revealed that, subject to certain input conditions, the circuit operates erroneously.

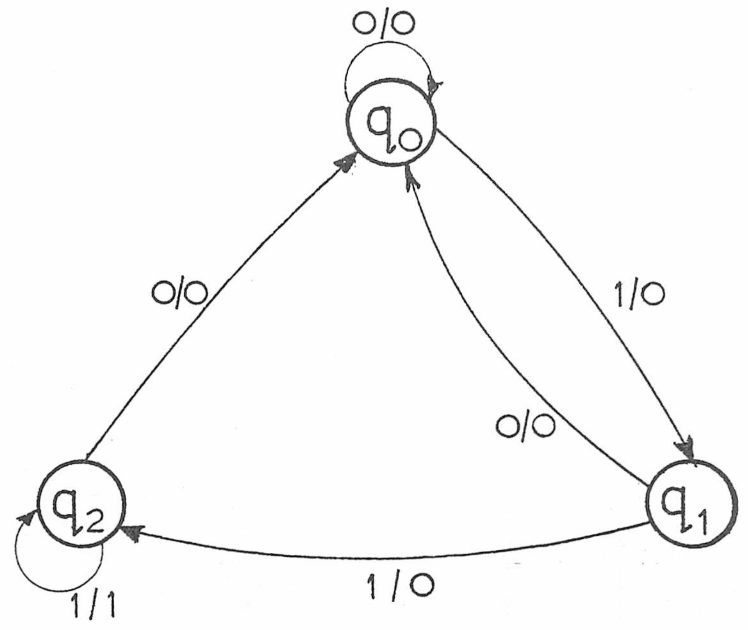
When the external input X is a logical 0 and the output of q_0 is fed back to its own input J, the required response is that q_0 will remain in the high state. However, the K-input line is also a logical 1 and, therefore, the cell-block will reset. The AND gates can no longer be enabled and the system ceases to operate. The circuit, therefore, must be amended if this situation is to be avoided.

(iv) This is done quite simply by insertion of an inverter and AND gate on the reset line as shown in Figure 6.4. The K-input is now controlled by the state input to the cell-block, so that, the inverter ensures that the K-input is a logical 0 and the flip-flop remains in the 'set' position. The added logic does not impair the various other operations of the cell-block.

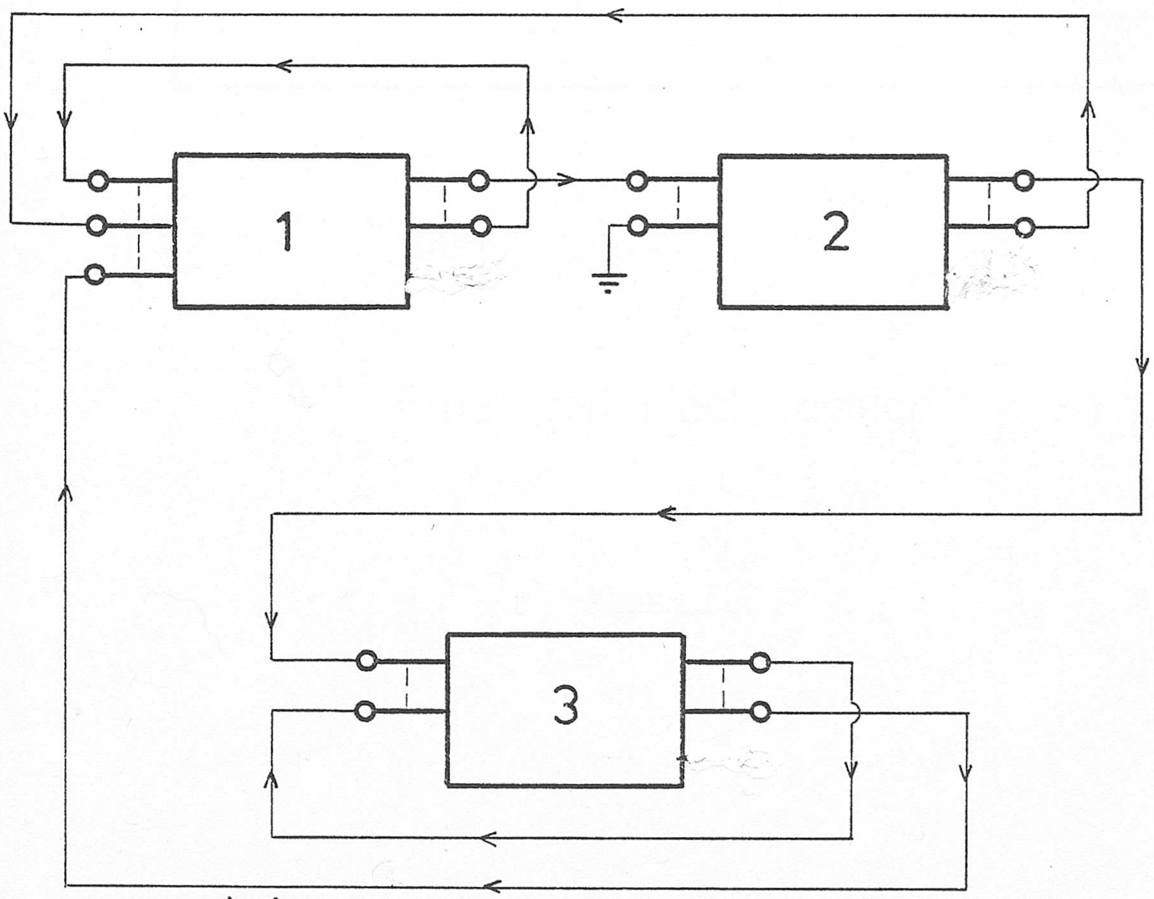
With a suitable cell-block developed, the next step was to produce a more reliable version using the method of fault-tolerance.

6.2 <u>Fault-Tolerant Cell-Block</u>

Various fault-tolerant design techniques are available, as outlined in Chapters 4 and 5, in order to produce a more reliable system. However, since the cell-block is of a sequential nature, many of these techniques cannot be applied. The choice, therefore, is between the method of <u>parity-check</u>

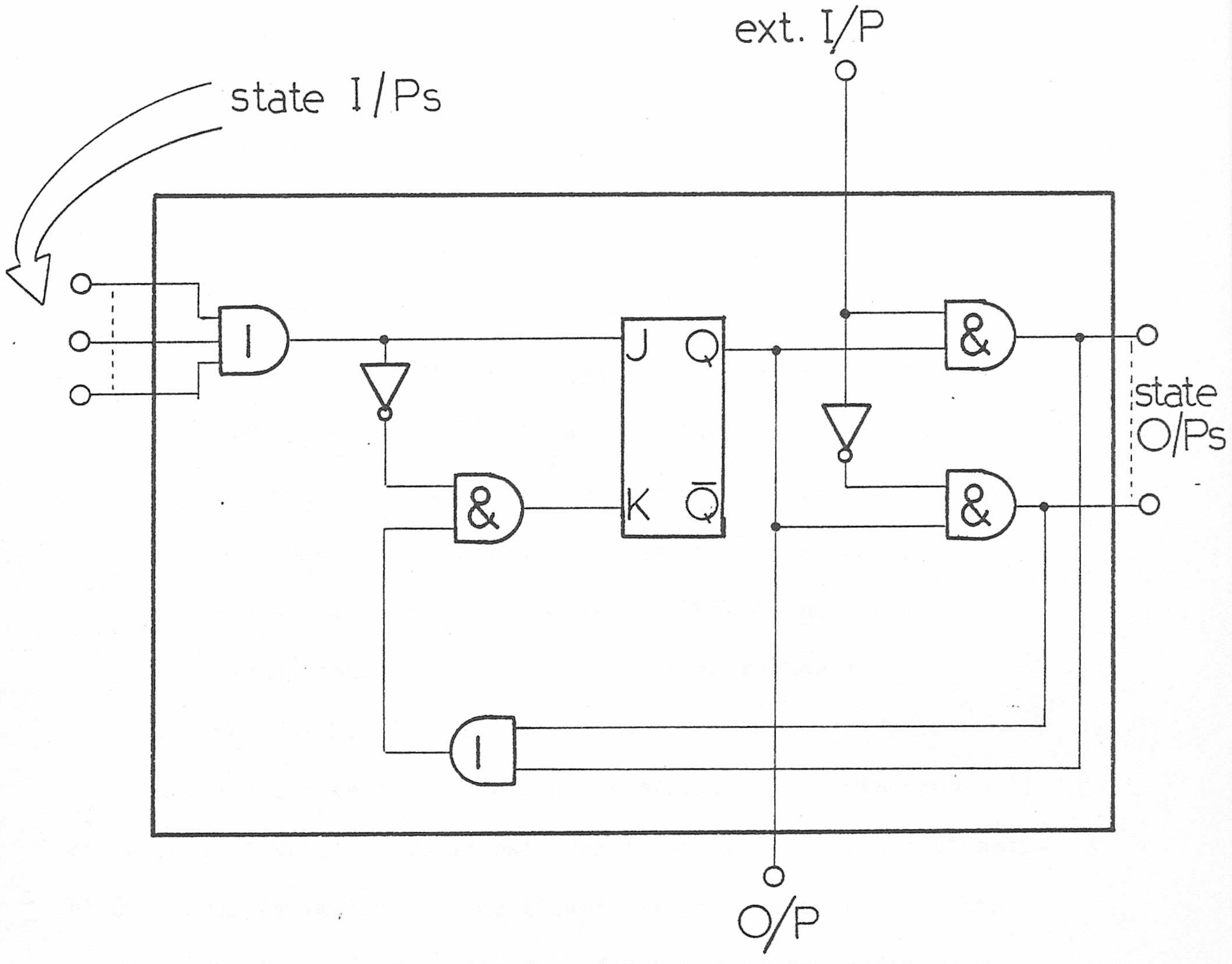


(a) State diagram example



(b) circuit implementation

Figure 6.3



Final cell-block design

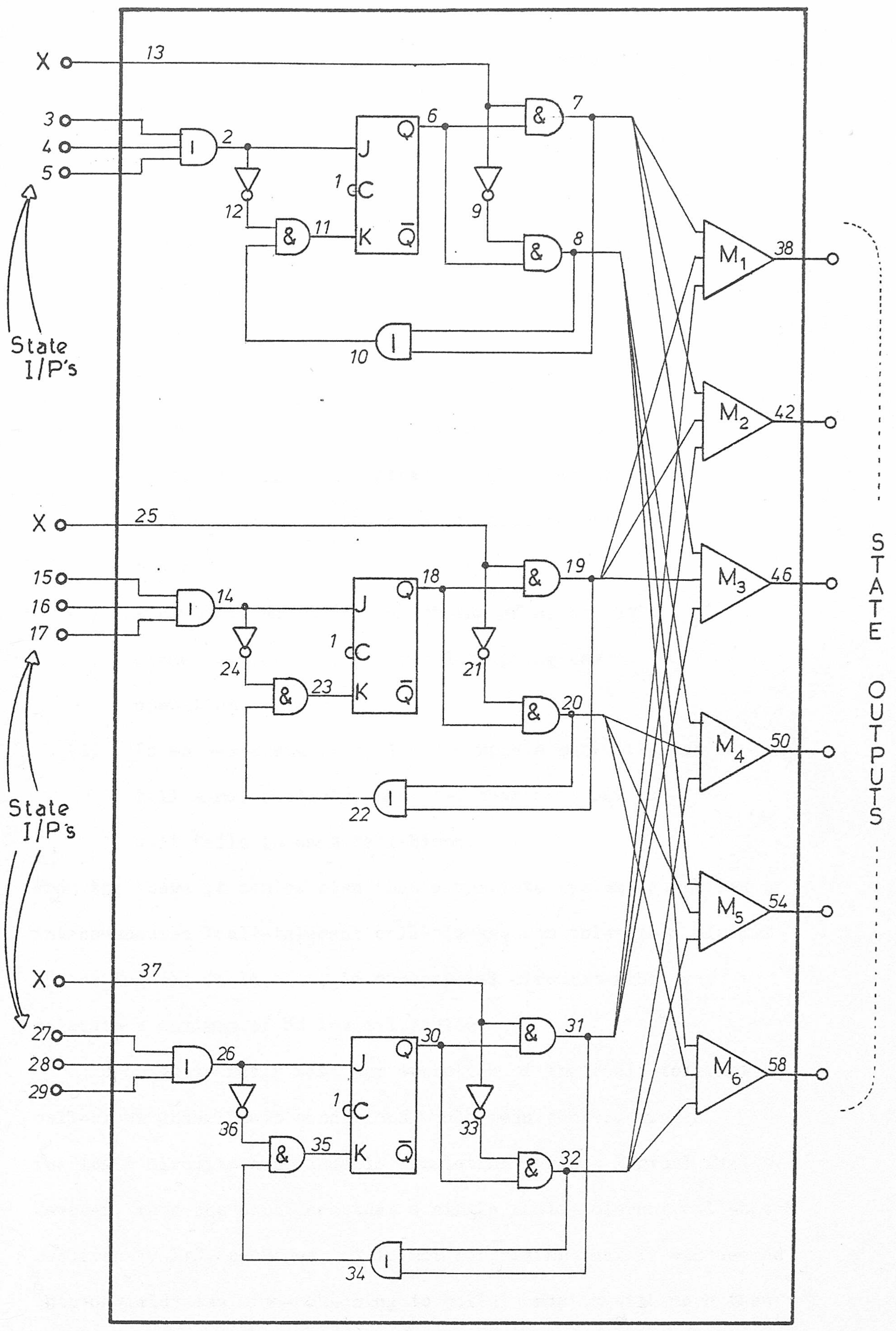
Figure 6.4

codes and triple-modular-redundancy. The former technique presents immediate problems as far as the cell-block is concerned for the following reasons:-

(a) This method is aimed primarily at autonomous systems. Although it is not altogether impossible to adapt the technique for sequential circuits, it is no simple matter to incorporate external inputs to produce a satisfactory design.

(b) This method requires at least three memory elements to produce a sufficient number of binary digits for checking purposes. Since the cell-block contains only one flip-flop, some additional redundancy would have to be introduced from the outset, even before the method was applied. This is not only very uneconomical, but also tends to reduce the initial level of reliability.

On the other hand, the cell-block offers no restrictions to the use of triple-modular-redundancy, and a fault-tolerant cell-block is simply implemented as illustrated in Figure 6.5. This is composed of three identical cell-block sub-units feeding six majority-logic gates. In normal operation, the signals applied to each majority gate are identical with the result that, depending on the value of the external input, three versions of the same signal are obtained at the outputs of the majority-logic gates. This means that the fault-tolerant cell-block continues to function correctly in the event of a single logical fault occurring in any of the circuit elements, including the majority logic. However, when this cell-block is used in conjunction with others to produce a required



Fault-tolerant cell-block

Figure 6.5

design, under certain conditions reliable operation may be achieved in the presence of various simultaneous faults.

(a) Two majority gates may fail within the same cell-block, provided that they receive different input signals. For example, in Figure 6.5, gates $M1_1$ and $M2_1$ may fail simultaneously without disrupting normal operation.

(b) In an N-state system, using N fault-tolerant cell-blocks, $1 \rightarrow N$ majority gates may fail without producing erroneous operation, provided that the failures occur in the same position in each cell-block.

(c) A complete sub-unit, comprising of eight logic elements, may fail without disrupting the operation.

(d) In an N-state system, $1 \rightarrow N$ complete sub-units may fail simultaneously, provided that only one sub-unit fails in each cell-block.

From the above it can be seen that a complete system, comprised of interconnected fault-tolerant cell-blocks, can tolerate a minimum of one logical fault, but, in exceptional circumstances, may tolerate a maximum of 8N logical faults.

In order to fully test the operation of the fault-tolerant cell-block under fault conditions would mean constructing a few logic circuits and manually simulating various logical faults. However, when one considers that a single fault-tolerant cell-block requires 10 I.C. packages, it is not surprising that it was deemed both unwieldy and time-consuming to build a system with more than three states using these discrete components.

It was decided, therefore, to simulate various logic systems on the computer using the Reynolds Logic Simulator.

6.3 Logic Simulation of Fault-Tolerant Cell-Block

The Reynolds Logic Simulation program is described in detail in Appendix 4.

A data file, entitled TMR. DAT, which describes the fault-tolerant cell-block, is shown in Figure 6.6. Note that every data file commences with -1, which indicates a new data file, and ends with -18, which returns control to the terminal keyboard. Note also that, since there is no facility in the program for majority-logic gates, these are replaced by equivalent logic networks whose function is given by:-

$$f = x.y + y.z + x.z$$

The program is first run under fault-free conditions to assess the normal operation of the fault-tolerant cell-block. The resulting computer printout is shown in Figure 6.7. Note that various comments and guidelines have been added for the sake of clarity. The circuit functions correctly under all possible input conditions, therefore various logical faults can now be simulated in order to test the fault-tolerant aspect of the system.

Logical faults are simulated quickly and simply using the Reynolds program. The command -2 allows any specified connections to be updated, so that any node may be assigned either a logical zero or a logical one using the system functions 14 or 15 respectively.

Using this technique, faults were induced in various circuit elements of the cell-block and the resulting annotated printout

```
-1
1 16 Ø
2 2 3 4 5 Ø
3 1 Ø
4 1 Ø
5 1 Ø
6 1Ø 1 2 11 Ø
7 4 6 13 Ø
8 4 6 9 Ø
9 3 13 Ø
1Ø 2 7 8 Ø
11 4 1Ø 12 Ø
12 8 2 Ø
13 1 Ø
14 2 15 16 17 Ø
15 1 Ø
16 1 Ø
17 1 Ø
18 1Ø 1 14 23 Ø
19 4 18 25 Ø
2Ø 4 18 21 Ø
21 8 25 Ø
22 2 19 2Ø Ø
23 4 22 24 Ø
24 8 14 Ø
25 1 Ø
26 2 27 28 29 Ø
27 1 Ø
28 1 Ø
29 1 Ø
3Ø 1Ø 1 26 35 Ø
31 4 3Ø 37 Ø
32 4 3Ø 33 Ø
33 8 37 Ø
34 2 31 32 Ø
35 4 34 36 Ø
36 8 26 Ø
37 1 Ø
38 2 39 4Ø 41 Ø
39 4 7 19 Ø
4Ø 4 19 31 Ø
41 4 7 31 Ø
42 2 43 44 45 Ø
43 4 7 19 Ø
44 4 19 31 Ø
45 4 7 31 Ø
46 2 47 48 49 Ø
47 4 7 19 Ø
48 4 19 31 Ø
49 4 7 31 Ø
5Ø 2 51 52 53 Ø
51 4 8 2Ø Ø
52 4 2Ø 32 Ø
53 4 8 32 Ø
54 2 55 56 57 Ø
55 4 8 2Ø Ø
56 4 2Ø 32 Ø
57 4 8 32 Ø
58 2 59 6Ø 61 Ø
59 4 8 2Ø Ø
6Ø 4 2Ø 32 Ø
61 4 8 32 Ø
-18
```

Figure 6.6 Data File

```
COMMAND EXPECTED                                              Comments
-4 13 7 3 19 2Ø 31 32 33 42 46 5Ø 54 53                   Monitor points
-9 3
13  7  3 19 2Ø 31 32 33 42 46 5Ø 54 53
+--+--+--+--+--+--+--+--+--+--+--+--+-
 Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø                 External input X=0
 Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø                                I1=0

COMMAND EXPECTED
-6 13 25 37                                                            X=1
-9 3
13  7  3 19 2Ø 31 32 33 42 46 5Ø 54 53                                I1=0
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø

COMMAND EXPECTED
-6 3 15 27                                                             X=1
-9 4
13  7  3 19 2Ø 31 32 33 42 46 5Ø 54 53                                I1=1
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 1  1  Ø  1  Ø  1  Ø  1  1  1  Ø  Ø  Ø
 1  1  Ø  1  Ø  1  Ø  1  1  1  Ø  Ø  Ø
 1  1  Ø  1  Ø  1  Ø  1  1  1  Ø  Ø  Ø

COMMAND EXPECTED
-5 3 15 27                                                             X=0
-9 4
13  7  3 19 2Ø 31 32 33 42 46 5Ø 54 53                                I1=1
+--+--+--+--+--+--+--+--+--+--+--+--+-
 Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø
 Ø  Ø  1  Ø  1  Ø  1  Ø  Ø  Ø  1  1  1
 Ø  Ø  1  Ø  1  Ø  1  Ø  Ø  Ø  1  1  1
 Ø  Ø  1  Ø  1  Ø  1  Ø  Ø  Ø  1  1  1
```

Figure 6.7 Results

is illustrated in Figures 6.8 and 6.9. Without exception, all logical faults, irrespective of type or position, are masked by the majority-logic gates. However, in some cases, the fault does not affect the operation of the cell-block. This is identical to the situation encountered earlier when dealing with fail-safe logic design. With the fault-tolerant cell-block fully tested, the next step was to simulate a complete circuit using these blocks in order to induce and observe fault conditions in a practical situation.

6.4. Fault-Tolerant Cell-Block Circuit Design

A data file consisting of three separate fault-tolerant cell-blocks was first drawn up. To avoid confusion, the modes of the second cell-block were numbered in the range 100 - 199 and the third cell-block in the range 200 - 299. That is, since the original flip-flops were numbered 6, 18 and 30, the flip-flops in the second cell-block were numbered 106, 118 and 130 and so on. The connections were then updated so that a circuit was formed, and the data file was named CIRC. DAT. The state graph is identical to that shown in Figure 6.3(b) while the circuit representation using fault-tolerant cell-blocks is illustrated in Figure 6.10.

Normal operation of the circuit was first checked using the simulation program and the results are shown in Figure 6.11. This is satisfactory since the circuit reacts in accordance with the state graph. Using the same technique as before, various faults were simulated. Since it has already been shown that logical faults within the cell-block itself are always masked, the faults, in this case, are restricted to the majority-logic gates.

Comments

```
COMMAND EXPECTED
-2
7 14 Ø
-6 3 15 27 13 25 37
-9 3
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  53
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   Ø   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   Ø   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   Ø   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
```

Update
Node 7 s-a-0

Fault masked by majority-logic gates

```
COMMAND EXPECTED
-2
7 4 6 13 Ø
8 14 Ø
-9 3
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  53
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
```

Update

Node 8 s-a-0

Normal operation

```
COMMAND EXPECTED
-2
8 15 Ø
-9 3
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  53
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   1   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   1   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   1   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
```

Update
Node 8 s-a-1

Fault masked

```
COMMAND EXPECTED
-2
8 4 6 9 Ø
18 14 Ø
-9 3
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  53
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   Ø   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   Ø   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   Ø   Ø   1   Ø   1   1   1   Ø   Ø   Ø
```

Update

Node 18 s-a-0

Fault masked

Figure 6.8 Results

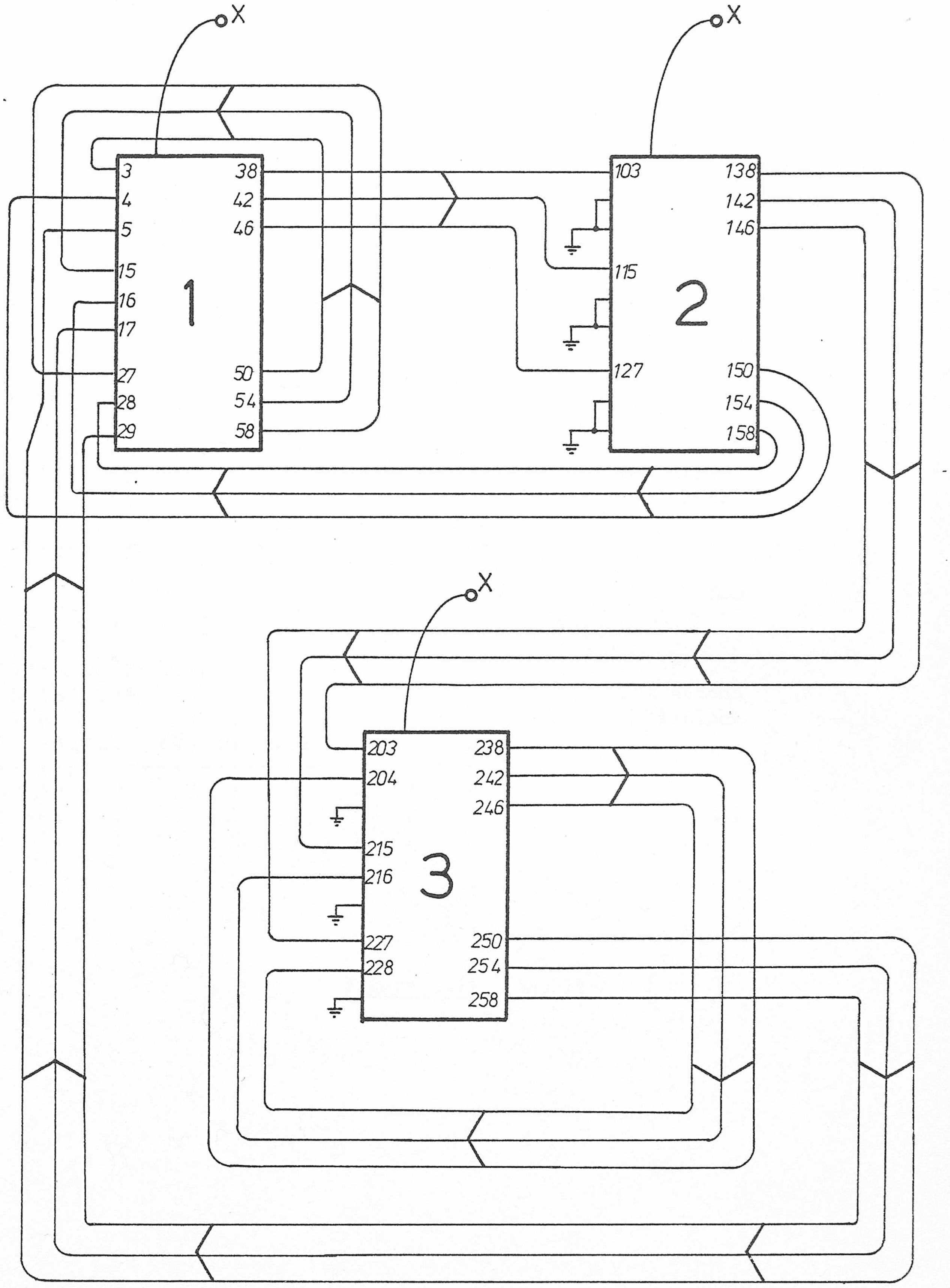
```
COMMAND EXPECTED                                                    Comments
-2                                                                   Update
13  1Ø  1  14  23  Ø
23  14  Ø                                                           Node 23 s-a-0
-9  3
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  58
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  Normal
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  operation
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø

COMMAND EXPECTED
-2                                                                   Update
23  15  Ø                                                           Node 23 s-a-1
-9  4
13   7   8  19  2Ø  31  32  33  42  46  5Ø  54  58
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   Ø   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  Oscillating
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  fault masked
 1   1   Ø   Ø   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø

COMMAND EXPECTED
-2                                                                   Update
23  4  22  24  Ø
36  14  Ø                                                           Node 36 s-a-0
-9  3
13   7   8  19  2Ø  31  32  38  42  46  5Ø  54  58
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  Normal
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  operation
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø

COMMAND EXPECTED
-2                                                                   Update
36  15  Ø                                                           Node 36 s-a-1
-9  4
13   7   8  19  2Ø  31  32  38  42  46  5Ø  54  58
+--+--+--+--+--+--+--+--+--+--+--+--+-
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø                  Oscillating
 1   1   Ø   1   Ø   Ø   Ø   1   1   1   Ø   Ø   Ø                  fault masked
 1   1   Ø   1   Ø   1   Ø   1   1   1   Ø   Ø   Ø
 1   1   Ø   1   Ø   Ø   Ø   1   1   1   Ø   Ø   Ø
```

Figure 6.9 Results



Fault-tolerant representation of state diagram Fig. 6·3 (a)

Figure 6.10

```
COMMAND EXPECTED                                        Comments
-4 6 13 3Ø 1Ø6 113 13Ø 2Ø6 213 23Ø                      Monitor points
-6 6 13 3Ø                                              Set first
-9 3                                                    cell-block
6  13  3Ø  1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø                          X=0
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø

COMMAND EXPECTED
-6 13 25 37 113 125 137 213 225 237                        X=1
-9 4
6  13  3Ø  1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø
Ø   Ø   Ø   1   1   1   Ø   Ø   Ø
Ø   Ø   Ø   Ø   Ø   Ø   1   1   1
Ø   Ø   Ø   Ø   Ø   Ø   1   1   1

COMMAND EXPECTED
-5 2Ø6 213 23Ø                                          Set third
-9 3                                                    cell-block
6  13  3Ø  1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
Ø   Ø   Ø   Ø   Ø   Ø   1   1   1
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø                          X=0
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø

COMMAND EXPECTED
-5 1Ø6 113 13Ø                                          Set second
-9 3                                                    cell-block
6  13  3Ø  1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
Ø   Ø   Ø   1   1   1   Ø   Ø   Ø
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø                          X=0
1   1   1   Ø   Ø   Ø   Ø   Ø   Ø
```

Figure 6.11   Results

The results of these tests are presented in Figure 6.12. The results show that, despite a faulty majority-logic gate, the correct information is still passed on to the following cell-blocks. This occurs since, although the faulty cell-block produces only two correct signals, the majority gate network in the following cell-block masks the fault by producing three correct signals, and so on. As before, the system may function correctly in the presence of a faulty majority-logic gate.

In order to simulate larger systems, the obvious requirement is a data file comprised of many separate fault-tolerant cell-blocks which may be interconnected according to a specified state graph. However, problems arise when more than three cell-blocks are required, due to the storage allocated to the simulator program. Up to 300 modes may be specified, in order, but the total number of list items, excluding the terminating zeroes, must not exceed 800.

However, now that the fault-masking process is fully understood, the original fault-tolerant cell-block may be replaced by a simpler system for simulation purposes.

6.5 <u>Simplified Version of Fault-Tolerant Cell-Block</u>

The circuit illustrated in Figure 6.13 performs exactly the same function as the original cell-block but requires only 50% of the logic. Although this new configuration would not operate in practice since all the majority-logic has been omitted, it is sufficient for simulation purposes.

A circuit which simply cycles through 7 states on the application of an external logical 1 signal and retains the same state when a logical 0 is applied was simulated using the new configuration and the results, showing normal operation, are presented in Figure 6.14.

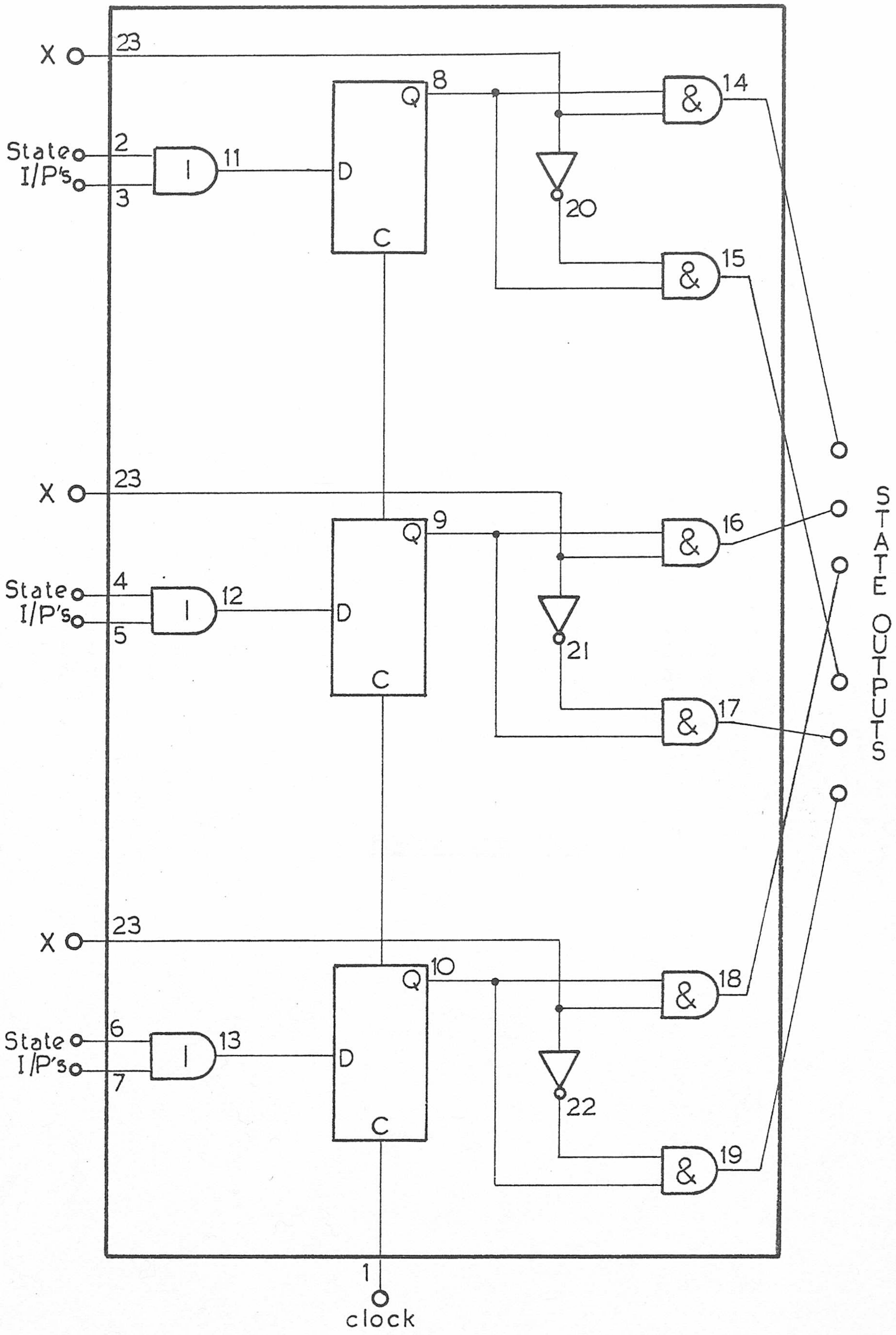
```
COMMAND EXPECTED                                                        Comments
-6 6 13 3Ø 13 25 37 113 125 137 213 225 237
-2                                                                       Update
38 14 Ø                                                                Node 38 s-a-0
-9 4
6   13   3Ø  .1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
1    1    1    Ø    Ø    Ø    Ø    Ø    Ø
Ø    Ø    Ø   $\boxed{Ø}$   1    1    Ø    Ø    Ø                      Fault condition
Ø    Ø    Ø    Ø    Ø    Ø    1    1    1
Ø    Ø    Ø    Ø    Ø    Ø    1    1    1

COMMAND EXPECTED
-2                                                                       Update
38 15 Ø                                                                Node 38 s-a-1
-9 4
6   13   3Ø   1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
1    1    1    Ø    Ø    Ø    Ø    Ø    Ø
Ø    Ø    Ø    1    1    1    Ø    Ø    Ø
Ø    Ø    Ø.  $\boxed{1}$   Ø    Ø    1    1    1                      Fault condition
Ø    Ø    Ø   $\boxed{1}$   Ø    Ø    1    1    1

COMMAND EXPECTED
-2                                                                       Update
38 2 39 4Ø 41 Ø
58 14 Ø                                                                Node 58 s-a-0
-9 3
6   13   3Ø   1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
1    1    1    Ø    Ø    Ø    Ø    Ø    Ø                                Normal
Ø    Ø    Ø    1    1    1    Ø    Ø    Ø                                operation
Ø    Ø    Ø    Ø    Ø    Ø    1    1    1

COMMAND EXPECTED
-2                                                                       Update
5Ø 15 Ø                                                                Node 50 s-a-1
-9 4
6   13   3Ø   1Ø6 113 13Ø 2Ø6 213 23Ø
+--+--+--+--+--+--+--+--+--+--+--+
·1    1    1    Ø    Ø    Ø    Ø    Ø    Ø
$\boxed{1}$    Ø    Ø    1    1    1    Ø    Ø    Ø                       Fault condition
$\boxed{1}$    Ø    Ø    Ø    Ø    Ø    1    1    1
$\boxed{1}$    Ø    Ø    Ø    Ø    Ø    1    1    1
```

Figure 6.12 Results



Improved simulation

Figure 6.13

```
COMMAND EXPECTED
-4 8 9 1Ø 38 39 4Ø 68 69 7Ø 98 99 1ØØ 128 129 13Ø 158 159
   16Ø 188 189 19Ø
-6 11 12 13
-9 3
8 91Ø3839 4Ø6869 7Ø9899 1ØØ128 129 13Ø158 159 16Ø188 189 19Ø
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø

COMMAND EXPECTED
-6 23
-9 9
8 91Ø3839 4Ø6869 7Ø9899 1ØØ128 129 13Ø158 159 16Ø188 189 19Ø
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  1  1  1  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  1  1  1  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  1  1  1   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   1   1   1   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   1   1   1   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   1   1   1
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  1  1  1  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
```

<u>Figure 6.14</u>   <u>Results</u>

Under fault conditions, the system reacts as expected and the results obtained are shown in Figure 6.15. In this case, a fault is not confined to the cell-block in which it occurs but is passed on after each clock pulse. This is because the majority-logic has been omitted and the fault is never masked. Nevertheless, this simplified version of the fault-tolerant cell-block is useful when simulation of larger systems is required.

This concludes the study of the fault-tolerant cell-block and its applications. Once a suitable self-resetting logic element was developed, triple-modular-redundancy was used to produce the final fault-tolerant cell-block. Although it is not viable to construct the cell-block using discrete components, it could be incorporated quite easily into a single chip using either MSI or LSI. In this way, fault-tolerant logic systems could be implemented almost as economically and compactly as ordinary digital systems.

The Reynolds Logic Simulation program was used extensively in this chapter and proved invaluable in the construction and testing of the fault-tolerant cell-block.

Finally, a simplified version of the cell-block was produced which proved useful in the analysis of larger logic systems.

The next chapter deals with the use of the programmable read only memory in both digital and fault-tolerant digital design.

```
COMMAND EXPECTED
-2
14 14 Ø
-9 12
8  9 1Ø38 39 4Ø68 69 7Ø98 99 1ØØ128 129 13Ø158 159 16Ø188 189 19Ø
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
Ø  Ø  Ø  Ø  Ø  Ø  1  1  1  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  1  1  1   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   1   1   1   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   1   1   1   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   1   1   1
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø [Ø] 1  1  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø [Ø] 1  1  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø [Ø] 1  1   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  [Ø]  1   1   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø  [Ø]  1   1   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø  [Ø]  1   1

COMMAND EXPECTED
-2
14 4 3 23 Ø
18 14 Ø
-5 11 12 13 23
-9 4
8  9 1Ø38 39 4Ø68 69 7Ø98 99 1ØØ128 129 13Ø158 159 16Ø188 189 19Ø
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--
1  1  1  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  1  1 [Ø] Ø  Ø  Ø  Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  1  1 [Ø] Ø  Ø  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  Ø  1  1 [Ø]  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
```

<u>Figure 6.15</u>   <u>Results</u>

CHAPTER 7. FAULT-TOLERANT DIGITAL DESIGN USING PROMS

Although many different design techniques have been used throughout this study, they have all been linked by a common factor. Every system has been designed and built to perform a certain function, as specified by a state graph. In this respect, these systems can be considered 'static', since the hard-wired logic involved can perform one function and one function only. To perform a different function using the same logic elements would require a complete re-design, resulting in a totally different hard-wired logic system.

However, consider a system whereby the state assignment is stored within some memory device. By using suitable interfacing and addressing techniques, it is possible for this system to operate in a fashion identical to a conventional logic system. Moreover, this system can be considered 'dynamic', since it can perform an unlimited number of different functions by simply reprogramming the memory, whilst still retaining the original hardware. If the system could then be made fault-tolerant, the result would be a very reliable and versatile digital system using very little hard-wired logic. Such a system, using a reprogrammable read-only-memory, is discussed in detail in this chapter.

7.1 The Read-Only-Memory and its Structure

A read-only-memory (ROM) consists of a matrix of transistors (either bipolar or MOS), which act as memory cells. This matrix is preceded by a decoder which effectively addresses each row of memory cells. As an example, consider a 256 bit ROM arranged in 32 words of 8-bit each. The decoder input is a 5 bit binary select code, and its outputs are the 32 word lines. The matrix consists of 32 bipolar transistors, with each base tied to a different line, and with 8 emitters on each transistor. This type of fixed ROM is programmed once and once only either at manufacture, or by the user. Usually, the customer compiles the truth table he wishes the ROM to satisfy, and a metallisation mask is then made to connect one emmitter of each transistor to the proper output line, or alternatively to leave the emitter floating. Field programmable ROMS are bipolar structures which the user programs by selectively 'blowing' fusable links in memory cells. Both these types have the disadvantage of being non-reprogrammable.

Three types of reprogrammable ROMS are commercially available. The first type is electrically programmed and erased by exposure to ultraviolet (U.V.) light through a window in the package. The U.V. light causes holes and electrons to recombine, clearing the stored charge. The other types are electrically alterable i.e. they may be erased by applying a pulse, usually of 30-40V amplitude to the programming pins. Some devices may be selectively erased, and this type offers significant advantages over the U.V. type in that erasing may be done in circuit in a comparatively shorter time.

7.2 Two Reprogrammable Logic Systems

The general state graph shown in Figure 7.1 is synthesised using two totally different reprogrammable logic systems, and these are discussed it turn.

(i) The first system is illustrated in Figure 7.2(a), while the organisation of the memory information is shown in Figure 7.2(b) The operation of this system is relatively straightforward. The initial address, corresponding to the first state of the graph, is set up in the address buffer, which simply consists of

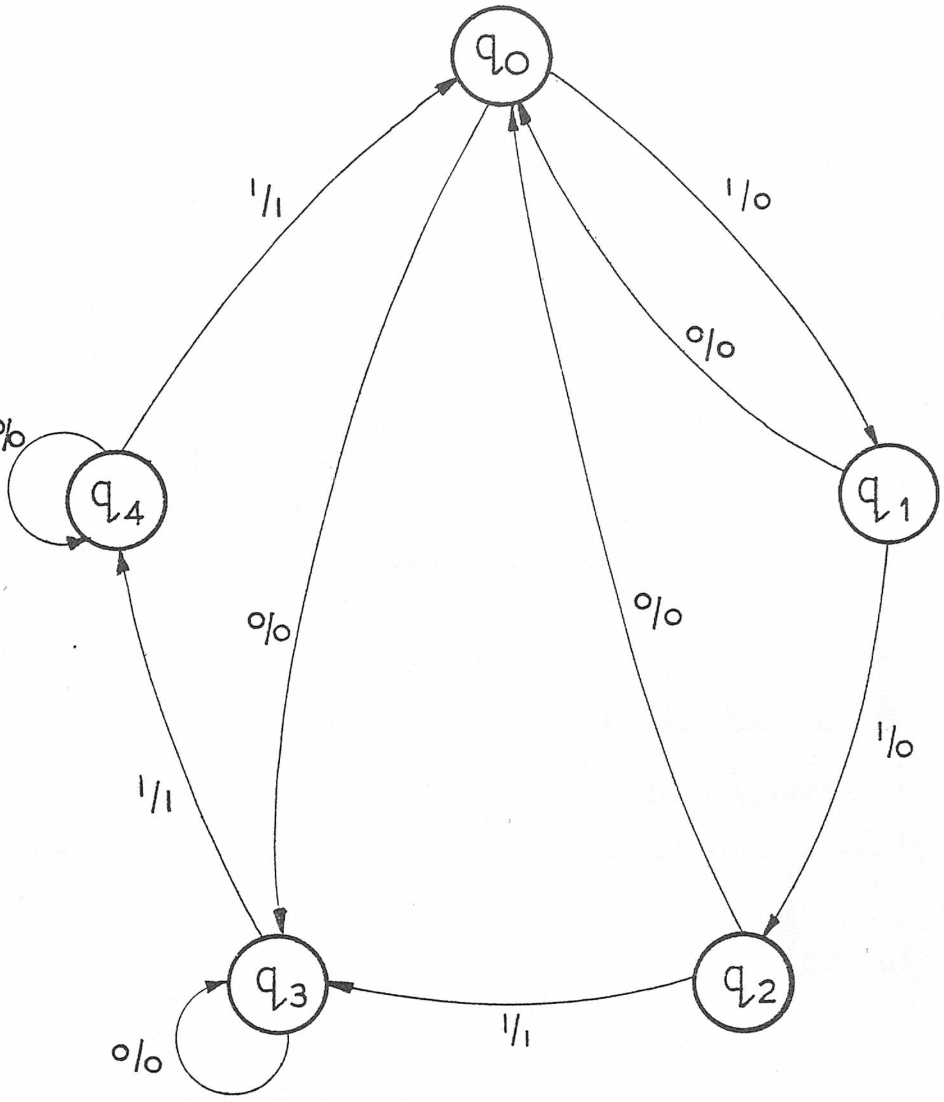
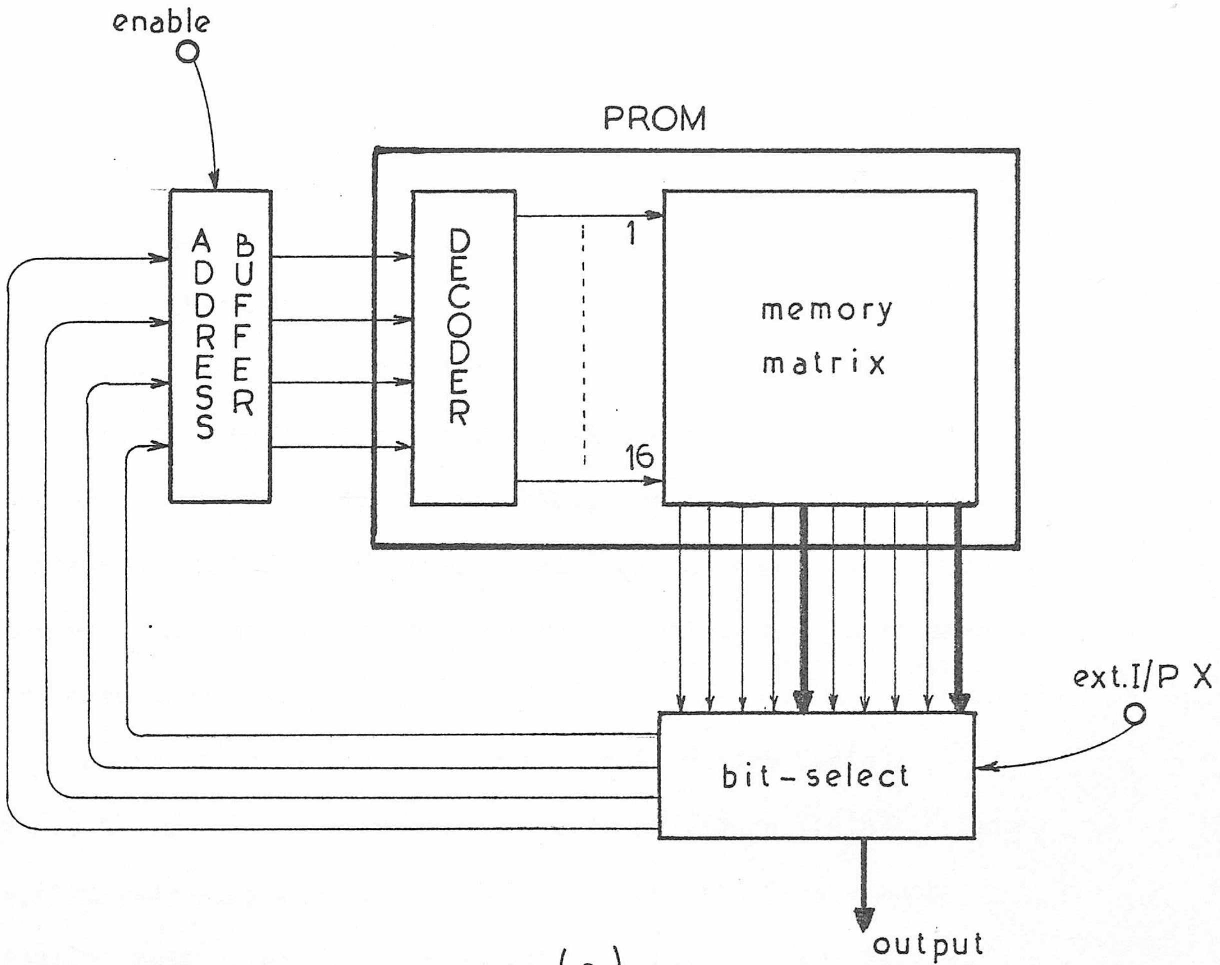


Figure 7.1 State Graph



(a)

current address	next address			
	X = 1		X = 0	
0000	0001	0	0011	0
0001	0010	0	0000	0
0010	0011	1	0000	0
0011	0100	1	0011	0
0100	0000	1	0100	0

(b)

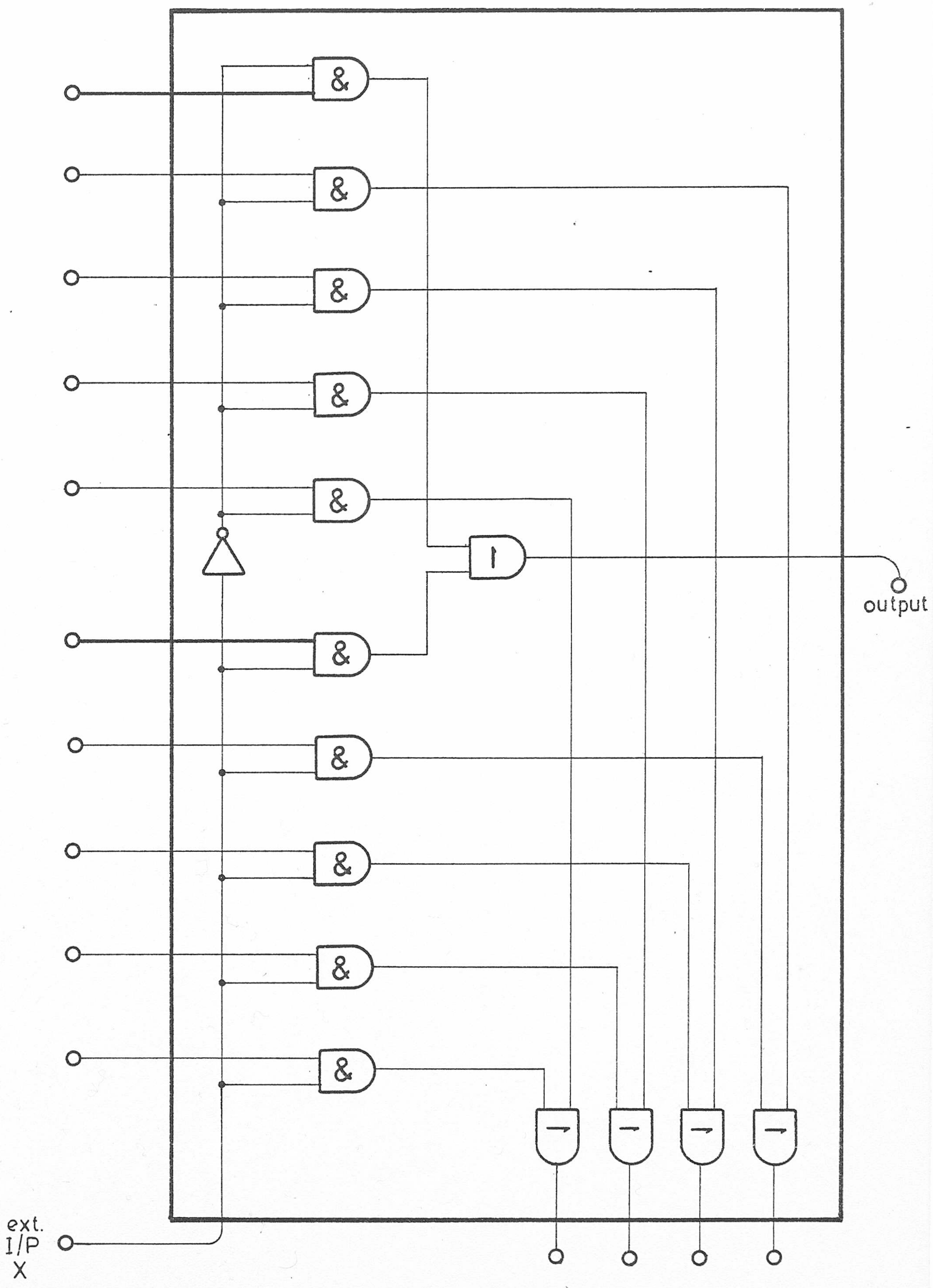
Figure 7.1 System 1

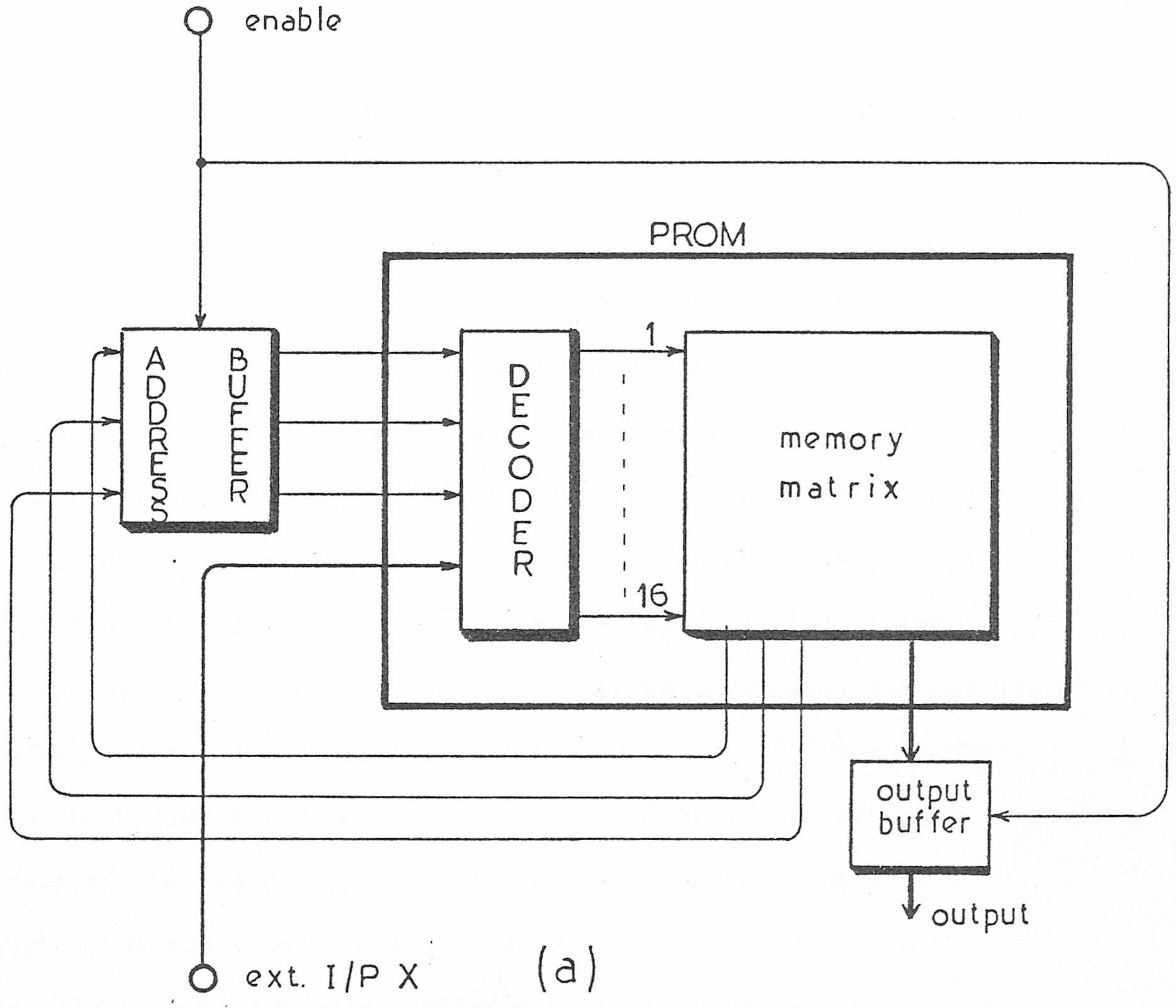
four D-type flip-flops. This address is decoded in the PROM and used to access one of sixteen word lines, which, in this case, is composed of eight data bits and two output bits. It is evident from Figure 7.2 that these ten bits actually represent two different states on the graph, depending on the value of the external input X. Therefore, this information is fed into a 'bit-select' circuit, shown in Figure 7.3, which outputs the correct address to the address buffer. When the system is clocked, this next address appears at the buffer output and accesses a new word line in the PROM, and so on. In this way, operation similar to a conventional logic system is achieved. However, this system is unnecessarily complex, and improvements are discussed below.

(ii) An improved system is illustrated in Figure 7.4(a), while the memory organisation is shown in Figure 7.4(b). This system uses very little external logic and requires a much smaller memory than the system outlined above. The initial address is set up in the buffer, and depending on the value of the external input X, one of two word lines is accessed in the PROM. When the system is clocked, this data, representing the next state, appears at the output of the address buffer, and so on.

Now that a practical system has been developed, the next step is to produce a more reliable version using some method of fault-tolerance. An obvious solution is to use triplication and majority-voting on the system as it stands. In this case, however, this requires the use of three separate memories, each of which has to be reprogrammed every time the complete system

Figure 7.3   Bit-Select Circuit





current address	X	next address	output
000	0	011	0
000	1	001	0
001	0	000	0
001	1	010	0
010	0	000	0
010	1	011	1
011	0	011	0
011	1	100	1
100	0	100	0
100	1	000	1

(b)

Figure 7.4 System 2

is reprogrammed. This is both clumsy and time-consuming. A much better solution is to check and correct the contents of a single memory using some form of error-correcting coding scheme, and triplicate the required hardware. This may be done using the Hamming code, a full description of which is given in Appendix 5.

7.3 Hamming Decoder

A logic circuit, capable of performing this correction procedure, is illustrated in Figure 7.5. If no fault occurs within the memory, the outputs of gates 1,2 and 3 are logical 0, therefore gate 4 is not enabled. The correct information simply passes directly through to the address buffer under these conditions. If, however, a fault occurs in an information bit, the circuit rectifies the situation by changing the logical value of the offending bit. For example, if bit six is incorrect, the outputs of gates 1, 2 and 3 become 110, therefore gates 4 and 8 are enabled, thus providing a logical 1 to gate 14. Now, if the original incorrect bit is a logical 1, the output of gate 14 becomes a logical 0, and vice versa. However, if a fault occurs in a check bit, gate 4 is not enabled and the correct information is again passed on. If the output bit, bit seven, is incorrect, the outputs of gates 1, 2 and 3 become 111, therefore gate 5 is enabled and a similar inversion operation takes place, thus correcting the output information.

By utilising three Hamming decoders, three address buffers and the necessary majority logic, a single-fault-tolerant reprogrammable logic system may be produced in a manner similar to the fault-tolerant cell-block discussed earlier. These systems were constructed and rigourously tested. Without exception, these

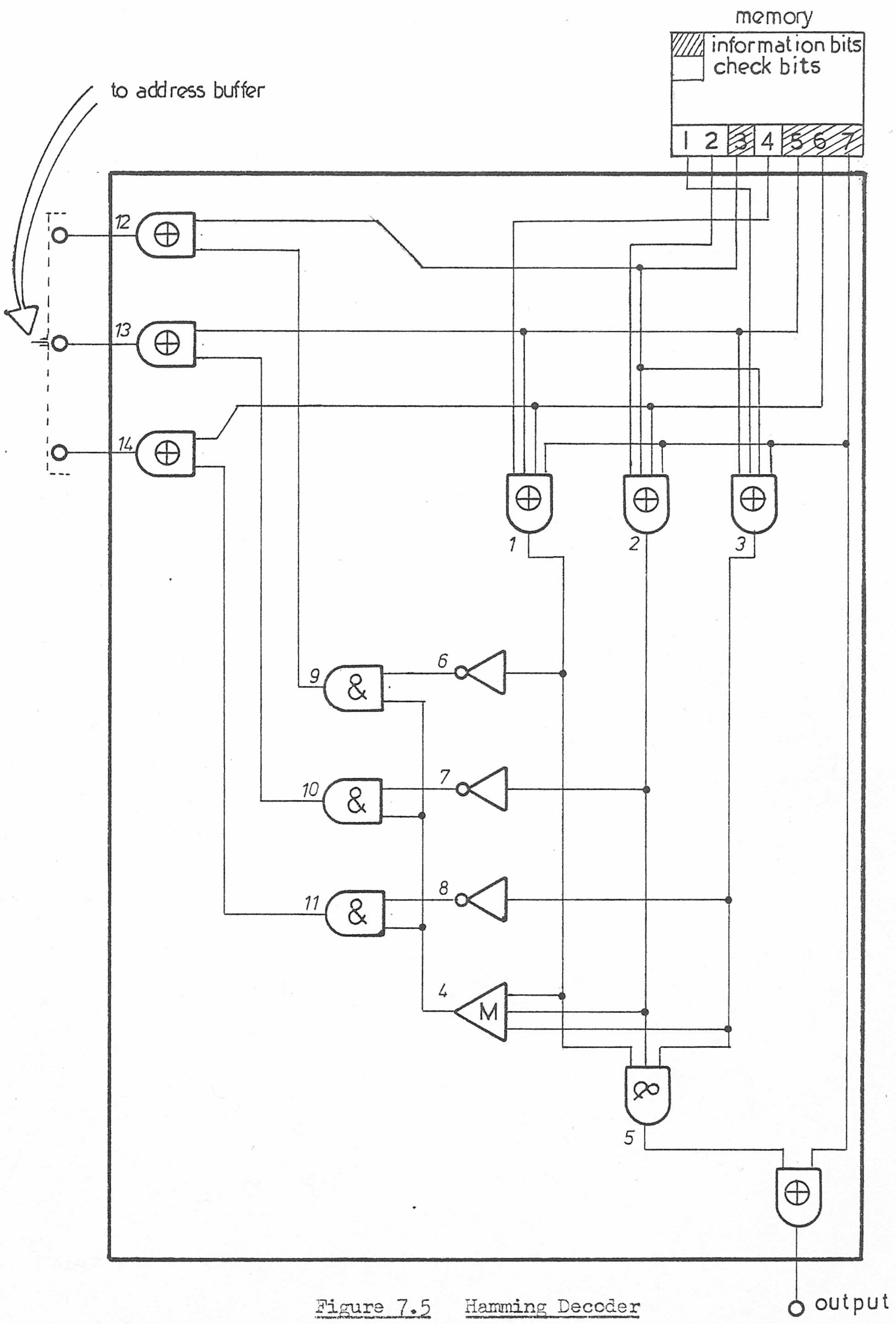


Figure 7.5   Hamming Decoder

circuits performed satisfactorily according to design requirements.

This concludes the study of digital design using programmable read-only-memories. The next chapter presents some overall conclusions and indicates some topics of future research.

CHAPTER 8. CONCLUSIONS

The design of fail-safe and fault-tolerant digital circuitry has been investigated in detail throughout this study. This chapter presents some overall conclusions and indicates topics for future work in this field. First of all, it is worth considering the aims of this study.

8.1. Aims

The aims can be categorised as outlined below :-

(a) To investigate various existing methods of fail-safe and fault-tolerant design with a view to adapting and improving these techniques.

(b) To establish new methods of fail-safe and fault-tolerant digital machine design.

(c) To generally simplify design techniques by the use of original hardware design and with the aid of appropriate software.

These aims have now been achieved.

8.2. Conclusions

A full investigation into the properties and requirements of fail-safe digital circuitry resulted in the development of two new design techniques. In general, both of these techniques required less hardware than existing methods, resulting in increased reliability, while the latter technique produced improved error indication. A computer program was written to aid state assignment selection.

First of all, various existing methods of fault-tolerant digital design were reviewed. The application of error-correcting codes in digital design resulted in the construction of a versatile computer program, capable of producing the design equations of any type of autonomous counter. A practical fault-tolerant cell-block was then developed. This greatly simplified design procedures and, at the same time, introduced a high degree of reliability. Finally, digital systems, utilising reprogrammable read-only-memories, were investigated, again with a view to simplified design, versatility and reliability.

8.3. Further Work

Due to its limited properties, fail-safe digital circuitry is unlikely to be of any benefit to the design engineer in the years ahead. In this respect, it hardly merits further consideration.

In contrast, fault-tolerant design is a very powerful technique, for reasons outlined earlier. In addition, it has become a viable concern, even in large systems, since the advent of integrated electronics. It is obvious, therefore, that further work should be directed towards the design of fault-tolerant circuitry. It was shown earlier that, where applicable, the use of error-correcting codes can produce a reliable system with very few components. By developing new and more versatile codes, it may be possible to synthesise circuits using a minimal number of components, thus improving the reliability, regardless of the size or classification of the system involved. This is a possible topic for future research.

REFERENCES and BIBLIOGRAPHY

1 R.G.Bennet, R.V.Scott, "Recent Developments in the Theory and Practice of Testable Logic Design."
IERE Radio and Electronic Engineer; Vol. 45, No. 11, Nov. 1975, pp. 667-669.

2 S.Das, H.Y.H.Chuang, "A Unified Approach to the Realization of Fail-Safe Sequential Machines."
International Symposium on Fault-Tolerant Computing, Illinois (U.S.A.), 19-21 June 1974.

3 S.Das, "Fault-Tolerant Digital Systems using Fail-Safe Logic."
D.Sc. Dissertation, Department of Electrical Engineering, Washington University, St. Louis; August 1973.

4 I.Aleksander, "Introduction to Logic Circuit Theory."
Engineering Science Monographs; Chapter 3.

5 I.Aleksander, "Introduction to Logic Circuit Theory."
Engineering Science Monographs; Chapter 7.

6 A.D.Friedman, P.R.Menon, "Fault Detection in Digital Circuits."
Prentice-Hall Electrical Engineering Series; Chapter 6.

7 J.Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components."
Automata Studies; No. 34, Princeton University, Princeton, 43-98, 1956.

8 J.G.Tyron, "Quadded Logic."
Redundancy Techniques for Computing Systems, Wilcox and Mann, Spartan Books; pp. 205-228, 1962.

9 T.F.Klaschka, "Two Contributions to Redundancy Theory."
Proc. Eigth Annual Symposium on Switching and Automata Theory; pp. 175-183, 1967.

10 D.B.Armstrong, "A General Method of Applying Error Correction in Synchronous Digital Systems."
BSTJ 40; pp. 577-593, 1961.

11 W.Keister, R.W.Ketchledge, H.E.Vaughan, "No. 1 ESS; System Organisation and Objective."
BSTJ 43; pp. 1831-1844, 1964.

12 A.Avizienis et al, "The STAR Computer; An Investigation of the Theory and Practice of Fault-Tolerant Digital Machines."
IEEE Trans. on Comp.; Vol. C-20, No. 11, pp. 1337-1352, 1971.

13 W.W.Peterson, "Error-Correcting Codes."
Cambridge Mass. M.I.T. Press, Chapter 3.

14 I.S.Reed, "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme."
IRE Trans. Inform. Theory; Vol. IT-4, pp. 38-49, 1954.

15 B.S.Walker, "The Design of Sequential Logic Circuits."
IERE Radio and Electronic Engineer; Vol. 44, No. 1, Jan. 1974, pp. 45-49

16 J.S.Reynolds, "A Conversational Logic Simulator for Use with a Time-Sharing Computer."
GEC-AEI(Electronics)Ltd., Applied Electronics Laboratories, Portsmouth.

Appendix 1.   Computer Program

```
  1 PRI"             PAIRWISE INCOMPARABLE BINARY NUMBERS"\PRI\PRI
  5 PRI "HOW MANY VARIABLES";\INP C\PRI\N=Ø
 15 FOR K=Ø TO C STEP 1
 2Ø N=2↑K-1\P=Ø
 25 GOSUB 2ØØ
 3Ø GOSUB 3ØØ
 35 X=Ø\Y=Ø
 4Ø FOR I=C TO 1 STEP -1
 45 IF A(I)<=B(I) THEN 55
 5Ø X=1
 55 NEX I
 6Ø FOR I=C TO 1 STEP -1
 65 IF B(I)<=A(I) THEN 75
 7Ø Y=1
 75 NEX I
 8Ø IF X+Y=2 THEN 1ØØ
 85 IF P=2↑C-1 THEN 145
 9Ø P=P+1
 95 GOTO 3Ø
1ØØ IF B(9)+B(8)+B(7)+B(6)+B(5)+B(4)+B(3)+B(2)+B(1)<>K THE 125
1Ø5 FOR I=C TO 1 STEP -1
11Ø PRI B(I);
115 NEX I
12Ø PRI
125 IF P=2↑C-1 THEN 14Ø
13Ø P=P+1\GOTO 3Ø
14Ø PRI
145 NEX K
15Ø END
2ØØ I=C\V=N\PRI
2Ø5 M=2↑(I-1)
21Ø IF M<=N THEN 22Ø
215 GOTO 245
22Ø A(I)=1
225 N=N-M
23Ø I=I-1
235 IF I<=Ø THEN 265
24Ø GOTO 2Ø5
245 A(I)=Ø
25Ø I=I-1
255 IF I<=Ø THEN 265
26Ø GOTO 2Ø5
265 FOR J=C TO 1 STEP -1
27Ø PRI A(J);
275 NEX J
28Ø PRI
285 N=V
29Ø RETURN
3ØØ I=C\S=P
3Ø5 M=2↑(I-1)
31Ø IF M<=P THEN 32Ø\GOTO 345
32Ø B(I)=1
325 P=P-M
33Ø I=I-1
335 IF I<=Ø THEN 365\GOTO 3Ø5
345 B(I)=Ø
35Ø I=I-1
355 IF I<=Ø THEN 365
36Ø GOTO 3Ø5
365 P=S
37Ø RETURN
```

Appendix 2. Computer Program

```
  1Ø PRI\PRI"VAR AND REF NO.";\INP C,D\K=Ø\PRI\PRI\P=D\A(K)=D
  47 GOSUB 1ØØØ
  5Ø FOR J=C TO 1 STEP -1\L(J)=B(J)\NEX J
  85 PRI TAB(27);"REFERENCE=";P\PRI\PRI"FIRST GROUP"\PRI
  9Ø FOR M=1 TO 2↑C-1\P=M\GOSUB 1ØØØ
 12Ø X=Ø\Y=Ø\FOR I=C TO 1 STEP -1\IF B(I)<=L(I) THEN 16Ø
 15Ø X=1
 16Ø NEX I
 165 FOR I=C TO 1 STEP -1\IF L(I)<=B(I) THEN 19Ø
 17Ø Y=1
 19Ø NEX I
 2ØØ IF X+Y=2 THEN 22Ø\GOTO 27Ø
 22Ø K=K+1\A(K)=M\PRI A(K);"\";
 27Ø NEX M
 275 PRI\PRI
 28Ø PRI"SECOND GROUP"\PRI\N=Ø\Q1=Ø\FOR F=1 TO K\P=A(F)\Z7=Ø
 283 GOSUB 1ØØØ
 314 FOR I=C TO 1 STEP -1\R(I)=B(I)\NEX I
 32Ø FOR G=1 TO K\P=A(G)\GOSUB 1ØØØ
 33Ø S=Ø\T=Ø\FOR I=C TO 1 STEP -1
 4ØØ IF R(I)<=B(I) THEN 42Ø
 41Ø S=1
 42Ø NEX I
 43Ø FOR I=C TO 1 STEP -1
 44Ø IF B(I)<=R(I) THEN 46Ø
 45Ø T=1
 46Ø NEX I
 47Ø IF S+T=2 THEN 49Ø\GOTO 54Ø
 49Ø N=N+1\Q1=Q1+1\V(N)=A(G)\PRI P;\Z7=1
 54Ø NEX G
 541 IF Z7<>1 THEN 542\GOTO 543
 542 PRI"NONE";
 543 PRI"\";\A1(F)=N\NEX F
 55Ø Z5=Ø\PRI
 555 PRI\N=1\Q1=Ø\Z1=Ø\X1=Ø\PRI"THIRD GROUP"\PRI\W=Ø\FOR F=1 TO
     Y1=Ø
 558 Z5=Z5+1\IF Z5>=2 THEN 58Ø
 56Ø IF A1(I)<1 THEN F=F+1
 58Ø FOR N=N TO A1(F)\Y1=Y1+1\P=V(N)\GOSUB 1ØØØ
 6Ø5 Z1=Z1+1\Z8=Ø
 61Ø FOR I=C TO 1 STEP -1\U(I)=B(I)\NEX I
 635 Q1=X1\FOR Q1=Q1+1 TO A1(F)\P=V(Q1)\GOSUB 1ØØØ
 67Ø C1=Ø\C2=Ø\FOR I=C TO 1 STEP -1\IF U(I)<=B(I) THEN 71Ø
 7ØØ C1=1
 71Ø NEX I
 72Ø FOR I=C TO 1 STEP -1\IF B(I)<=U(I) THEN 75Ø
 74Ø C2=1
 75Ø NEX I
 76Ø IF C1+C2=2 THEN 78Ø\GOTO 84Ø
 78Ø W=W+1\V1(W)=V(Q1)\PRI P;\Z8=1
 84Ø NEX Q1
 841 IF Z8<>1 THEN 842\GOTO 843
 842 PRI"NONE";
 843 PRI"\";\B1(N)=W\NEX N
 865 X1=X1+Y1\NEX F
 986 PRI\END
1ØØØ J=C\Z=P
1Ø2Ø Q=2↑(J-1)\IF Q<=P THEN 1Ø5Ø\GOTO 11ØØ
1Ø5Ø B(J)=1\P=P-Q\J=J-1\IF J<=Ø THEN 114Ø\GOTO 1Ø2Ø
11ØØ B(J)=Ø\J=J-1
112Ø IF J<=Ø THEN 114Ø\GOTO 1Ø2Ø
114Ø P=Z
115Ø RETURN
```

Appendix 3. Computer Program

```
      DIMENSION IC(2Ø),ID(2Ø),IP(2Ø),M(2Ø),IONE(2Ø),ITWO(2Ø),
     *ITHR(2Ø),IFOU(2Ø),IFIV(2Ø),ISIX(2Ø),ISEV(2Ø),IEIG(2Ø),
     *ININ(2Ø),ITEN(2Ø),JG(2Ø),JN(2Ø),JL(2Ø),JA(2Ø,2Ø),
     *JSA(2Ø,2Ø),JR(2Ø),NY(1Ø)
      DATA KA5/2HA5/,KA4/2HA4/.KA3/2HA3/,KA2/2HA2/,KA1/2HA1/,
      DATA KA5P/3HA5'/,KA4P/3HA4'/,KA3P/3HA3'/,KA2P/3HA2'/,
      DATA KA1P/3HA1'/
      WRITE(5,3ØØ1)
 3ØØ1 FORMAT(1H ,38HNO. OF VARIABLES,NO. OF TERMS AND TYPE)
      READ(5,1ØØ1)IP4,IY8,IZ1
 1ØØ1 FORMAT(I1,I2,I1)
      IVAR=IP4-2
      WRITE(5,3ØØ3)
 3ØØ3 FORMAT(1H ,33HINPUT REED-MULLER MATRIX REQUIRED)
      GO TO(1,2,3),IVAR
    1 READ(5,1ØØ2)JX1,JX2,JX3,JX4,JX5,JX6
 1ØØ2 FORMAT(6I1)
      GO TO 4
    2 READ(5,1ØØ3)JX1,JX2,JX3,JX4,JX5,JX6,JX7,JX8
 1ØØ3 FORMAT(8I1)
      GO TO 4
    3 READ(5,1ØØ4)JX1,JX2,JX3,JX4,JX5,JX6,JX7,JX8,JX9,JX1Ø
 1ØØ4 FORMAT(1ØI1)
    4 JS=-1
      JC1=Ø
      WRITE(5,3ØØ2)
 3ØØ2 FORMAT(1H ,17HSTATE ASSIGNMENT?)
      READ(5,1ØØ5)(IC(J),J=1,IY8)
 1ØØ5 FORMAT(31I1)
      WRITE(5,1ØØ6)
 1ØØ6 FORMAT(1H ,9HINFO BITS,6X,1ØHCHECK BITS,6X,14HCHANGE
     *OP FORM)
      WRITE(5,1ØØ7)
 1ØØ7 FORMAT(1H ,31X,9HDELTA A'S,6X,9HDELTA B'S)
      IL=-1
      DO 1ØØ J=1,IY8
      IP1=IC(J)
      IL=IL+1
      CALL BINARY(IP1,IP4,IP)
      IF(IL.EQ.Ø)GO TO 5
      IF(IZ1.EQ.2)GO TO 6
      DO 9 K1=1,IP4
      IF(ID(K1).EQ.IP(K1))GO TO 8
      M(K1)=1
      GOTO 9
    8 M(K1)=Ø
    9 CONTINUE
      GOTO 5
    6 DO 11 K1=1,IP4
      M(K1)=IP(K1)
   11 CONTINUE
    5 GO TO(12,13,14),IVAR
   12 JS=JS+1
      IONE(JS)=M(IP4)
      ITWO(JS)=M(IP4-1)
      ITHR(JS)=M(IP4-2)
      GO TO 15
   13 JS=JS+1
      IONE(JS)=M(IP4)
      ITWO(JS)=M(IP4-1)
      ITHR(JS)=M(IP4-2)
      IFOU(JS)=M(IP4-3)
      GO TO 15
   14 JS=JS+1
      IONE(JS)=M(IP4)
      ITWO(JS)=M(IP4-1)
      ITHR(JS)=M(IP4-2)
      IFOU(JS)=M(IP4-3)
      IFIV(JS)=M(IP4-4)
   15 IF(IP(JX1).EQ.IP(JX2))GO TO 17
      JG(IP4)=1
      GO TO 16
   17 JG(IP4)=Ø
   16 IF(IP(JX3).EQ.IP(JX4))GO TO 18
      JG(IP4-1)=1
      GO TO 19
   18 JG(IP4-1)=Ø
   19 IF(IP(JX5).EQ.IP(JX6))GO TO 21
      JG(IP4-2)=1
      GO TO 2Ø
   21 JG(IP4-2)=Ø
   2Ø IF(IP4.EQ.3)GO TO 3Ø
      IF(IP(JX7).EQ.IP(JX8))GO TO 22
      JG(IP4-3)=1
      GO TO 23
   22 JG(IP4-3)=Ø
   23 IF(IP4.EQ.4)GO TO 3Ø
      IF(IP(JX9).EQ.IP(JX1Ø))GO TO 25
      JG(IP4-4)=1
      GO TO 3Ø
   25 JG(IP4-4)=Ø
   3Ø IF(IL.EQ.Ø)GO TO 6Ø
      IF(IZ1.EQ.2)GO TO 4Ø
      DO 35 K1=1,IP4
      IF(JL(K1).EQ.JG(K1))GO TO 32
      JN(K1)=1
      GO TO 35
   32 JN(K1)=Ø
   35 CONTINUE
      GO TO 45
   4Ø DO 41 K1=1,IP4
      JN(K1)=JG(K1)
   41 CONTINUE
   45 GO TO(46,47,48),IVAR
   46 JC1=JC1+1
      ISIX(JC1)=JN(IP4)
      ISEV(JC1)=JN(IP4-1)
      IEIG(JC1)=JN(IP4-2)
      GO TO 6Ø
   47 JC1=JC1+1
      ISIX(JC1)=JN(IP4)
      ISEV(JC1)=JN(IP4-1)
      IEIG(JC1)=JN(IP4-2)
      ININ(JC1)=JN(IP4-3)
      GO TO 6Ø
   48 JC1=JC1+1
```

```
      ISIX(JC1)=JN(IP4)
      ISEV(JC1)=JN(IP4-1)
      IEIG(JC1)=JN(IP4-2)
      ININ(JC1)=JN(IP4-3)
      ITEN(JC1)=JN(IP4-4)
   6Ø DO 61 K1=1,IP4
      ID(K1)=IP(K1)
      JL(K1)=JG(K1)
   61 CONTINUE
      IF(IL/EQ/Ø)GO TO 65
      GO TO(73,74,75),IVAR
   73 WRITE(5,2ØØØ)IP(3),IP(2),IP(1),JG(3),JG(2),JG(1),
     *M(3),M(2),M(1),JN(3),JN(2),JN(1)
 2ØØØ FORMAT(1H ,3I2,1ØX,3I2,6X,3I2,8X,3I2)
      GO TO 1ØØ
   74 WRITE(5,2ØØ1)IP(4),IP(3),IP(2),IP(1),JG(4),JG(3),
     *JG(2),JG(1),M(4),M(3),M(2),M(1),JN(4),JN(3),JN(2),
     *JN(1)
 2ØØ1 FORMAT(1H ,4I2,1ØX,4I2,6X,4I2,3X,4I2)
      GO TO 1ØØ
   75 WRITE(5,2ØØ2)IP(5),IP(4),IP(3),IP(2),IP(1),JG(5),
     *JG(4),JG(3),JG(2),JG(1),M(5),M(4),M(3),M(2),M(1),
     *JN(5),JN(4),JN(3),JN(2),JN(1)
 2ØØ2 FORMAT(1H ,5I2,1ØX,5I2,6X,5I2,8X,5I2)
      GO TO 1ØØ
   65 GO TO(7Ø,71,72),IVAR
   7Ø WRITE(5,1Ø12)IP(3),IP(2),IP(1),JG(3),JG(2),JG(1)
 1Ø12 FORMAT(1H ,3I2,1ØX,3I2)
      GO TO 1ØØ
   71 WRITE(5,1Ø13)IP(4),IP(3),IP(2),IP(1),JG(4),JG(3),
     *JG(2),JG(1)
 1Ø13 FORMAT(1H ,4I2,15X,4I2)
      GO TO 1ØØ
   72 WRITE(5,1Ø14)IP(5),IP(4),IP(3),IP(2),IP(1),JG(5),
     *JG(4),JG(3),JG(2),JG(1)
 1Ø14 FORMAT(1H ,5I2,1ØX,5I2)
  1ØØ CONTINUE
  3ØØ GO TO(31Ø,32Ø,33Ø),IVAR
  31Ø NY(1)=KA3
      NY(2)=KA2
      NY(3)=KA1
      NY(4)=KA3P
      NY(5)=KA2P
      NY(6)=KA1P
      GO TO 9ØØ
  32Ø NY(1)=KA4
      NY(2)=KA3
      NY(3)=KA2
      NY(4)=KA1
      NY(5)=KA4P
      NY(6)=KA3P
      NY(7)=KA2P
      NY(8)=KA1P
      GO TO 9ØØ
  33Ø NY(1)=KA5
      NY(2)=KA4
      NY(3)=KA3
      NY(4)=KA2
      NY(5)=KA1
      NY(6)=KA5P
      NY(7)=KA4P
      NY(8)=KA3P
      NY(9)=KA2P
      NY(1Ø)=KA1P
  9ØØ MCQ=1
  4ØØ N=Ø
      WRITE(5,3ØØ6)
 3ØØ6 FORMAT(1H )
      L3=Ø
      IR5=IY8-1
      DO 13Ø MS=1,IR5
      ML5=IP4
      GO TO(1Ø1,1Ø2,1Ø3),IVAR
  1Ø1 GO TO(11Ø,112,114,12Ø,122,124),MCQ
  1Ø2 GO TO(11Ø,112,114,116,12Ø,122,124,126),MCQ
  1Ø3 GO TO(11Ø,112,114,116,118,12Ø,122,124,126,128),MCQ
  11Ø IF(IONE(MS).EQ.1)GO TO 129
      GO TO 13Ø
  112 IF(ITWO(MS).EQ.1)GO TO 129
      GO TO 13Ø
  114 IF(ITHR(MS).EQ.1)GO TO 129
      GO TO 13Ø
  116 IF(IFOU(MS).EQ.1)GO TO 129
      GO TO 13Ø
  118 IF(IFIV(MS).EQ.1)GO TO 129
      GO TO 13Ø
  12Ø IF(ISIX(MS).EQ.1)GO TO 129
      GO TO 13Ø
  122 IF(ISEV(MS).EQ.1)GO TO 129
      GO TO 13Ø
  124 IF(IEIG(MS).EQ.1)GO TO 129
      GO TO 13Ø
  126 IF(ININ(MS).EQ.1)GO TO 129
      GO TO 13Ø
  128 IF(ITEN(MS).EQ.1)GO TO 129
      GO TO 13Ø
  129 N=N+1
      IP1=IC(MS)
      L3=L3+1
      CALL BINARY(IP1,IP4,IP)
      DO 132 JC1=1,IP4
      JA(N,JC1)=IP(ML5)
      JSA(N,JC1)=JA(N,JC1)
      ML5=ML5-1
      JR(N)=JR(N)+JA(N,JC1)
  132 CONTINUE
  13Ø CONTINUE
      GO TO(14Ø,142,143),IVAR
  14Ø GO TO(15Ø,153,156,165,168,172),MCQ
  142 GO TO(147,15Ø,153,156,162,168,172),MCQ
  143 GO TO(144,147,15Ø,153,156,159,162,165,168,172),MCQ
  144 IF(N.GT.Ø)WRITE(5,1Ø2Ø)
```

```
1Ø2Ø FORMAT(1H ,9HDELTA A5=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 147 IF(N.GT.Ø)WRITE(5,1Ø3Ø)
1Ø3Ø FORMAT(1H ,9HDELTA A4=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 15Ø IF(N.GT.Ø)WRITE(5,1Ø4Ø)
1Ø4Ø FORMAT(1H ,9HDELTA A3=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 153 IF(N.GT.Ø)WRITE(5,1Ø5Ø)
1Ø5Ø FORMAT(1H ,9HDELTA A2=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 156 IF(N.GT.Ø)WRITE(5,1Ø6Ø)
1Ø6Ø FORMAT(1H ,9HDELTA A1=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 159 IF(N.GT.Ø)WRITE(5,1Ø7Ø)
1Ø7Ø FORMAT(1H ,9HDELTA B5=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 162 IF(N.GT.Ø)WRITE(5,1Ø8Ø)
1Ø8Ø FORMAT(1H ,9HDELTA B4=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 165 IF(N.GT.Ø)WRITE(5,1Ø9Ø)
1Ø9Ø FORMAT(1H ,9HDELTA B3=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 168 IF(N.GT.Ø)WRITE(5,11ØØ)
11ØØ FORMAT(1H ,9HDELTA B2=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 172 IF(N.GT.Ø)WRITE(5,111Ø)
111Ø FORMAT(1H ,9HDELTA B1=,$)
     IF(N.EQ.IY3-1)GO TO 18Ø
     GO TO 2ØØ
 18Ø WRITE(5,112Ø)
112Ø FORMAT(' 1')
 2ØØ IF(N.EQ.IT3-1)GO TO 25Ø
     IF(N.GT.Ø)CALL MINIM(IP4,L3,JR,JA,JSA,ID,NY)
 25Ø MCQ=MCQ+1
     IF(MCQ.EQ.2*IP4+1)GO TO 26Ø
     GO TO 4ØØ
 26Ø STOP
     END
     SUBROUTINE MINIM(IP4,L3,JR,JA,JSA,ID,NY)
     DIMENSION ID(2Ø),IB(2Ø,2Ø),IW(2Ø),JR(2Ø),JA(2Ø,2Ø),
    *IL(2Ø),IT(2Ø),IE(2Ø,2Ø),JSA(2Ø,2Ø),IU(2Ø),JM(2Ø),
    *JIF(2Ø),NY(2Ø)
     JV=1
     JD=IP4
     JK1=L3
     JG=JK1
```

```
      DO 410 N=1,JK1
      ID(N)=0
410   CONTINUE
      DO 420 J=1,10
      DO 420 IC1=1,10
      IB(J,IC1)=0
420   CONTINUE
425   J=1
      DO 430 M=1,10
      JIF(J)=0
430   CONTINUE
      DO 500 N=1,JK1
      DO 450 IP1=1,JK1
      IF(JR(IP1).EQ.JR(N)+1)GO TO 431
      GO TO 450
431   JQ3=0
      DO 434 IC1=1,JD
      IF(JA(N,IC1).EQ.JA(IP1,IC1))GO TO 434
      JQ3=JQ3+1
434   CONTINUE
      IF(JQ3.NE.1)GO TO 450
      DO 440 IC1=1,JD
      IF(JA(N,IC1).EQ.JA(IP1,IC1))GO TO 438
      IB(J,IC1)=2
438   IF(JA(N,IC1).NE.JA(IP1,IC1))GO TO 440
      IB(J,IC1)=JA(N,IC1)
440   CONTINUE
      IL(IP1)=1
      IL(N)=1
      J=J+1
450   CONTINUE
      IF(IL(N).GT.0)GO TO 500
      IF(N.EQ.1)GO TO 470
      NN=N-1
      DO 468 JF=1,NN
      MS=0
      DO 465 IC1=1,JD
      IF(JA(JF,IC1).NE.JA(N,IC1))GO TO 465
      MS=MS+1
465   CONTINUE
470   DO 480 IC1=1,JD
      IE(JV,IC1)=JA(N,IC1)
480   CONTINUE
      JV=JV+1
500   CONTINUE
      I=0
      DO 520 IP8=1,J
      IT(IP8)=0
      DO 510 IC1=1,JD
      IT(IP8)=IT(IP8)+IB(IP8,IC1)
510   CONTINUE
      I=I+IT(IP8)
520   CONTINUE
      IF(I.NE.0)GO TO 521
      GO TO 550
521   DO 522 N=1,J
```

```
      DO 522 IC1=1,JD
      JA(N,IC1)=IB(N,IC1)
      IB(N,IC1)=Ø
  522 CONTINUE
      DO 523 N=1,JK1
      IL(N)=Ø
  523 CONTINUE
      JJ=J-1
      DO 524 N=1,JJ
      JR(N)=Ø
      DO 524 IC1=1,JD
      JR(N)=JR(N)+JA(N,IC1)
  524 CONTINUE
      JK1=J-1
      GO TO 425
  55Ø JV=JV-1
      DO 56Ø M=1,JG
      IF(ID(M).EQ.1)GO TO 56Ø
      DO 559 N=1,JV
      DO 555 IC1=1,JD
      IF(IE(N,IC1).EQ.2)GO TO 555
      IF(IE(N,IC1).NE.JSA(M,IC1))GO TO 56Ø
  555 CONTINUE
      IB(N,M)=1
  559 CONTINUE
  56Ø CONTINUE
      DO 6ØØ M=1,JG
      IF(ID(M).EQ.1)GO TO 6ØØ
      MZ3=Ø
      DO 57Ø N=1,JV
      IF(IB(N,M).NE.1)GO TO 57Ø
      MZ3=MZ3+1
      IF(MZ3.GT.1)GO TO 6ØØ
      JF=N
  57Ø CONTINUE
      IF(MZ3.NE.1)GO TO 571
      JIF(JF)=1
  571 DO 59Ø IC1=1,JD
      IF(IE(JF,IC1).EQ.1)WRITE(5,113Ø)NY(IC1)
 113Ø FORMAT(1H ,A3,$)
      IVAR=IP4-2
      GO TO(53Ø,533,536),IVAR
  53Ø IF(IE(JF,IC1).EQ.Ø)WRITE(5,114Ø)NY(IC1+3)
 114Ø FORMAT(1H ,A3,$)
      GO TO 59Ø
  533 IF(IE(JF,IC1).EQ.Ø)WRITE(5,115Ø)NY(IC1+4)
 115Ø FORMAT(1H ,A3,$)
      GO TO 59Ø
  536 IF(IE(JF,IC1).EQ.Ø)WRITE(5,116Ø)NY(IC1+5)
 116Ø FORMAT(1H ,A3,$)
  59Ø CONTINUE
      IF(IU(JF).EQ.JG)GO TO 6ØØ
      WRITE(5,117Ø)
 117Ø FORMAT(1H ,1H+,$)
  6ØØ CONTINUE
      DO 63Ø N=1,JV
      IF(JIF(N).EQ.1)GO TO 630
      DO 620 M=1,JG
      IF(ID(M).EQ.1)GO TO 620
      IF(IB(N,M).EQ.0)GO TO 620
      DO 610 IN=1,JV
      IF(JIF(IN).NE.1)GO TO 610
      IF(IB(IN,M).EQ.1)GO TO 620
  610 CONTINUE
      GO TO 630
  620 CONTINUE
      JIF(N)=1
  630 CONTINUE
      DO 640 N=1,JV
      IF(JIF(N).NE.1)GO TO 645
  640 CONTINUE
      GO TO 800
  645 DO 652 N=1,JV
      IF(JIF(N).NE.1)GO TO 652
      DO 651 M=1,JG
      IF(ID(M).EQ.1)GO TO 647
      IF(IB(N,M).NE.1)GO TO 651
  647 DO 650 IN=1,JV
      IB(IN,M)=1
  650 CONTINUE
  651 CONTINUE
  652 CONTINUE
      MZ4=0
  653 DO 660 N=1,JV
      JH(N)=0
      IF(JIF(N).EQ.1)GO TO 660
      DO 655 M=1,JG
      IF(IW(M).EQ.1)GO TO 655
      JH(N)=JH(N)+IB(N,M)
  655 CONTINUE
      IF(JH(N).LE.JH(N-1)GO TO 660
      JF=N
  660 CONTINUE
      DO 665 M=1,JG
      IF(IW(M).EQ.1)GO TO 664
      IF(IB(N,M).NE.1)GO TO 664
      IW(M)=1
  664 IU(JF)=IU(JF)+IW(M)
  665 CONTINUE
      IF(IU(JF).LT.MZ4)GO TO 653
      DO 680 IC1=1,JD
      IF(IE(JF,IC1).EQ.1)WRITE(5,1200)NY(IC1)
 1200 FORMAT(1H ,A3,$)
      IVAR=IP4-2
      GO TO(670,673,676),IVAR
  670 IF(IE(JF,IC1).EQ.0)WRITE(5,1220)NY(IC1+3)
 1220 FORMAT(1H ,A3,$)
      GO TO 680
  673 IF(IE(JF,IC1).EQ.0)WRITE(5,1230)NY(IC1+4)
 1230 FORMAT(1H ,A3,$)
      GO TO 680
  676 IF(IE(JF,IC1).EQ.0)WRITE(5,1240)NY(IC1+5)
```

```
1240 FORMAT(1H ,A3,$)
 680 CONTINUE
     IF(IU(JF).EQ.JG)GO TO 682
     WRITE(5,1250)
1250 FORMAT(1H ,1H+,$)
 682 MZ4=IU(JF)
     IF(IU(JF).NE.JG)GO TO 653
 800 DO 801 N=1,10
     JR(N)=0
 801 CONTINUE
     RETURN
     END
     SUBROUTINE BINARY(IP1,IP4,IP)
     DIMENSION IP(20)
     II=IP4
 840 MQ3=2**(II-1)
     IF(II.EQ.1)MQ3=1
     IF(MQ3.LE.IP1)GO TO 850
     GO TO 860
 850 IP(II)=1
     IP1=IP1-MQ3
     II=II-1
     IF(II.LE.0)GO TO 870
     GO TO 840
 860 IP(II)=0
     II=II-1
     IF(II.LE.0)GO TO 870
     GO TO 840
 870 RETURN
     END
```

Appendix 4. The Reynolds Logic Simulation Program

The Reynolds Logic Simulator is a FORTRAN program which allows the simulation of logical systems on a time-sharing computer. Each circuit element and input terminal of the system to be simulated is called a node and is assigned a number. Each node is described by a list comprising node number, function, and usually a list of the nodes which are its inputs. The end of each list is marked by a zero. This information, which totally describes the circuit, is then fed into the computer as a data file. The program is started and controlled by means of program commands. These also take numerical form but are distinguished from other numerical data by being negative. A full description of the logic simulator is given in reference [16] and a complete list of system commands and functions is presented in Figure A4.1

COMMANDS AND FUNCTIONS FOR 'LOGSIM'

COMMANDS:-

```
 -1:- DISCONNECT + READ IN NEW CONNECTIONS
 -2:- UPDATE CONNECTIONS
 -3:- PRINT OUT CONNECTIONS
 -4:- READ IN MONITOR POINTS
 -5:- READ INPUTS, FIRST SETTING ALL NODES TO ZERO
 -6:- READ INPUTS
 -7:- FREE-RUN, NORMAL TIMEBASE
 -8:- FREE-RUN, EXPANDED TIMEBASE
 -9:- FREE-RUN, NUMERICAL OUTPUT
-10:- FREE-RUN, NO MONITOR
-11:- READ IN STORE NUMBERS
-12:- PRINT OUT STORE NUMBERS
-13:- SINGLE-INPUT MODE
-14:- SUPPRESS HEADINGS
-15:- RESTORE HEADINGS
-16:- TITLE
-17:- READ FILE*1
-18:- READ CONSOLE
-19:- WRITE FILE*2
-20:- READ FILE*2
-21:- REWIND FILES
-22:- RESTART PROGRAM
-23:- QUIT PROGRAM
```

FUNCTIONS:-

Figure A4.1

```
 1: INPUT TERMINAL
 2: OR
 3: NOR
 4: AND
 5: NAND
 6: EQUIV
 7: NONEQUIV
 8: NOT
 9: D-TYPE FLIP-FLOP (CLOCK, J)
10: J-K (CLOCK, J, K, S, R)
11: TOGGLE (CLOCK, GATE)
12: STEERING CCT (CLOCK, J)
13: BISTABLE (S, R)
14: LOGICAL ZERO
15: LOGICAL ONE
16: CLOCK GEN.
17: MASTER-SLAVE (CLOCK, J, K, S, R, WKG.STORE)
18: DELAY CANCELLOR
19: NO-DELAY DUMMY
20: NO-DELAY INVERT
21: COUNTER (CLOCK, GATE, DEF.STORE, WKG.STORE)
22: REGISTER (CLOCK, J, DEF.STORE, WKG.STORE)
23: STORE
24: TRIP
25: CLOCKED COMPARATOR (CLOCK, INPUT STORE, INV.INPUT STORE)
26: GATED OR   (GATE, INPUTS)
27: GATED NOR  (GATE, INPUTS)
28: GATED AND  (GATE, INPUTS)
29: GATED NAND (GATE, INPUTS)
```

Appendix 5. The Hamming Code

All error-correcting codes require the introduction of one or more 'check-bits', and the Hamming code is one of the most convenient. As shown in Figure 7.5, if the bit positions are numbered in sequence from left to right, positions numbered as powers of two are reserved for parity check bits, while the remaining positions are information bits. If the three check bits are denoted P_1, P_2 and P_4, then they are determined as follows:-

P_1 is selected to establish even parity over bits 1,3,5,7.

P_2 is selected to establish even parity over bits 2,3,6,7.

P_4.is selected to establish even parity over bits 4,5,6,7.

In this way, various 7-bit code words are produced as shown in Figure A5.1.

If a fault occurs so that any bit in the code word is in error, then it can be detected and corrected simply by checking for odd parity over the same three combinations of bits for which even parity was initially established. For example, if the code word:-

0001111

becomes:-

0001011

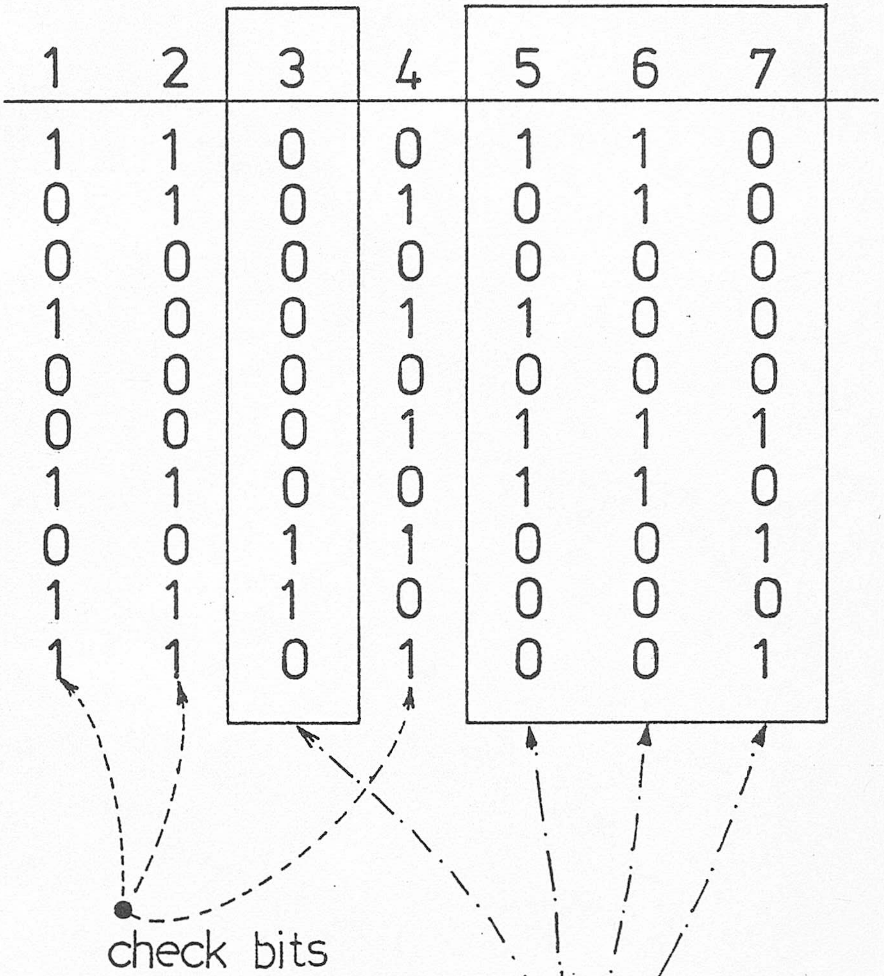
then the three parity check combinations become:-

$$P_4 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$P_2 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$$

$$P_1 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$$

This corresponds to decimal five, therefore the error is correctly assigned to position five in the code word.



1	2	3	4	5	6	7
1	1	0	0	1	1	0
0	1	0	1	0	1	0
0	0	0	0	0	0	0
1	0	0	1	1	0	0
0	0	0	0	0	0	0
0	0	0	1	1	1	1
1	1	0	0	1	1	0
0	0	1	1	0	0	1
1	1	1	0	0	0	0
1	1	0	1	0	0	1

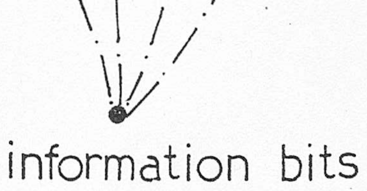


Figure A5.1