IMPLEMENTATION OF A TESTING AND INITIALIZATION ALGORITHM FOR A GENERALIZED TREE STRUCTURE

by

M. Hoptiak

Thesis submitted to the Graduate Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

APPROVED:

F. G. Gray, Chairman

R. A. Thompson

P. H. Wiley

May, 1977

Blacksburg, Virginia

LD
5655
V855
1977
H66
c.2

Acknowledgements

The author would like to express his appreciation to Dr. F. Gail Gray, Richard A. Thompson and Paris H. Wiley for serving on his advisory committee. Special thanks are extended to Professor F. Gail Gray, for his encouragement and guidance throughout the research and to Professor Richard A. Thompson for his help with some aspects of the research.

This research was funded by the National Science Foundation under grant DCR75-06543. This funding is hereby gratefully acknowledged.

The author wishes to extend a special note of thanks to his parents for their patience and encouragement and to a neighbor Mrs. Charlotte K. Thompson whose patience and encouragement can best be summeraized as follows: She was willing to listen even though she did not understand.

Table Of Contents

Acknowledgements ii

1. Introduction 1

2. Testing Algorithm 11

3. The Testing Structure 20

4. Testing The Modified Structure 46

4.1 Testing The Generalized Tree 47

4.2 Testing The CDSTLSR 50

4.3 Testing The CDSTITs 56

4.4 Testing The Combinational Logic 69

5. Initialization 97

6. Conclusions And Future Directions . . .110

Bibliography 113

Vita 114

1. Introduction

As computers continue to permeate the fabric of our every day life, the idea of reliability assumes even more importance in the design of digital computational systems. Although redundancy may be used as a means of improving the relaibility of a system, as the system size increases, this approach leads to greatly increased cost for very little additional reliability. In systems which have to operate reliably over extended time periods, the capability of fault diagnosis and self repair becomes very useful. Methods of designing such a system therefore take on additional importance.

If, in addition to the above properties, a functionally complete array of cells [1,2,3,4] had the capability of dynamic reconfiguration, i.e. the capability of changing the topology or function of the network while it was running, additional power would be gained. The capabilty of dynamic reconfiguration has been discussed in the literature by F. G. Gray and R. A. Thompson [5]. It could be used to allow the faulty network to continue computing without making use of the faulty component and therefore without loss of accuracy. This would of course allow higher utilization of the components in the network and would also allow

computation to continue well beyond that point in time at which it would have had to stop if the network were non-reconfigurable.

One structure which allows fault diagnosis with relative ease is that of the modular tree. The diagnosis of faults in this particular structure has been studied extensively. This iterative structure may be used for realizing any combinational or definite sequential machine. Further, two such trees with a single binary feedback variable can be used to realize any sequential machine. The problem addressed by this thesis is basically that of configuring these trees in such a manner as to allow the testing of the tree structure with relative ease. In addition, the question of how to reinitialize the tree for useful work after having tested it will also be examined.

The two types of trees to be considered in this thesis have very similar structure. They are as follows:

1) Universal iterative tree structure for realizing combinational functions: Figure 1.1 shows a universal iterative tree structure of depth μ. The inputs, $x_0,\ldots,x_{\mu-1}$ to the structure are called <u>primary input variables</u> and the horizontal inputs $C_0,\ldots,C_{2^{\mu}-1}$ are <u>control input variables</u>. The tree structure is capable of realizing any combinational function of μ or fewer variables. Both Cioffi and Fiorillo [6] and Kohavi and Berger [7] considered

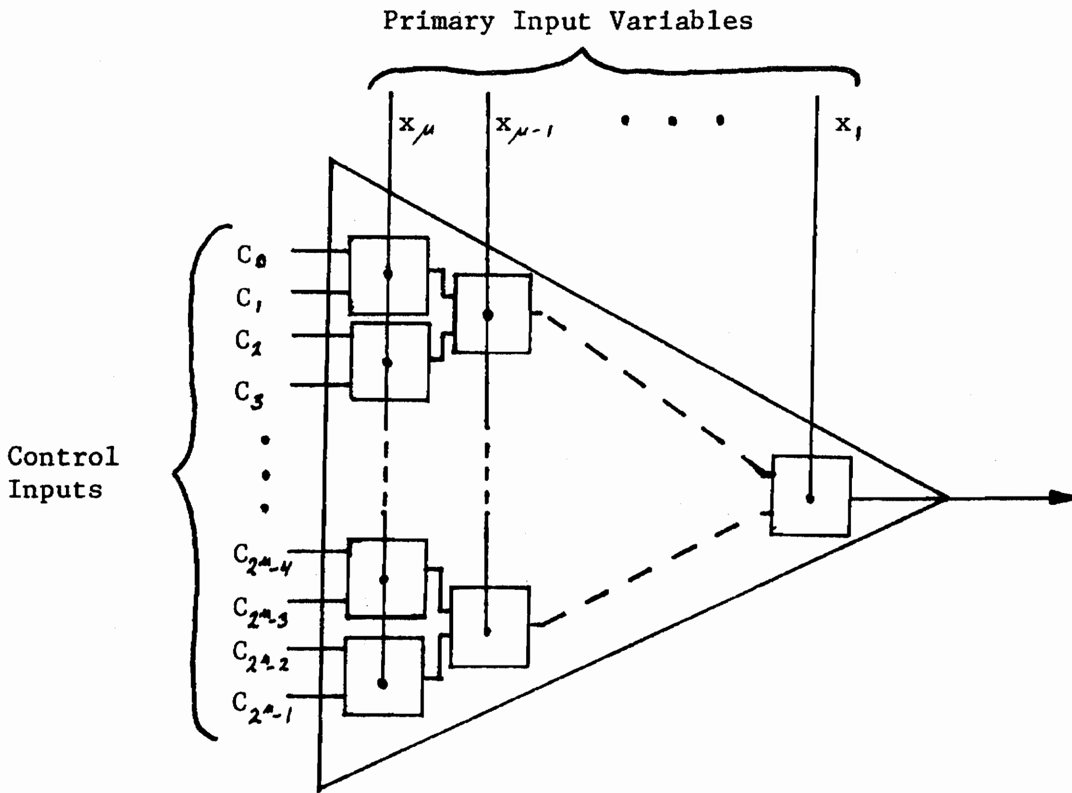


Figure 1.1: Universal Iterative Tree Structure for Realizing Combinational Functions

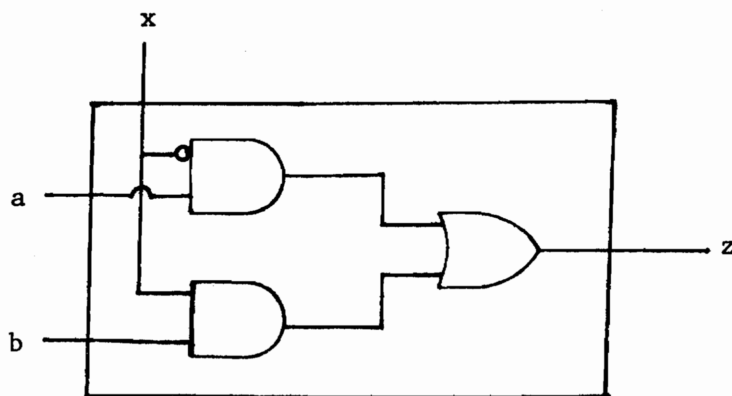


Figure 1.1(a): Details of each Module shown in Figure 1.1.

fault detection under unrestricted fault assumptions in combinational trees.

2) Universal iterative tree structure for realizing definite machines: Figure 1.2 shows a universal iterative tree structure which is a modified version of that shown in Figure 1.1. This structure has the capability of realizing any binary definite machine of order μ (Note that a binary definite machine of order μ is a sequential machine having a binary input alphabet such that its present state can be uniquely determined from a knowledge of its past μ inputs).

After studying the operation of the tree structure it has been noted that no more power is obtained with the structure as shown in Figures 1.1 and 1.2 than can be obtained if all the primary inputs were connected. Figure 1.3 shows the general block diagramatic representation for such a tree structure. Note that now, the structure is a tree with a single input. It would however be preferable to have a structure which would have multiple inputs, in other words, would be non-binary in nature. The gate equivalents for such a modification are given by Arnold [4] and are displayed in Figure 1.4, while its block diagramatic representation is given by Figure 1.5. Note that at present, there is no means of distinguishing the block diagrams of the sequential tree structure and that of the combinational tree. Later a convention will be established

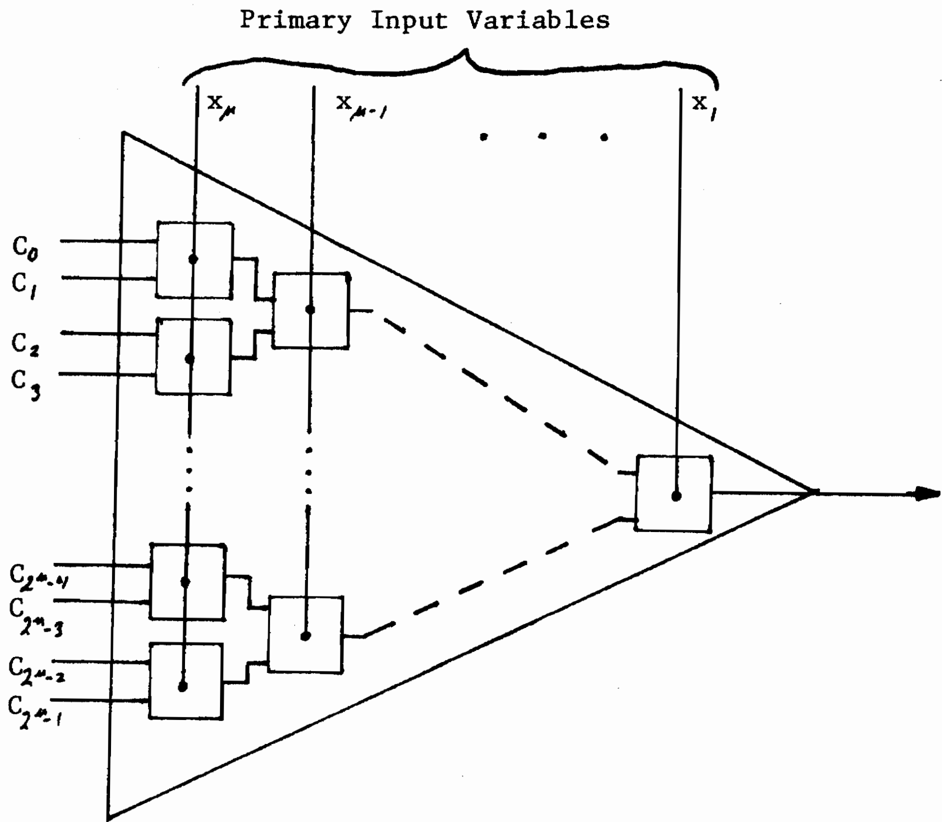


Figure 1.2: Universal Iterative Tree Structure for Realizing Binary Definite Machines.

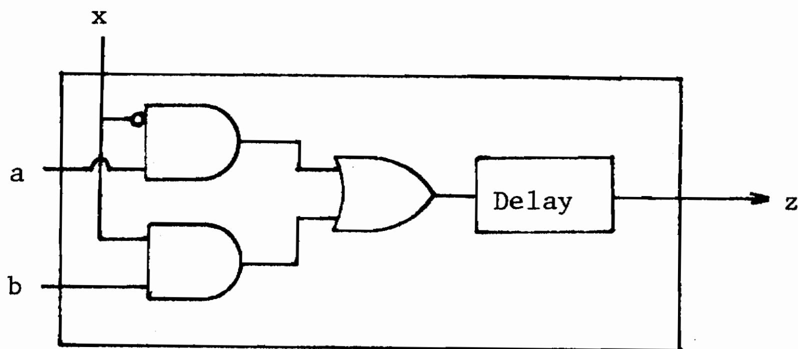


Figure 1.2(a): Details of each Module of Figure 1.2.

Primary Input Variable

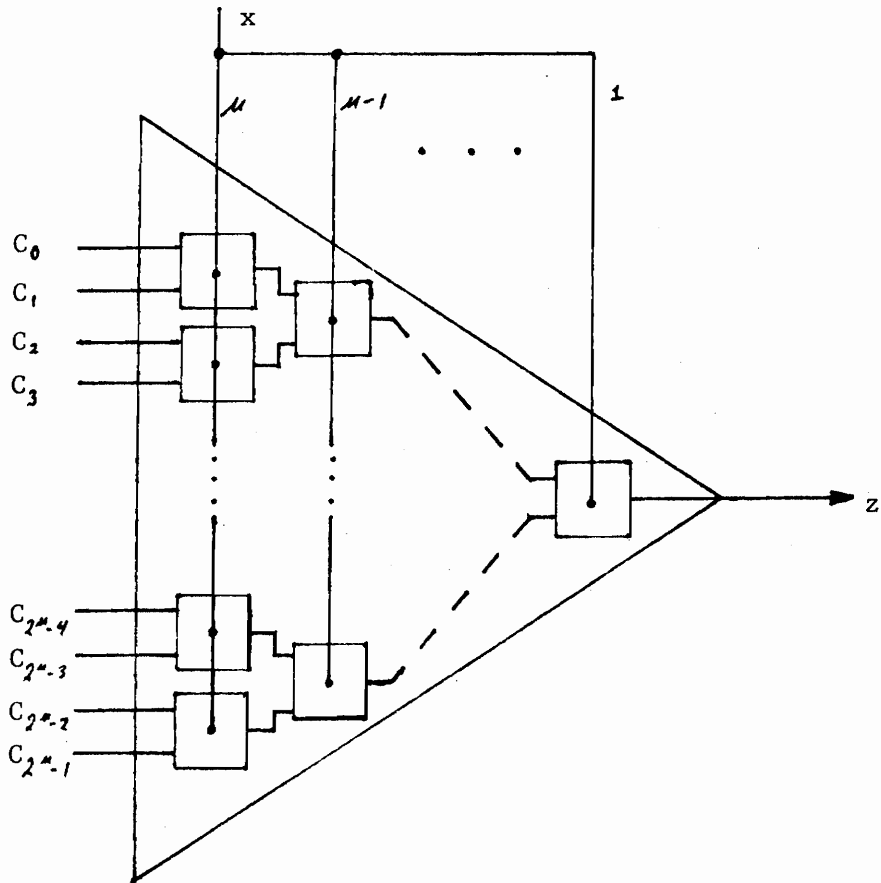


Figure 1.3: Universal Iterative Tree Structure with one input and of Depth μ . (Note that each Module may be either Combinational or Sequential in Nature)

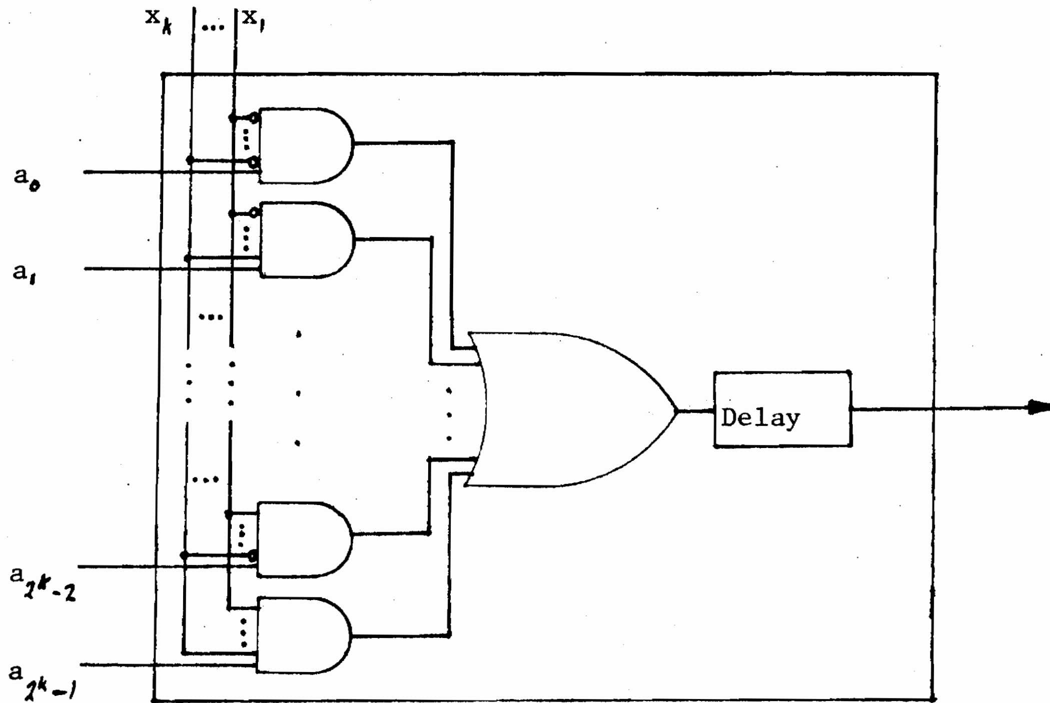


Figure 1.4: Details of each Module used in the Universal Iterative Tree Structure for realizing non-binary Definite Machines.

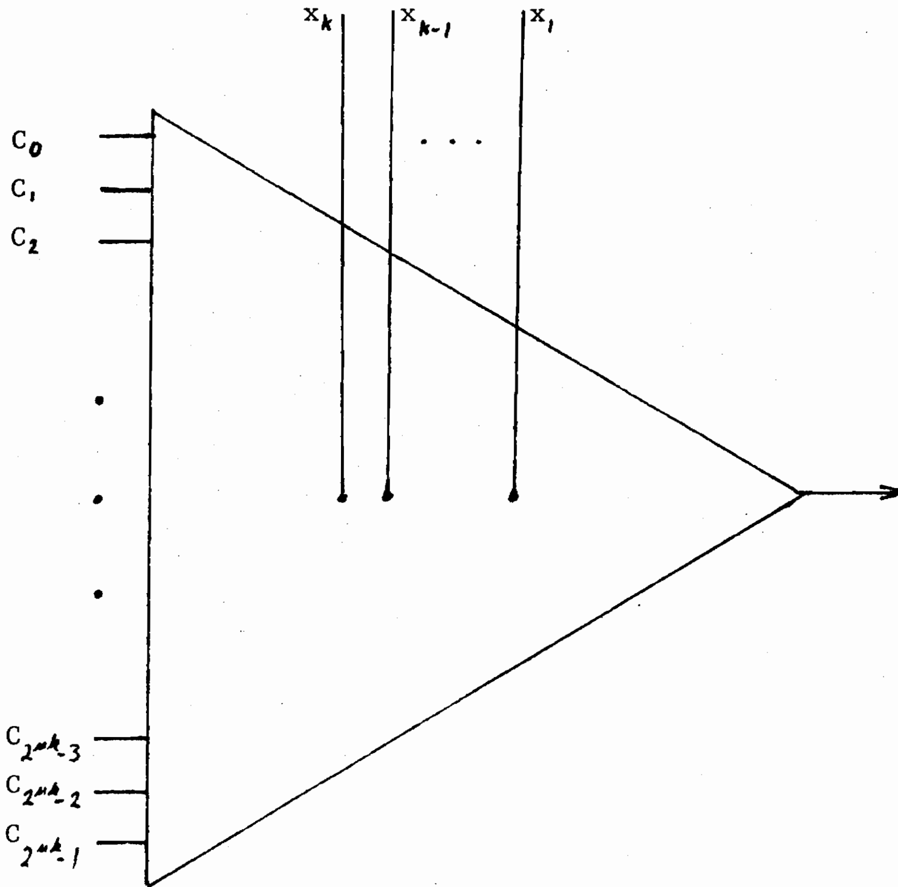


Figure 1.5: Diagramatic Representation of a Multiple Input Definite Tree.

to differentiate between the two. The addition of inputs is necessary if feedback is to be used in the design of the machine.

Before moving too deeply into the area of fault detection, it would be well worth while to define the type of faults that are being tested for. The set of <u>single stuck-at faults</u>, forming perhaps the most important fault class, is the class in which one and only one given line in the network acts as if it were a logical constant and so becomes insensitive to any changes in the network inputs. It is possible to have a <u>single stuck-at fault</u> (only one line in the entire tree, or network, can be faulty) or <u>multiple stuck-at faults</u> (in which case more than one line may be faulty simultaneously). The use of a stuck-at fault model is only valid for those networks which can be discribed down to the gate level. As has already been shown, the generalized universal modular tree structure can be represented at this level. The problem of stuck-at faults in the sequential tree structure has been examined by Ramos and Smith [8] and by Prasad and Gray [9]. The detection algorithm developed by Prasad and Gray [9] and given in Chapter 2 is one which will detect all single and multiple stuck-at faults in the tree structure. While the stuck-at fault model does not include all possible failures, it does include the common ones. Additionally, a detection

algorithm has not been developed which will find all faults. Prasad and Gray have created a testing algorithm which will catch most faults under the unrestricted fault assumption (the network may fail in any functional manner and not just to stuck-at conditions). However, as yet no classification has been made as to what types of faults will be missed or how likely these faults are to occur. For this reason, only the stuck-at faults will be tested for by the algorithm which is given in the following chapter. For all of the above reasons, the testing algorithm which will be implemented will be one which is capable of detecting all single or multiple stuck-at faults in the generalized modular tree structure.

2. The Testing Algorithm

The following algorithm was originally developed by B. A. Prasad and F. G. Gray [9]. Its function is that of finding all stuck-at faults in the generalized modular tree structure. The same algorithm can be used in the testing of both the sequential and combinational tree structures. The proof that the algorithm actually does test the tree structure may be found in the above reference. Only the algorithm and testing experiment will be stated in this chapter. The Algorithms given below as 2.1 and 2.2 correspond to 5.2 and 5.3 in Prasad and Gray's work, while Experiments 2.1 and 2.2 correspond to 5.1 and 5.2.

Algorithm 2.1 generates the test patterns C and $\overline{C}$ which must be placed on the control inputs (leaves) of the Binary Definite Tree.

__Algorithm 2.1__:

(i) Let μ=the tree depth

(ii) Set i=1

(iii) Set the pattern to 01

(iv) If $i<\mu$, go to (vi)

(v) print the pattern calling it C. Print the complemented pattern calling it $\overline{C}$. Stop.

(vi) Complement the pattern. Concatenate the

complemented pattern to the original pattern.

(vii) Let i=i+1. Go to (iv).

The above algorithm defines the values of the control patterns (C and $\bar{C}$) to be placed on the leaves of the tree. The actual testing experiment uses these strings as follows:

<u>Experiment 2.1</u>:

(i) Obtain the tree depth μ.

(ii) Obtain the control patterns C and $\bar{C}$ using Algorithm 2.1.

(iii) Obtain a sequence of length $2^{\mu}+\mu-1$ containing all sequences of length μ as subsequences [10].

(iv) Set the control pattern on the leaves to C.

(v) If the tree under test is a definite tree, as in Figure 1.1, apply the sequence obtained in (iii) at the primary input terminal and note the output sequence ignoring the first $(\mu-1)$ outputs. If the tree under test is a combinational tree as in Figure 1.2, apply all the 2^{μ} binary μ-tuples at the primary inputs and note the outputs.

(vi) Set the control pattern to $\bar{C}$. Do step (v) once again. Stop.

As already stated, although we have a testing algorithm for the binary tree, it is necessary that there exist an algorithm which is capable of testing non-binary trees. Such an algorithm and testing experiment are given below as Algorithm 2.2 and Experiment 2.2.

<u>Algorithm 2.2</u>:

(i) Obtain μ, the order of definiteness, and k, the number of input lines.

(ii) Set $K = 2^k$ = number of different input combinations

(iii) Obtain two binary patterns C and $\bar{C}$ of length K using Algorithm 2.1.

(iv) Set i = 1, c = C, and $\bar{c} = \bar{C}$.

(v) If $i < \mu$, go to (vii).

(vi) Print c and $\bar{c}$. Stop.

(vii) Replace each 1 in c by C, and each 0 in c by $\bar{C}$.

(viii) Set i = i + 1, and go to (v).

The method used in detecting faults in multiple input trees is shown in the following experiment.

<u>Experiment 2.2</u>:

(i) Obtain c and $\bar{c}$ by Algorithm 2.2.

(ii) Denote the 2^k inputs as 0,1,2,...,K-1.

(iii) Obtain a K-ary input sequence of length $K^{\mu}+\mu-1$ which contains as subsequences all K-ary input sequences of length μ.

(iv) Set the control pattern to c, and apply the input sequence obtained in (iii). Note the outputs ignoring the first $\mu-1$.

(v) Set the control pattern to c, and apply the input sequence obtained in (iii). Note the outputs ignoring the first $\mu-1$. Stop.

With the above two algorithms and experiments there now exists a basis for constructing a usable testing algorithm which can be mounted on a set of trees. So far we have assumed that the control patterns could be placed on the leaves and that there would be no difficulty in changing that pattern. Immediately it is seen that the control inputs (the term leaves will be used interchangably with that of control inputs hereafter) can not be tied to constants. Yet it is also noted that they must have the constants which determine the tree's function applied to them whenever the tree is performing calculations. If a parallel output shift register were to be placed at the leaves of the tree, then the control pattern could be varied at any time merely by shifting in the new pattern. All that must be determined is if this addition still allows the

trees to be tested.

It is assumed that the only type of failure which can occur in the shift register is a stuck-at fault or a set of stuck-at faults at the outputs from each stage or the inputs to each stage. To begin, assume that there is only a single stuck-at fault. It has already been noted that the shift register must have $2^{\mu k}$ stages, as a k input tree of depth μ has $2^{\mu k}$ leaves and only one stage should be necessary per leaf. Therefore let stage i, $0 \leq i \leq 2^{\mu k}-1$, be stuck-at p, where p is defined to be a logical constant. This creates two ($i<2^{\mu k}-1$ and $i=2^{\mu k}-1$) cases which must be taken care of; Since i is stuck-at p, and the data is being transferred serially, the leaves from i+1 to $2^{\mu k}-1$ must all output the value p. Because of the manner in which the testing algorithm is constructed, when either c or $\bar{c}$ is shifted into the register, then stage $2^{\mu k}-2$ must be a 1 and the stage $2^{\mu k}-1$ must be a 0 or vice versa. Unless $i=2^{\mu k}-1$ and p=0 the fault will be found. If $i=2^{\mu k}-1$ and p=0 then when the complement of the pattern above is shifted into the register the output of stage i will be 0 where as it should have been 1. Therefore all single stuck-at faults in the shift register will be detected. Assume now that there are multiple stuck-at faults. Consider that fault closest to the last stage of the register. Let the stage at which the fault occurs be stage i. By using the same argument as

above, either the pattern c or $\bar{c}$ must catch the fault and since both will be applied in the normal course of testing, a fault will be detected. Figures 2.1 and 2.2 show the conventions which will be used to denote the tree with a shift register attached to the leaves. Figure 2.1 shows the convention used to represent combinational trees with shift registers attached to the leaves whereas Figure 2.2 shows that which will be used to represent sequential trees with shift registers attached to the leaves. Note the difference in the markings on the shift registers. This convention will be used as a means of differentiating between the two types of trees when both are used in the same diagram.

In summary it can therefore be seen that testing can still be accomplished with the same algorithm as before when a shift register is added to the leaves of the tree. It is indeed possible to change the pattern on the control inputs (leaves) which was assumed to be possible when the testing algorithm was developed. In this chapter a convention was established which differentiates between the combinational and sequential trees when both are to be used in the same drawing. This convention will later provide clarity in the diagrams containing both types of trees. This distinction is necessary as the outputs of the combinational tree are not delayed at all, while those of the sequential tree are delayed for one clock pulse (the sequential trees are Moore

machines). The next chapter will build upon the structure given in this chapter and so allow greater flexability in computation, reconfiguration, and testing of the tree structure.

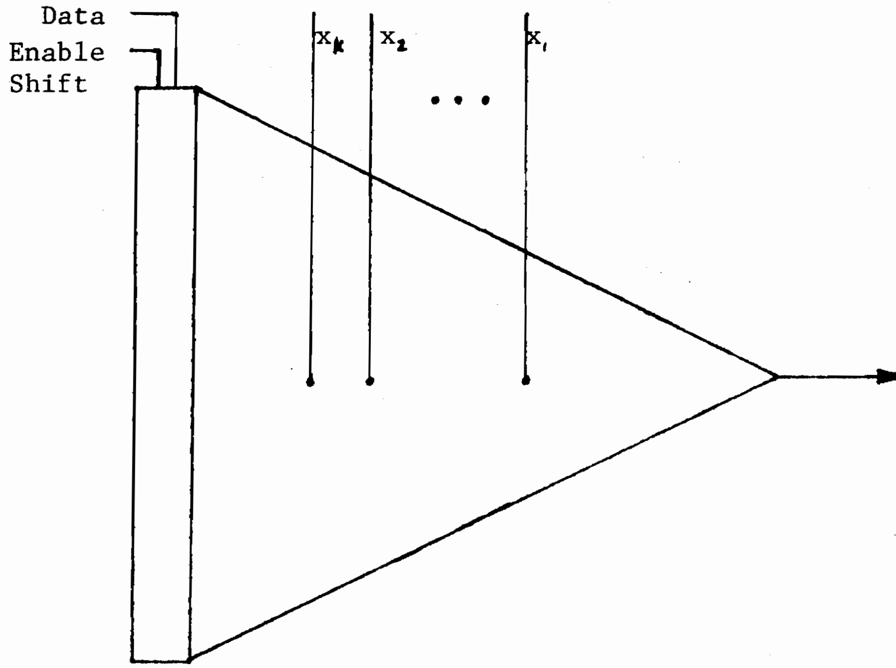


Figure 2.1; Diagramatic Convention adopted to Display a k-input Universal Modular Combinational Tree.

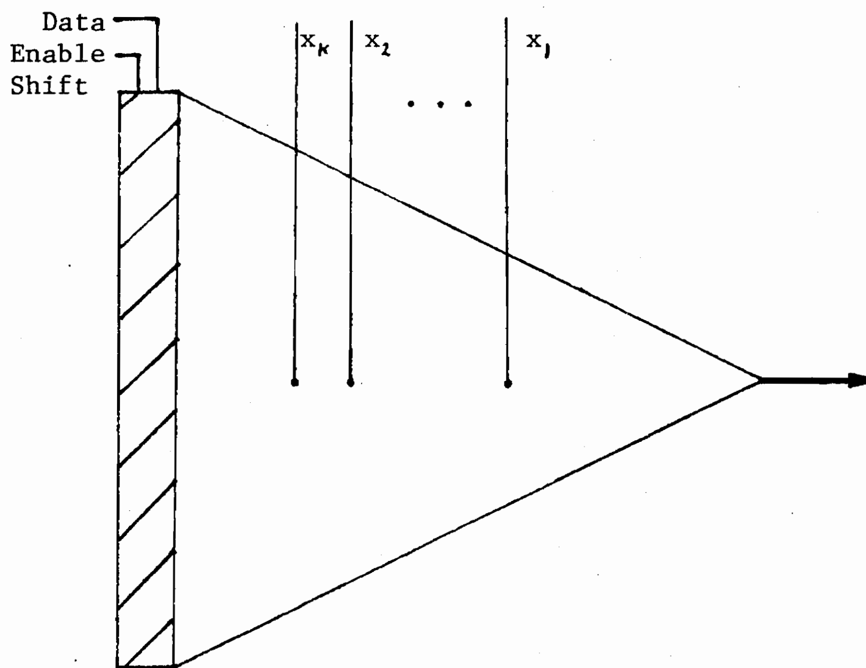


Figure 2.2: Diagramatic Convention adopted to Display a k-input Universal Modular Definite Sequential Tree.

3. The Testing Structure

In both Experiments 2.1 and 2.2 it was assumed that a sequence could be developed and applied to the inputs of the tree under test and also that the appropriate patterns could be applied to the leaves of the tree. At present, the latter has been partially solved and so it will not be necessary to provide any more structure for the storage of the control patterns. Therefore assume that the data lead of the shift register is the output of some tree which generates the patterns c and $\bar{c}$ and further, that the shift register will only be enabled at the appropriate times. This leaves only the question of how to apply the testing sequence to the inputs of the tree under test as a problem to be solved.

Friedman [1] showed that any sequential machine may be realized by a sequential network with only one feedback wire. Arnold [4] then took this result and applied it to the universal tree structure. The result is that any definite machine of order μ may be implemented as a set of trees, each producing one of the outputs of the machine and one additional tree producing the feedback signal. The structure shown in Figure 3.1 is an n output machine of depth μ (each of the trees is of depth μ). This result

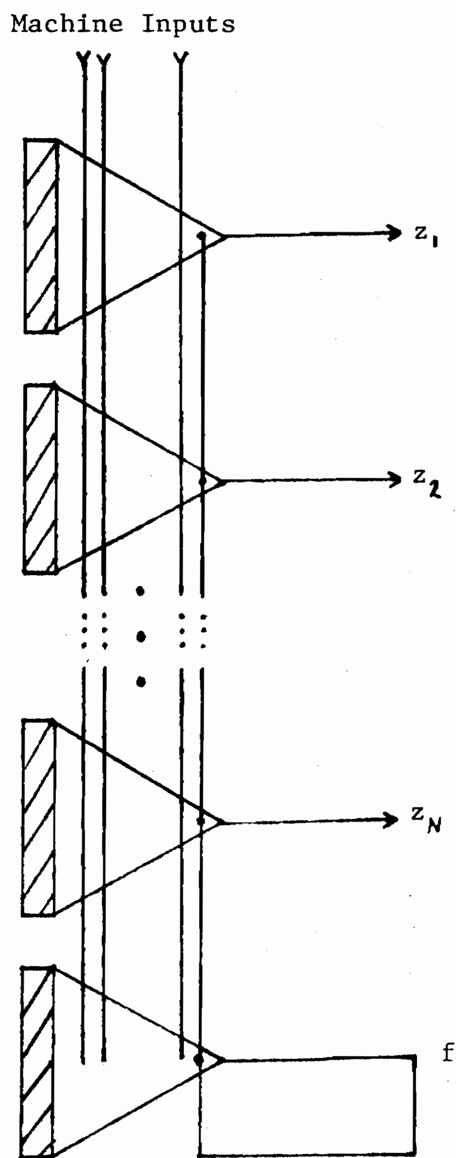


Figure 3.1: General N output Sequential Machine of Depth μ.

should however not be allowed to influence, too strongly, the design of the testing algorithm. Additionally, it would be desirable for the sequence generater to be capable of driving itself, i.e. to be entirely self contained and be able to generate the K-ary sequence without the insertion of any external signals. If the above constraint is to be met, the sequence generater should have a structure like that shown in Figure 3.2.

The network of Figure 3.2 has no external inputs. It must now be shown that this structure will actually be capable of generating the testing sequence of step (iii) of Experiment 2.1 (generating the K-ary sequence containing all subsequences of length μ). To answer this question the structure of the mathematical sequence to be generated will be examined. First, the sequence can be no shorter than $2^{\mu k}+\mu-1$, where k is the number of inputs, if it is to contain all K-ary ($K=2^k$) sequences of length μ. If this is the case, then no K-ary sequence of length μ can be included twice. The last is obvious, if some K-ary sequence of length μ were included twice then the sequence containing this replicated sequence could not be the shortest, as the sequence without any replications would be shorter. Therefore, each K-ary sequence must occur once and only once. Additionally, as we have already noted, the testing sequence need be of length no greater than $2^{\mu k}+\mu-1$. Looking

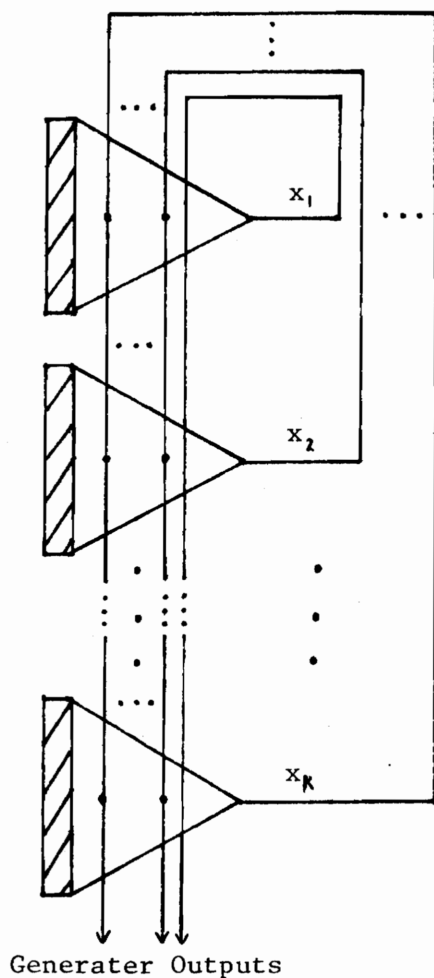


Figure 3.2: Testing Sequence Generater

at the definite tree of depth μ which has k inputs, it can be easily shown that there are $2^{\mu k}$ leaves. Note also, that since the tree is of depth μ, therefore $\mu-1$ outputs will occur before the first value from a leaf is output. It should at once be seen that the k input definite sequential tree of depth μ can therefore theoretically produce a sequence of length $2^{\mu k}+\mu-1$. If k of these trees were connected into a network having the same topology as that of Figure 3.2, it might be possible to produce the sequence. Earlier it was noted that no sequence of length μ was repeated. Each leaf would therfore be accessed once and only once. This implies that there is no possibility that a leaf would be required to have two different values. Additionally, since the order of the K-ary digits in the sequence is known and the means in which data travels through the tree structure is known, it is possible to place the data on the leaves (and to initialize the tree) in such a way as to have the present K-ary digit "calling" that digit which is $\mu-1$ positions further in the testing string. Because such a capability is present, the machine which generates the testing sequence is much "simpler" than would ordinarily be expected. In addition, this machine requires no external inputs for proper functioning and therefore does not have to utilize any of the power of the system controller or of any other trees in generating the testing

sequence. This structure also uses only k trees to generate the testing sequence as opposed to the k+1 trees used in a "normal" machine which produces the same outputs. Algorithm 3.1 can be used to generate the leaf values and initialization pattern which will produce the sequence of all K-ary subsequences of length μ while Example 3.1 shows how Algorithm 3.1 works.

<u>Algorithm 3.1</u>:

(i) Compute the sequence containing all K-ary subsequences of length μ and call it T.

(ii) Let i=1.

(iii) Construct a major column with the major column heading of z and a minor column heading of blank.

(iv) If $i>\mu$ then go to (xi).

(v) Construct a set of $2^{k(i-1)}$ major columns such that each major column has as a heading a minor column heading of the leftmost major column which has not been previously expanded.

(vi) Expand all major columns of the previous set.

(vii) Construct K minor columns in each major column just generated.

(viii) Label each minor column with the major column heading concatenated with the next K-ary number

varying from 0 to K-1.

(ix) Let i=i+1.

(x) Go to (iv).

(xi) Place sequence T in major column z.

(xii) Let the first row be row 0.

(xiii) If all columns are non-blank then go to (xxvi).

(xiv) Begin working on the leftmost major non-blank column.

(xv) Set H equal to the major column heading of this column.

(xvi) Let $i=\mu$.

(xvii) If $i>2^{\mu k}+\mu-2$ then go to (xxiv).

(xviii) Set F equal to the entry in row i of the column with the major column heading which matches H unless $i=2^{\mu k}+\mu-2$, then set F equal to the entry in row $\mu-2$ of the column with the minor column heading which matches H.

(xix) Obtain the entry in row i-1 of the column with the major column heading of z and set G equal to it.

(xx) Place entry F in the position given by the intersection of row i-1 and the column with the minor column heading of H concatenated with G.

(xxi) Place dashes (-) at the intersection of all other minor columns of the major column with the heading which matches H and row i-1.

(xxii) Let i=i+1.

(xxiii) Go to (xvii).

(xxiv) Set the entries of the first $\mu-1$ rows of the major column H equal to those of the last $\mu-1$ rows of said column.

(xxv) Go to (xiii).

(xxvi) Print the table Stop. (the table is now complete) The value to be placed on each leaf of the tree is the non-dashed entry in the minor column with the same designation as the leaf. The initialization states for the tree are contained in the table as the entries in row 0 and the column which has the major column heading of the output of the internal Flip-Flop.

The following example partially shows the construction of the Table 3.1 using Algorithm 3.1. This table is constructed for a K-ary tree with $\mu=2$ and k=2. Figure 3.3 shows the line numbering scheme used in Algorithm 3.1 as it applies to the above generalized tree.

Example 3.1:

Let a table be built which will allow a 2 input tree of depth 2 to generate the 4-ary sequence containing all

Table 3.1: Table of leaf and initialization values for a tree with μ=2 and k=2 generated from Algorithm 3.1.

row	z					0				1				2				3			
		0	1	2	3	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
0	0	0	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	0	1	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-
2	1	-	1	-	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-	-	-
3	1	-	0	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-
4	0	2	-	-	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-
5	2	-	-	2	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-
6	2	-	-	1	-	-	-	-	-	-	-	2	-	-	-	-	-	-	-	-	-
7	1	-	2	-	-	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	-
8	2	-	-	0	-	-	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-
9	0	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-
10	3	-	-	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	2
11	3	-	-	-	2	-	-	-	-	-	-	-	-	-	-	-	3	-	-	-	-
12	2	-	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-
13	3	-	-	-	1	-	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
14	1	-	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	-	-
15	3	-	-	-	0	-	-	-	0	-	-	-	-	-	-	-	-	-	-	-	-
16	0	0	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

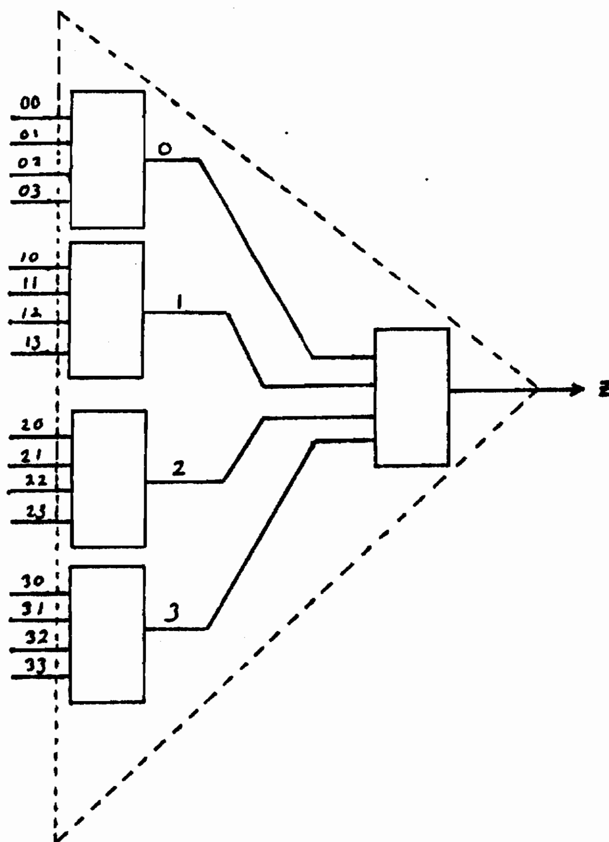


Figure 3.3: Example of Line Numbering Scheme used in Algorithm 3.1.

subsequences of length 2. Using Algorithm 3.1 the following steps are generated.

T=00110221203323130

Construct column z of Table 3.1

i=1 Construct $2^{2(0)}$ major columns each with a major heading of "blank"

Construct K minor columns in each major column (4).

Label each minor column with the major column heading concatenated with the next 4-ary number varying from 0 to k-1; i.e. the minor columns would be labeled 0, 1, 2, 3.

i=2 Construct $2^{2(1)}$ major columns with the headings 0, 1, 2, 3.

Construct 4 minor columns in each major column.

Label each minor column with the major column heading concatenated with the next 4-ary number varying from 0 to K-1 (3); i.e. the minor columns of major column 0 would be labeled 00, 01, 02, 03, while the minor columns of major column 1 would be labeled 10, 11, 12, 13, etc.

i=3 $i>\mu$ therefore go to (xi).

Place sequence T in major column z.

H="blank"

i=2 Set F=1 (the entry in row i).

G=0

H concatenated with G is 0 (ignore the leading blank).

Place the value of F in column 0 and dashes in columns 1, 2, and 3.

i=3

F=1

G=1

Place the value of F in column 1 (i.e. place a 1 in column 1) and dashes in columns 0, 2, and 3.

i=4

Continue these operations.

i=15

F=0

G=3

Place the value of F in column 3 (place a 0 in column 3) and dashes in columns 0, 1, and 2.

i=16 Therefore $i=2^{k\mu}+\mu-2$ so look up F in row $\mu-2$ (0).

F=0

G=0

Place a 0 in column 0 and dashes in columns 1, 2, and 3.

Set the entries of the first $\mu-1$ (1) row(s), of the major column which has the heading equal to the value of H, equal to the the last $\mu-1$ (1) row(s); i.e. Set row 0 equal to row 16 (0---).

Begin working on the leftmost major non-blank column;

i.e. column 0.

H=0

i=2

Set F equal to the entry in row 2 of minor column 0.

F=-.

Obtain G from row i-1 (1) of column z.

G=0.

Place the contents of F (-) into row i-1 and column H concatenated with G (i.e. 00) and dashes in all other columns (this of course places dashes (-) in all 4 columns).

continue...

i=4

Set F equal to the entry in row 4 of column 0.

F=2.

Obtain G from row i-1 (3) of column z.

G=1.

Place F (2) in row i-1 (3) of column H concatenated with G (01) and dashes in all other columns (this places the sequence -2-- in row 3 columns 00, 01, 02, 03 respectively).

continue...

Complete the rest of the Table.

Table 3.2 shows a partial table of a 1 input tree of depth 3. The algorithm was used to develop this table also.

In general however, the structure of Figure 3.2 is not satisfactory for general calculations. If the machine to be designed had the structure of Figure 3.1 with n greater than k, it would be necessary to break the feedback path before testing the tree which generates the feedback function. Also, if the machine is to be reconfigurable, the inputs to the various trees should be able to be modified. It has already been determined that the structure of a generalized tree is relatively easy to test. Perhaps there is a way of using the tree structure to aid in this data steering. As the combinational tree acts as a data steering network it might be possible to use it in the design. Figure 3.4 displays a possible configuration for the generalized modular tree. An example of the operation of a Combinational Data Steering Tree will be given later.

The shift registers attached to the leaves of the combinational trees are parallel in / parallel out types and a possible internal configuration for a single stage is shown in Figure 3.5. Note that the D Flip-Flop should be of a NOR type Master-Slave construction, with respect to the Set and Clear inputs. This constraint is added so that if a failure causes both the Set and Clear inputs of a given Flip-Flop to be 1 simultaneously, the output of the

Table 3.2: Table generated from Algorithm 3.1 for a 1 input tree Of Depth 3.

row	z			0		1		00		01		
		0	1	00	01	10	11	000	001	010	011	100...
0	0	0	-	0	-			1	-	-	-	
1	0	0	-	1	-			-	-	-	-	
2	0	1	-	-	-			-	-	-	-	
3	1	-	1	-	-			-	-	-	-	
4	1	-	1	-	-	...		-	-	-	1	...
5	1	-	0	-	1			-	-	-	-	
6	0	1	-	-	-			-	-	0	-	
7	1	-	0	-	0			-	0	-	-	
8	0	0	-	0	-			1	-	-	-	
9	0	0	-	1	-			-	-	-	-	

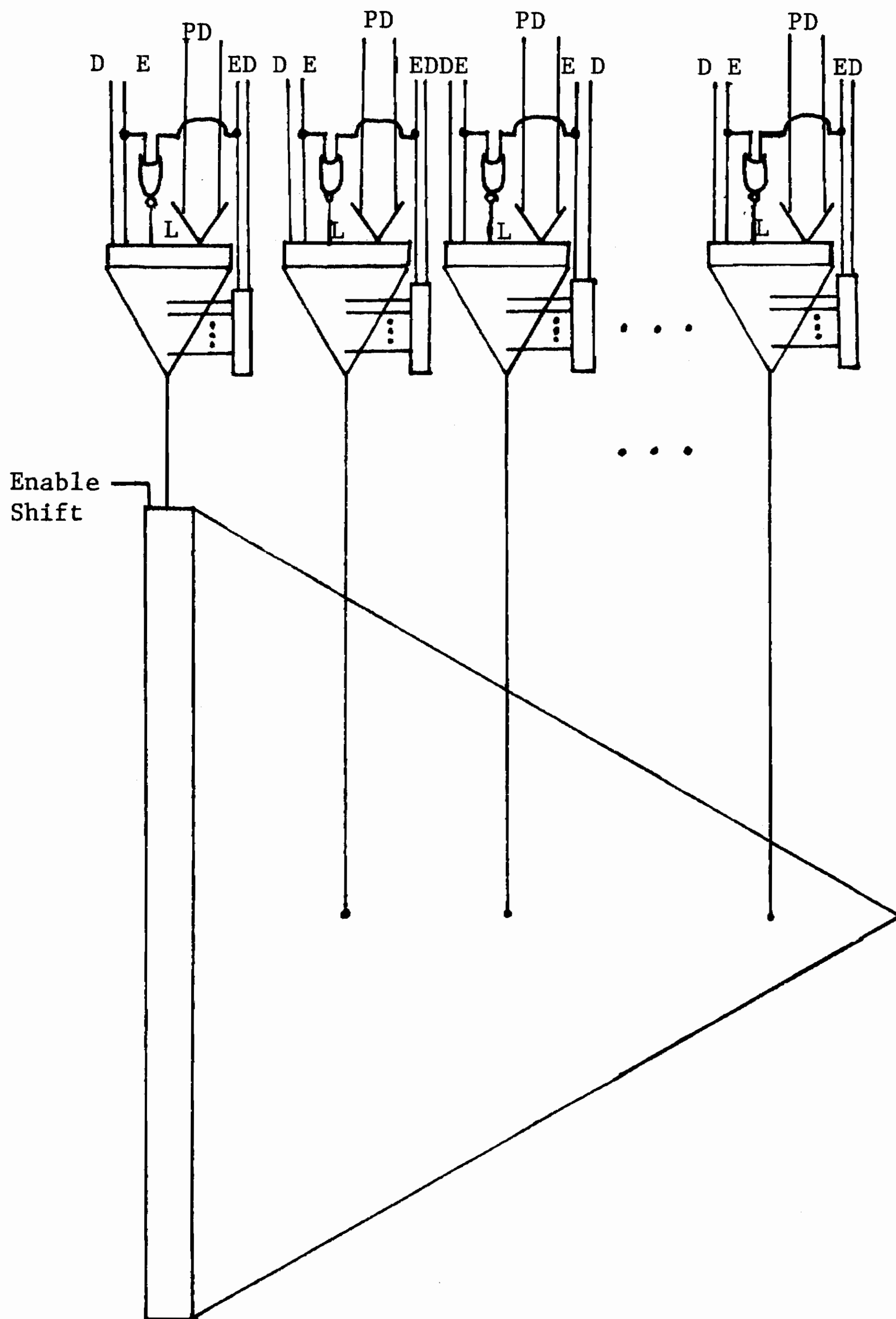


Figure 3.4: Possible Configuration for the New Generalized Modular Tree Structure.

D= Data Input E= Enable Shift P= Parallel L= Load Broadside

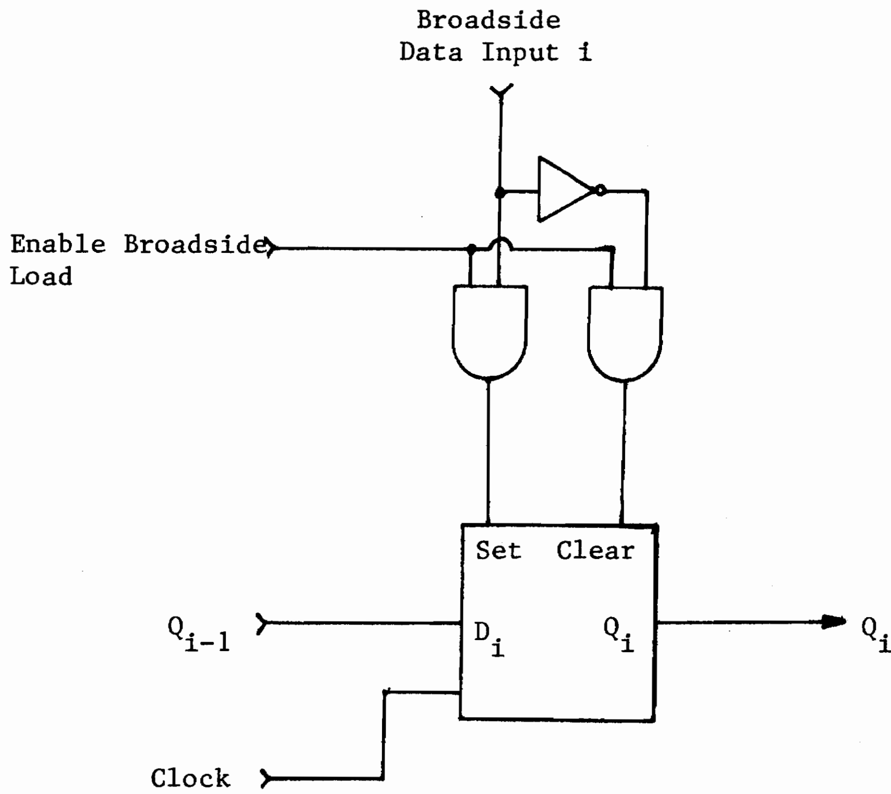


Figure 3.5: Internal Configuration of the ith stage of the Parallel In/Parallel Out Shift Register attached To the leaves of a Combinational Data Steering Tree.

Flip-Flop will be 0 and will not be oscillating between 0 and 1. A gate equivalent of the NOR type D Flip-Flop is given by Figure 3.6.

With the structure of Figure 3.4, it is now possible to set up the testing topology given in Figure 3.2 as well as that of the generalized sequential machine given in Figure 3.1. In addition, such a structure forces the number of leaf inputs to grow exponentially while the input shift registers of the combinational trees grow linearly. Some of these inputs may therefore be used for constants and so will allow a tree with k inputs to emulate a tree with less than k inputs without using some tree in the network to provide the constant function (either all 0's or all 1's independent of the inputs).

Unfortunately, some combinational logic is necessary in this structure. This additional logic is however, rather limited. Chapter 4 will modify the testing algorithm so as to test the additional structure that has been added to the tree. There are two means of handling the shift inputs to the various registers. These leads may be connected to a controlling structure or they may be controlled by a set of trees within the machine. If the latter is to be done, then more gates must be added to the structure of Figure 3.4. These gates essentially decode the outputs of a set of "special" trees which would control the shift registers of

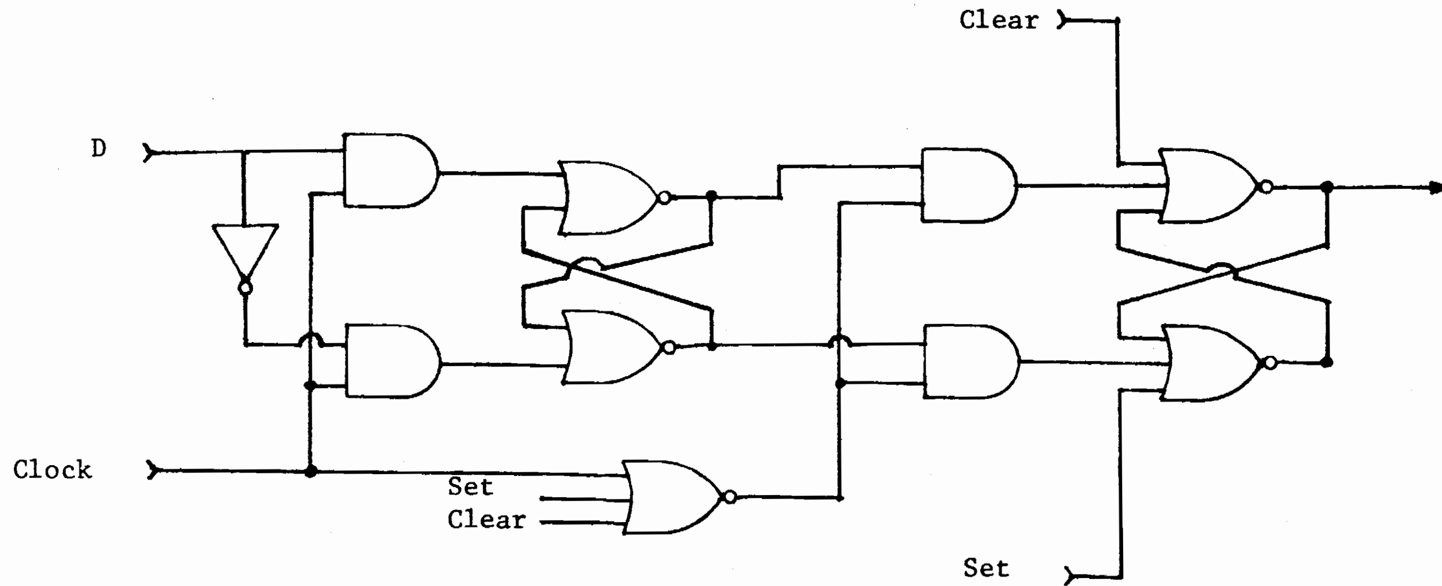


Figure 3.6: NOR D-Type Master Slave Flip-Flop with Set and Clear Inputs.

all the trees in the machine. Figures 3.7 and 3.8 show the additional logic which would be necessary to implement such a system, while Figure 3.9 shows the "control trees" added on as an additional section of the machine. Table 3.3 displays the codes used on the AND gates in the designs given in Figures 3.7 and 3.8. It should be noted that the codes 000 and 111 are not used. These codes should not be used in any encoding, as this would not allow the control trees to be easily tested.

To examine the operation of the structure of Figure 3.7, a very simple example will be used. Let the ISRLSR be of length 3. This implies that given a μ of 1, the length of the LSRLSR will be 2^3. Assume that $\mu=1$ in the Combinational Data Steering Tree. Let the contents of the ISRLSR be the address of a given leaf, say leaf 0. This means that at the beginning of this example, any broadside input (the output of another tree for example) which is attached to leaf 0 will be routed to the output of the Data Steering Tree and from there, directly into the LSR data input. Assume that the input to the LSR should now come from the lead attached to some other leaf, say leaf 5. The "control trees" would output the pattern 110 for 3 time periods while the code 101 (5) was shifted into the ISRLSR. After the above action occured the output of the Data Steering Tree would reflect the broadside input of leaf 5 of

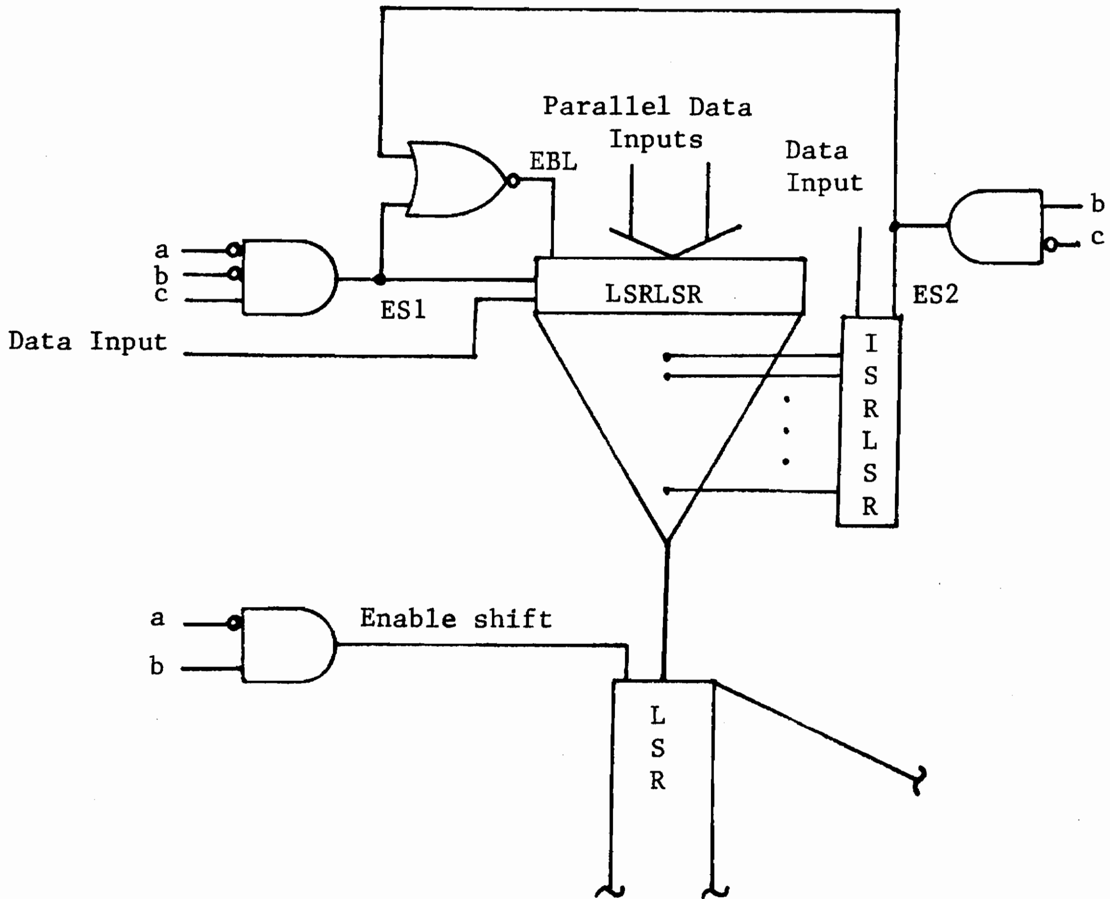


Figure 3.7: Combinational Data Steering Tree attached to the Leaf Shift Register of a Generalized Modular Tree.

Signal	Definition
a	Produced by "control tree" 1
b	Produced by "control tree" 2
c	Produced by "control tree" 3
ES1	Enable Shift to Leaf Shift Register of Combinational Tree
EBL	Enable Broadside Load
ES2	Enable Shift to Input Register of Combinational Tree.

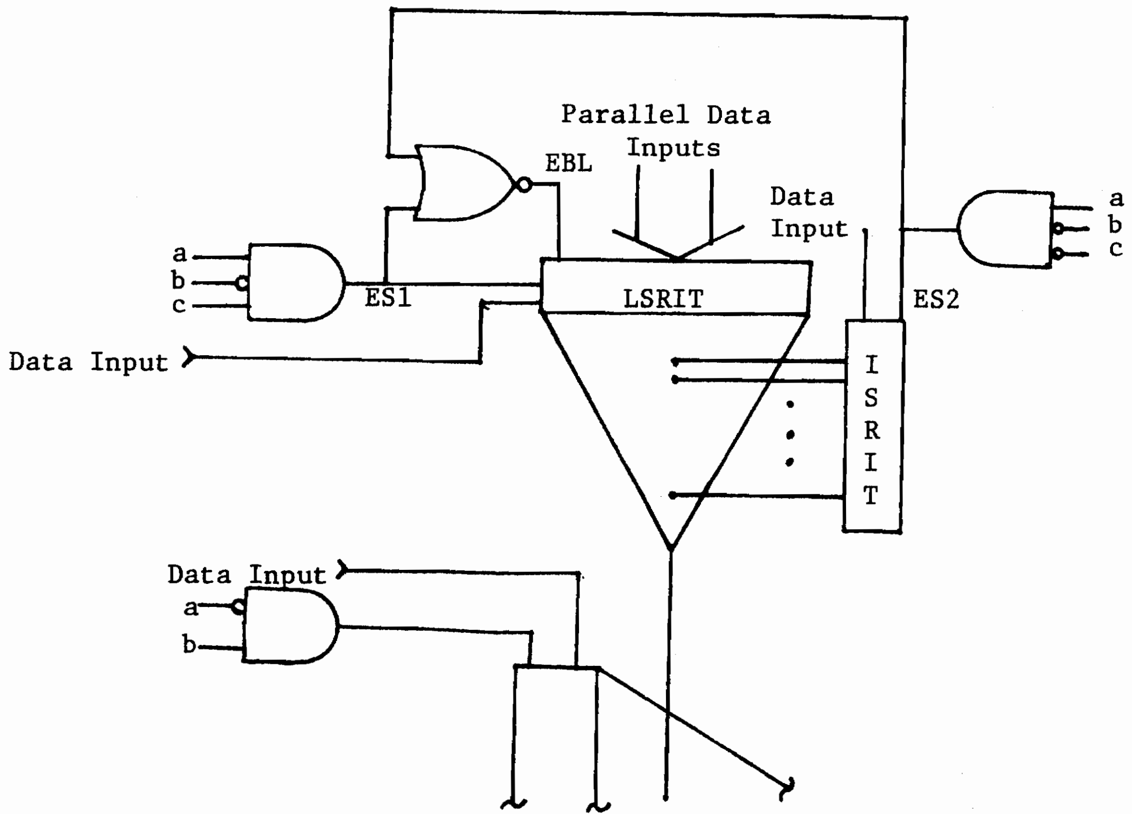


Figure 3.8: Combinational Data Steering Tree attached to the Input of The Generalized Modular Tree.

Signal	Definition
a	Produced by "control tree" 1.
b	Produced by "control tree" 2.
c	Produced by "control tree" 3.
ES1	Enable Shift to the Leaf Register of Combinational Tree.
EBL	Enable Broadside Load.
ES2	Enable Shift to Input Register of Combinational Tree.

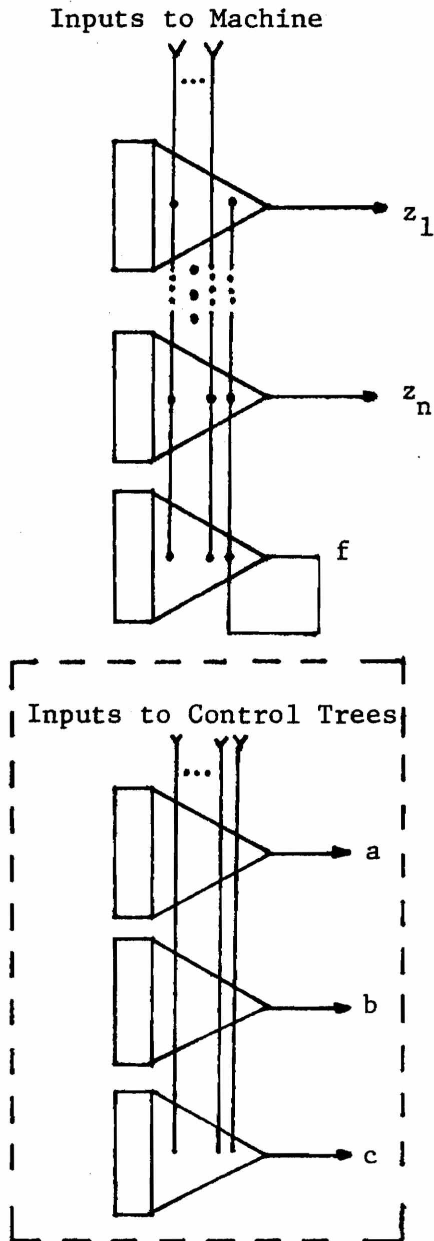


Figure 3.9: The "control trees" as an added section of the machine.

Table 3.3: Codes used in Combinational Logic to enable various circuits.

CODES Tree Number			Abrev. of Register Enabled	Leads that are to be enabled by the given code
1	2	3		
0	0	0	none	Must remain unused for testing purposes
0	0	1	LSRLSR	Enable Leaf Shift Register of Combinational Tree attached to Leaf Shift Register of Generalized Tree
0	1	0	ISRLSR LSR	Enable Input Shift Register of Combinational Tree attached to Leaf Shift Register of Generalized Tree Enable Leaf Shift Register of Generalized Tree
0	1	1	LSR	Enable Leaf Shift Register of Generalized Tree
1	0	0	ISRIT	Enable Input Shift Register of Combinational Tree attached to the Input of the Generalized Tree
1	0	1	LSRIT	Enable Leaf Shift Register of Combinational Tree attached to the Input of the Generalized Tree
1	1	0	ISRLSR	Enable Input Shift Register of Combinational Tree attached to the Leaf Shift Register of the Generalized Tree
1	1	1	none	Must remain unused for testing purposes
a	b	c		
Output of Control Trees				

said tree. If a code of 011 were applied, this data would be shifted into the LSR. The Combinational Data Steering Tree of Figure 3.8 would act in a similar manner. The only difference would be in the codes that would be applied to the combinational logic and in the effect of the output of the Combinational Tree on the generalized tree. Essentially it is the purpose of these trees to select a single data input to the generalized tree (at any one point in time) from the multitude of potential inputs (one input for every leaf on the Combinational Data Steering Tree). The address of this particular path (through the Data Steering Tree) is contained in the Input Shift Register to the given Data Steering Tree.

Actually there must be at least two such groups of control trees. Figure 3.10 is a diagram of the logical arrangement of two sets of trees within a machine; while Figure 3.10a shows the way in which the various trees in 3.10 are related. With this type of structure, if just one of the "control trees" in either section fails in such a manner as to not allow the production of the correct outputs, all the trees in both Groups I and II must be logically eliminated from the machine.

Such a statement immediately brings to mind the question of trade offs. How many trees should be placed in a computational group which is controlled by three trees?

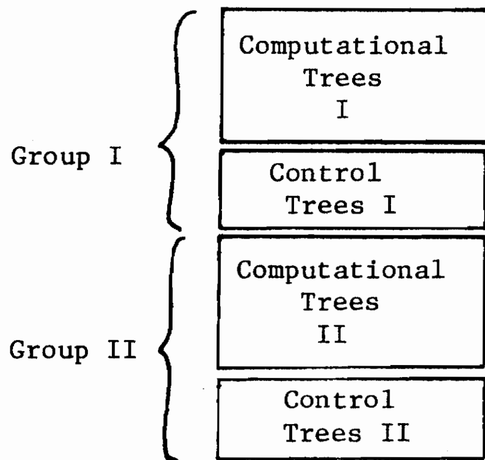


Figure 3.10(a): Group Concept of Control and Computational Trees Arranged in Pairs.

Control Tree Group	Sphere of Influence
I	Computational Trees I Control Trees II
II	Computational Trees II Control Trees I

Figure 3.10(b): Spheres of Influence of Trees within a Control Block.

This is a question which can only be answered by the user or by the system designer. In general there is, of course, no "right" answer. If the probability of a "disabling" fault occuring is fairly small, then a large number of computational trees should form a computational block. With the current state of LSI processing technology, the assumption of low probabilities of failure is probably valid. Another trade off may be found in the number of parallel output lines, from trees under test, which can be routed out of the machine simultaneously. In so far as this work is concerned, the latter will not be considered a problem. It will be assumed that any number of output lines may be brought out simultaneously.

4. Testing the Modified Tree Structure

With the addition of all the new structure to the generalized modular tree, it has become necessary to modify or enhance the testing algorithm given in Chapter 2. This algorithm was developed to test only the tree structure itself. It was seen that it would actually test the modularized tree structure with a shift register attached to the leaves. Since that time, numerous additions have been made to the tree structure. Data steering trees were added to the tree structure and combinational logic was added to control these trees.

Two means were suggested for controlling the combinational logic. The first was to allow a master controller to control the data flow in the structure. As this controller was "outside" the tree structure and was probably a set of trees which were assumed to be good (they had been recently tested), it was outside the scope of this work to study their implementation. The second means given for controlling the data flow in the modified tree structure was the addition of a set of special "control trees" and the attendant combinational logic necessary to control the data flow. The former means of control will not be discussed as it can be included in the latter because the outputs of the

AND gates are equivalent to the inputs from the master controller. This implies that if the testing algorithm developed will test the path from the "control trees" to the various control inputs on the shift registers, it will also test these same inputs when they are controlled by some master controller.

4.1 Testing the Generalized Tree:

In order to test the data steering networks, it must first be determined that the generalized modular tree is fault free. If this were not done, then a fault in the generalized tree might mask a fault in one of the data steering trees. It will be shown that a fault in a data steering tree will not allow the generalized tree to test out as being good. In order to test the generalized modular tree it is necessary to show that there exists a fault free data path from the tree generating the testing sequence, given in Algorithm 2.2, to the first stage of the shift register of the tree under test. It may be assumed that the tree generating the sequence is fault free. If the testing algorithm works, the generating tree may be tested before it is assigned the task of generating the testing sequence. An alternative is that the sequence could be examined by a tree outside the group being tested to determine if it is

correct. In either case a correct sequence will be applied to the data path, or the data path will be switched to the output of another tree which produces the correct sequence. It can therefore be assumed that a correct sequence is being produced and applied to the data steering network. If there is a stuck-at fault in the data path from the sequence generator to the shift register of the generalized modular tree, then this fault could be modeled as being a stuck-at fault at the input to the first stage of the generalized tree's shift register, in so far as the present testing algorithm is concerned. It has already been shown that such a fault will be detected. Therefore, it may be assumed that either the data path is fault free or the tree will be found faulty. In either case, the tree may be tested. So far, the first half of the testing experiment on the generalized tree can be performed, i.e. the sequence c can be placed on the leaves.

A closer study of the two patterns c and $\bar{c}$ reveals that the pattern $\bar{c}$ is actually the pattern c starting in the middle and wrapping around to include the first half of the sequence. In other words, $\bar{c}$ may be generated by placing pattern c in a circular shift register of length $2^{\mu k}$ and shifting $2^{(\mu k-1)}$ times. The latter approach is particularly appropriate as the pattern c is already contained in the leaf shift register of the generalized modular tree. In

fact, there is no reason why the testing pattern can not be shifted $2^{(\mu k-1)}$ times and the output of the register be brought to the input. To do the latter, all that is necessary is for the output of the shift register to be attached to one of the leaves of the data steering tree which is attached to the leaf shift register of the generalized tree as shown in Figure 4.1. If there is a fault along this path, it would either be a stuck-at fault at the output of the leaf shift register or within the combinational tree doing the data steering. In either case a stuck-at fault would be found as it was when c was input. Now, it has been shown that the modified generalized tree without including the data steering trees can be tested with both c and $\bar{c}$ and that this major portion of the modified structure can therefore be totally tested.

4.2 Testing the CDSTLSR:

As the generalized modular tree itself may be totally tested under the present modifications, if it is found fault free then it may be used in testing the rest of the modifications. The next section of the modified structure to be tested is the data steering (combinational) tree attached to the generalized tree's leaf shift register (CDSTLSR). So far, at least two of the data paths through

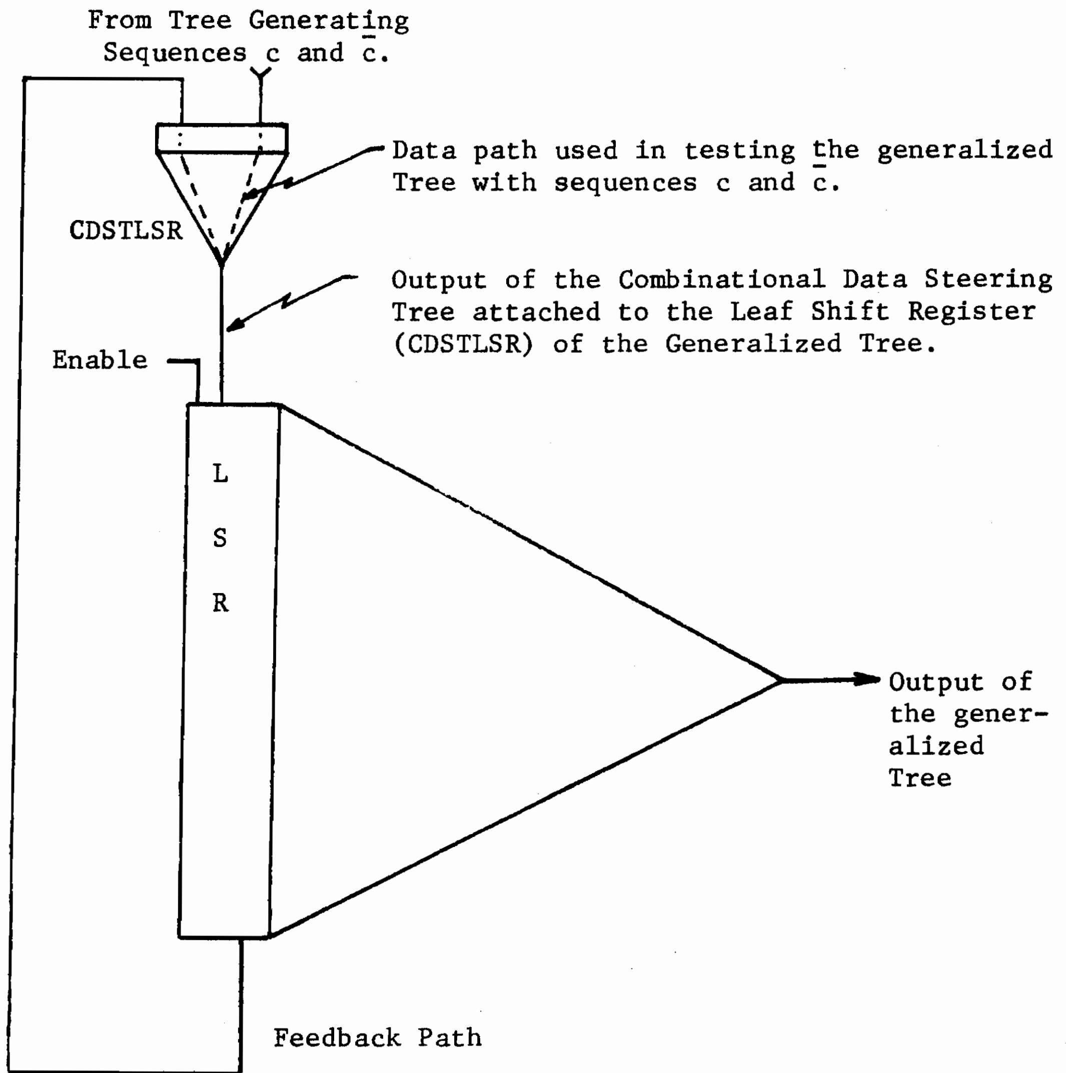


Figure 4.1: The two data paths through the CDSTLSR used in testing The Generalized Tree.

this network have been tested. Presumably there will be more than just two such data paths, consequently an addition to the testing algorithm is necessary for the testing of this tree. As this data steering tree is strictly combinational in nature, Algorithm 2.1 can be used to develop the testing sequence. In the case of a combinational tree μ is the number of inputs. Therefore, in this case, is the length of the input shift register of the CDSTLSR (hereafter abreviated as ISRLSR). The testing sequence C (designed for a tree with the number of inputs equal to the length of the ISRLSR) is shifted into the LSRLSR (Leaf Shift Register of the CDSTLSR) when the "control trees" output the code 001. The broadside load is disabled at this time so that the sequence C may be entered without the parallel inputs interfering with it. Immediately after C has been entered, the "control trees" must output the code 010 which enables both the ISRLSR and the leaf register of the generalized tree under test (LSR). The sequence which should be entered in the ISRLSR is that given in Experiment 2.1, consisting of all subsequences of length equal to the length of the ISRLSR (LL). This input sequence accesses the information on the leaves of the tree and routes the information into the shift register of the generalized tree. This information may then be accessed and checked for correctness. In fact, the information may be

accessed while it is being entered into the shift register by entering in 0...0 on the input lines to the tree if the flip-flops internal to the tree (assuming the tree to be sequential) are of the NOR type construction given earlier. This type of D flip-flop follows the input until the clock goes low. Accessing the data from the CDSTLSR while it is being entered into the LSR is probably the best way in which to perform the test as there is only a delay of μ-1 time units between when the data is entered and when it is output for checking. The latter method will be used in the testing algorithm. As with the testing of the generalized tree, only the first portion of the testing pattern must be shifted into the LSRLSR in order to get the complement of the original pattern. In this case however, a circular shift register can not be used to implement the complemented pattern as there is effectively only one input to the shift register. The CDSTLSR should now be checked with the pattern $\bar{C}$ using the same procedure as was used with pattern C.

It is now necessary to check the broadside load for proper operation. There are three catagories of inputs to be checked. They are as follows: those inputs already checked, which can be ignored; those inputs which are constants, which need be checked only once; and the "normal" inputs which must be checked with both a 1 and a 0 on the

inputs. The first two catagories may of course be checked twice but they should not be expected to be different. It is not necessary to check for a stuck-at fault on either the constant 1's or 0's more than once, as they are supposed to remain constant. Those inputs which have already been checked need not be checked again. An example of such an input is that which comes from the last stage of the LSR (as shown in Figure 4.1). It might not be very convenient to have this stage produce the required output at the exact time that such an output is necessary. Yet it is known that this data path is good as it has already been tested by shifting the data into the leaf shift register. Such inputs might be located so as to allow the testing sequence to be shortened. This implies that the two data paths which are used to test the generalized tree could be attached to the two leaves of the LSRLSR which are called last by the testing sequence being entered into the ISRLSR. In this way the first pass of the testing sequence could be shortened by two. If the constants were also positioned in such a manner, then the second pass of the testing sequence could be further shortened. The latter however might not be as advantagous as it might first appear. If the constants were to be used for fault location in addition to fault detection such placement might not yield any new information and since the decrease in length of the testing sequence is relatively

negligible, shortening the testing sequence might make the entire algorithm longer. For this reason the testing sequences entered into the ISRLSR will not be shortened in the testing Experiments which follow. As the actual structure of the combinational data steering tree has already been tested, it is only necessary that two patterns be applied to the leaves of the CDSTLSR; one a complement (not including the already checked inputs or constants) of the other. Let these broadside loaded testing patterns be called s and s'. If a tree that is known to be good has exactly the same inputs as the tree under test then this "good" tree can be used to provide an output sequence which can then be compared to that of the tree under test. If there is a discrepancy between the two then the one under test is faulty. After both patterns s and s' have been used to test the CDSTLSR and no error has been found then the CDSTLSR (not including the combinational logic) can not be faulty. This means of checking the CDSTLSR requires that the parallel inputs of the LSRLSR be at specific values at only 2 points during the test. At all other times they may be allowed to vary. The exact patterns for s and s' are not important and may in fact be anything, as the combinational tree has already been tested. The patterns s and s' are used only to test the parallel inputs to the data steering tree. With the testing of the broadside load capability of

the data steering tree the modified tree structure has been tested except for the data steering trees attached to the inputs of the generalized modular tree and the combinational logic.

At this point the following tests have been made. The generalized tree has been tested with patterns c and $\bar{c}$ obtained from Algorithm 2.2 and the CDSTLSR has been tested with patterns C and $\bar{C}$ obtained from Algorithm 2.1. If no fault is encountered during the latter test then the CDSTLSR (not including the broadside inputs and combinational logic) must be fault free. Next the CDSTLSR was tested with both patterns s and s'. If it was found to be fault free then the CDSTLSR (excluding combinational logic used to enable the registers) is fault free.

4.3 Testing the CDSTITs:

After testing the generalized tree and the data steering network attached to the leaf register, the next step is to test the data steering trees on the inputs of the generalized tree (each of which will be abreviated as CDSTIT and refered to in plural as CDSTITs. These are shown in Figure 3.3). It is first necessary to generate a testing pattern for the leaves of the generalized tree so as to allow the output to, in some way, show the proper operation

of the input trees. The testing of the CDSTITs is done in eight major steps for all trees with more than one input (one input trees may be tested in four steps). The experiment will be given in a more complete manner after the explanation which follows. If all data steering trees were to be tested simultaneously a great increase in speed would be obtained. There is however a problem with this approach. If all the trees have exactly the same fault then the outputs would be the same even though they were incorrect. Such a situation, if were to occur, would allow the tree to test out as good even though the structure was actually faulty. This might suggest that each of the data steering trees should be individually checked. The latter is not necessary if an addition is made to the testing algorithm. That addition would be to check the output produced by a single tree after having checked all of the trees together. It would also be well worth while, with regard to time, to use the same leaf pattern in the LSR for doing as much of the checking as possible. The parallel input capability of the CDSTITs also needs to be checked during the testing operation. To meet all these constraints is difficult at best. The following, Algorithm 4.1, generates a pattern which is to be placed on the leaves of the generalized tree in order to perform the test on the CDSTITs.

Algorithm 4.1:

(i) Set B equal to 0 concatenated with d, where d is the pattern 011...110 of length 2^k (where the center region consists of all 1's).

(ii) Set b equal to d.

(iii) Set i=2.

(iv) If $i>\mu$ then go to (x).

(v) Let b' equal the string formed by replacing every 0 in b by 2^k 0's and every 1 in b by 2^k 1's.

(vi) Set B equal to B concatenated with b'.

(vii) Set b equal to b'.

(viii) Let i=i+1.

(ix) Go to (iv).

(x) Print patterns B and b.

(xi) Stop.

Example 4.1:

Let $\mu=3$ and k=2 which implies K=4. Also let d=0110.

i	b	B
2	0110	0 0110
3	0000 1111 1111 0000	0 0110 0000 1111 1111 0000
4	0000 0000 0000 0000 1111 1111 1111 1111	0 0110 0000 1111 1111 0000 0000 0000 0000 0000 1111 1111 1111 1111

1111 1111 1111 1111   1111 1111 1111 1111
0000 0000 0000 0000   0000 0000 0000 0000

Figure 4.2(c) shows a tree (μ=2 and k=2) with the initialization and leaf pattern B contained in the tree.

The testing algorithm for the CDSTITs begins with the computation of pattern b using Algorithm 4.1. This pattern is then shifted into the LSR and the generalized tree is initialized. The actual process of initialization will be covered in the following chapter. Next, after having determined the length (LI) of the input register of a CDSIT (ISRIT) the patterns C and $\overline{C}$ are calculated using Algorithm 2.1. These patterns are the ones Prasad and Gray [9] developed for testing combinational trees. The pattern C is then shifted into the leaf registers of all of the CDSTITs (which will in singular be refered to as LSRIT and in plural as LSRITs) and the sequence of all subsequences of length equal to that of an ISRIT (call this sequence T) is then applied to the ISRIT data inputs of all the CDSTIT of the structure under test. If any of the outputs of the generalized tree are 1's then there is a fault in at least one of the CDSTITs. Figure 4.2 shows the data flow under both fault free and faulty conditions. The trees are then similarly tested with the pattern $\overline{C}$. Again a 1 at the output of the generalized tree means that a fault has been detected. Instead of proceeding to test an individual tree

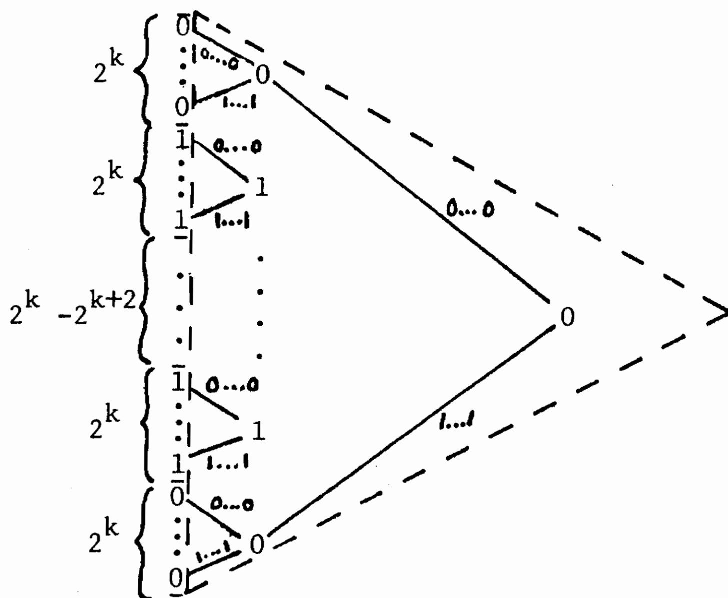


Figure 4.2(a): Data flow in a Generalized tree of depth 2 and k inputs When either all 1's (a K-ary K-1 digit) or all 0's (a K-ary 0) are applied to the inputs.

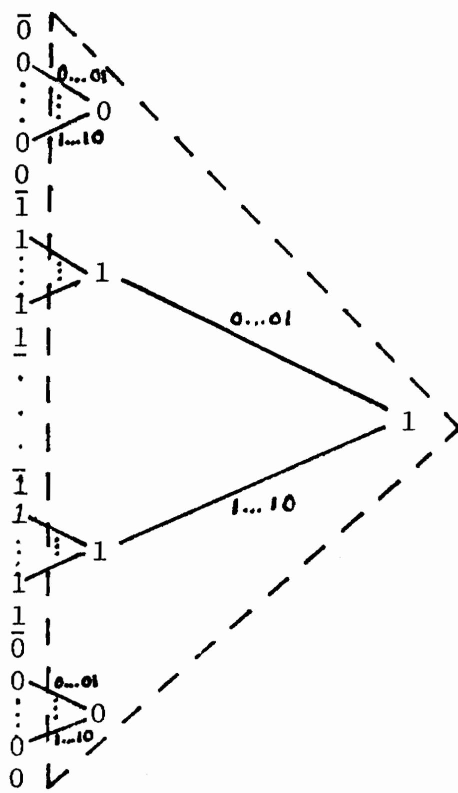


Figure 4.2(b): Data flow in the tree of part a, with at least 1 but Not all the CDSTITs in error.

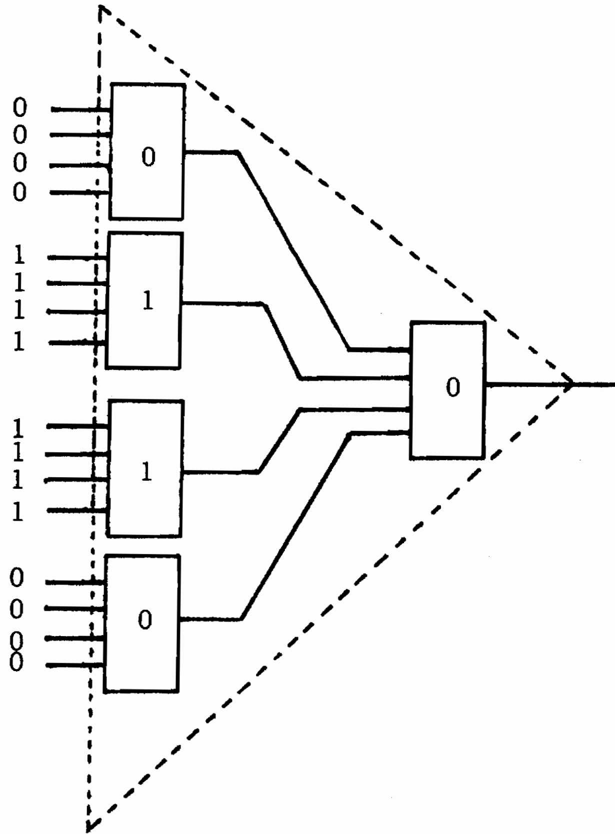


Figure 4.2(c): Sequential tree with μ=2 and k=2 which has been Initialized.

as the next step in the testing process, the broadside inputs will be tested next. The reason is simply that the testing of the broadside inputs requires no change in the leaf inputs, where as testing the single tree does require such a change.

Testing the broadside inputs (BSIs) of the CDSTITs is very similar to that of the combinational tree attached to the leaf shift register of the generalized tree (LSR) in that two patterns s and s' must be generated. It differs in the fact that the output of the generalized tree should remain a constant 0 if the steering trees are not faulty or if they all have the same fault. Therefore s is entered into the LSRITs and the sequence T is applied. While T is applied the output is scanned for 1's. If no 1's are encountered then s' is entered into the LSRITs and T is once again applied. If any 1's occur in the output string there is a fault.

If all the CDSTITs are good then with exactly the same inputs applied to each, it is obvious that all the outputs will be the same and a 0 will be output from the generalized tree (see Figure 4.2(a)). If there exists at least one good tree and one faulty tree in the set of CDSTITs of the generalized tree then the output sequence for the two trees will not match and the generalized tree will output a 1 (see Figure 4.2(b)). If all the trees are faulty and manage to

escape the test in which all the outputs are examined to see if they are the same, then when a random tree is checked alone, it will be found faulty.

If at this time no errors have occured, then either all the CDSTITs are good, or they all have the same fault. To test for the latter condition, it is necessary to test only one tree while the other trees output a constant. The same test procedure will be used as above except that the leaf pattern in the LSR will be modified somewhat. This modification takes the form of shifting $[2^{k(\mu-2)}+1]$ ones followed by $2^{k(\mu-1)}$ zeros into the leaf shift register of the generalized tree. This tree must then be reinitialized. If data flow is incorrect one of two conditions can occur. The first condition is when the data paths referenced have incorrect constants on them. In this case, the error will immediately be found. The second condition is if the data paths have the correct constants but are still not either the all 0 data path or the 0...01 data path. In this case the tree will still be checked out normally and will be found faulty as all the CDSTITs must either have the same fault or be fault free. The leaf pattern and proper data flow are shown in Figure 4.3.

Because of the structure of the combinational logic it will be necessary to reenter s'. Additionally the inputs to the ISRITs other than the one closest to the output must all

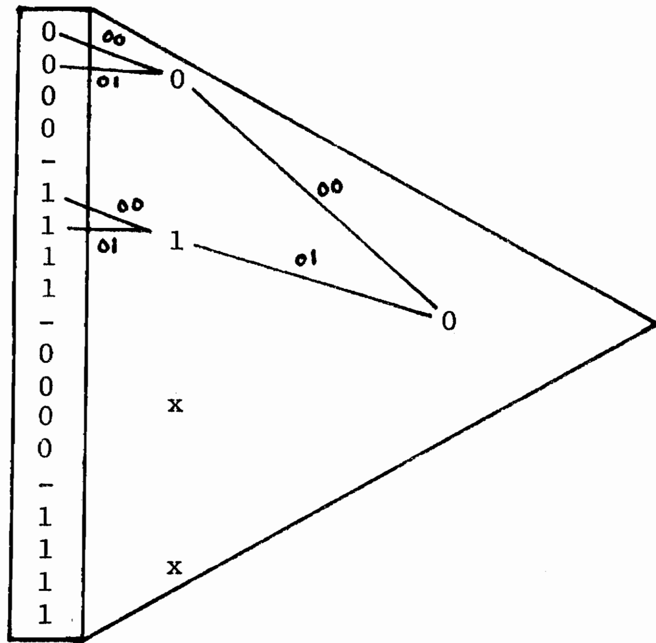


Figure 4.3: Correct Data Flow Through a Generalized Tree with =2 and k=2.

be zeros and a 0 must be the constant which is applied to the right-most input of the LSRITs (this is the data path accessed by all zeros in the ISRITs). The ISRIT closest to the output will have the sequence T applied while the other ISRITs are have a constant 0 applied. With the above conditions the pattern generated by the calling sequence T and the inputs to the leaves, s', will be output by the generalized tree. The output sequence of the generalized tree under test may be checked against another such sequence from an already tested tree. If they differ then the structure under test is faulty. If the output sequence is not incorrect then s must be reapplied to the LSRIT and the test must be performed again. The latter two broadside inputs, s and s', do not have to be the same as their predecessors but the same relationship between s and s' holds. The pattern C must once again be shifted into the LSRIT. This time however, only the tree closest to the output should receive this pattern. All the other LSRITs should have a string of 0's shifted in. T will again be applied to the single CDSTIT under test. The output will be C as accessed by T and may be checked as above. The same test is repeated for $\bar{C}$. After having performed all the above tests, the new structure has been almost completely tested. The testing Experiment so far is given below.

Experiment 4.1:

(i) Compute b from Algorithm 4.1.

(ii) Shift b into the LSR of the generalized tree.

(iii) Initialize the tree.

(iv) Using Algorithm 2.1 with μ replaced by the length of ISRITs, find C and $\bar{C}$.

(v) Find the shortest sequence containing all subsequences of length equal to that of ISRIT and call it T.

(vi) Shift pattern C into the LSRIT of each CDSTIT.

(vii) Apply T in parallel to the ISRIT of each CDSTIT under test and note the outputs.

(viii) If the output string of the generalized tree contains a 1 then go to (xxxii).

(ix) Shift the pattern $\bar{C}$ into the LSRIT of each CDSTIT.

(x) Apply T to the ISRIT of each CDSTIT and note the outputs.

(xi) If the output string of the tree under test contains a 1 then go to (xxxii).

(xii) Broadside load each CDSTIT with pattern s.

(xiii) Apply T to the ISRIT of each CDSTIT and note the outputs.

(xiv) If the output string contains a 1 then go to (xxxii).

(xv) Broadside load each CDSTIT with s'.

(xvi) Apply T to the ISRIT of each CDSTIT and note the outputs.

(xvii) If the output string contains a 1 then go to (xxxii).

(xviii) Shift $[2^{k(\mu-2)}+1]$ ones followed by $2^{k(\mu-1)}$ zeros into the LSR and initialize the generalized tree.

(xix) Broadside load all the CDSTITs with s', then initialize the ISRITs to all 0's.

(xx) Apply T to the ISRIT of the tree closest to the output while applying 0's to all the other ISRITs. Note the outputs.

(xxi) If the output string differs from the one generated by a good tree with the same inputs then go to (xxxii).

(xxii) Broadside load pattern s into each of the LSRITs. of the generalized tree.

(xxiii) Apply T to the ISRIT of the tree closest to the output while applying 0's to all other ISRITs. Note the outputs.

(xxiv) If the output string differs from one generated by a good tree with the same inputs the go to (xxxii).

(xxv) Shift pattern C into the LSRIT of the tree closest to the output of the generalized tree while shifting 0's into all the other LSRITs.

(xxvi) Apply T to the ISRIT of the tree closest to the output of the generalized tree and 0's to all the other ISRITs.

(xxvii) If the output string differs from one generated by a good tree with the same inputs then go to (xxxii).

(xxviii) Shift pattern $\overline{C}$ into the LSRIT of the CDSTIT closest to the output while shifting 0's to all the other LSRITs.

(xxix) Apply T to the ISRIT closest to the input and 0's to all the other ISRITs. Note the outputs.

(xxx) If the output string differs from one generated by a good tree with the same inputs then go to (xxxii).

(xxxi) Stop. All CDSTITs are fault free.

(xxxii) Stop. There exists at least one faulty tree.

The above experiment assumes that all the inputs to the CDSTITs are identical and that there is a constant 0 on the leaf addressed by all zeros. It also assumes that there exists a set of good trees somewhere and that one of them can be loaded with the same leaf pattern as the tree under test. The data inputs to the LSRITs must be independent of each other and so must the data inputs of the ISRITs.

Actually the above experiment will work quite well if a set of trees were to be processed simultaneously instead of just one.

4.4 Testing the Combinational Logic:

At this point all the components of the proposed structure have been tested except the combinational logic elements. These elements can fail only to the following modes: stuck-at fault enabling a device at all times, stuck-at fault disabling a device at all times, stuck-at fault enabling a device more often than the control inputs call for, stuck-at fault disabling a device more often than is ordered by the control inputs. The term "control inputs" will be used to mean those inputs to the combinational logic which come from the set of control trees. In the rest of the chapter the following abreviations will be used: LL to mean Length of the ISRLSR, LI to mean the length of the ISRIT, s-a to mean stuck-at, and s-a-f to mean stuck-at fault. Figures 4.4 and 4.5 show both portions of the combinational logic which need to be tested. The numbers entered near the interconnection lines are the numbers which will be used in refering to these leads.

The following information will be very useful in developing a set of tests which will test the circuit [11].

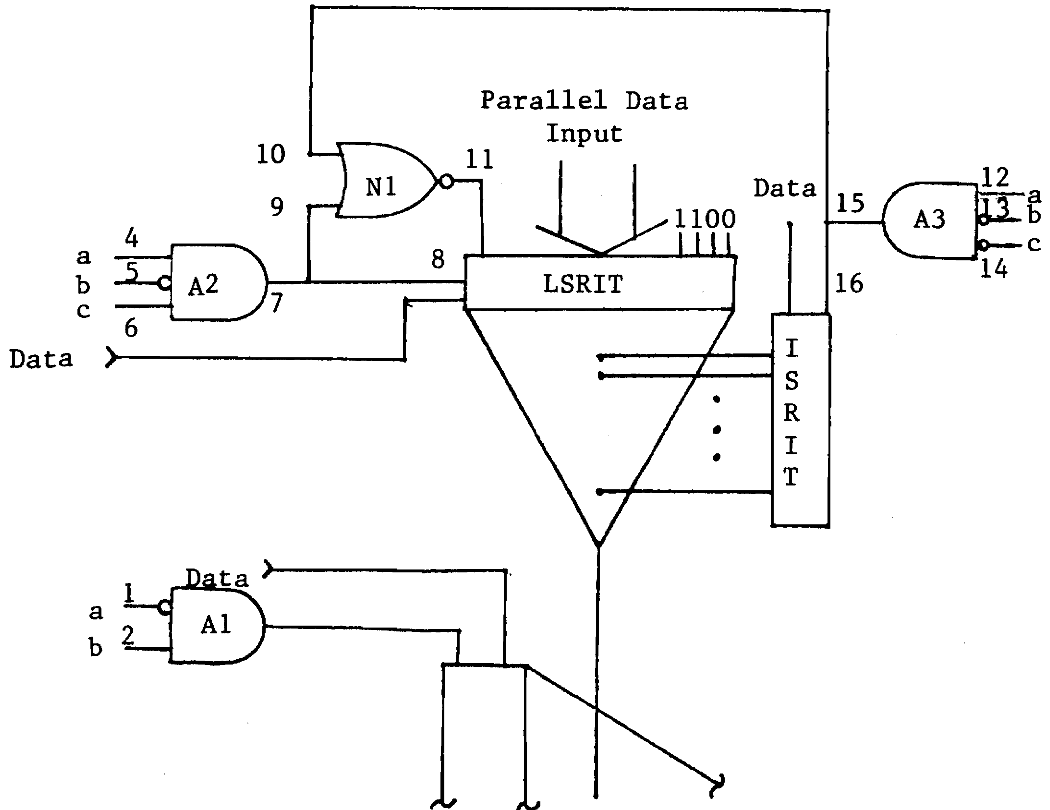


Figure 4.4: CDSTIT with attendant logic and all combinational lines Numbered.

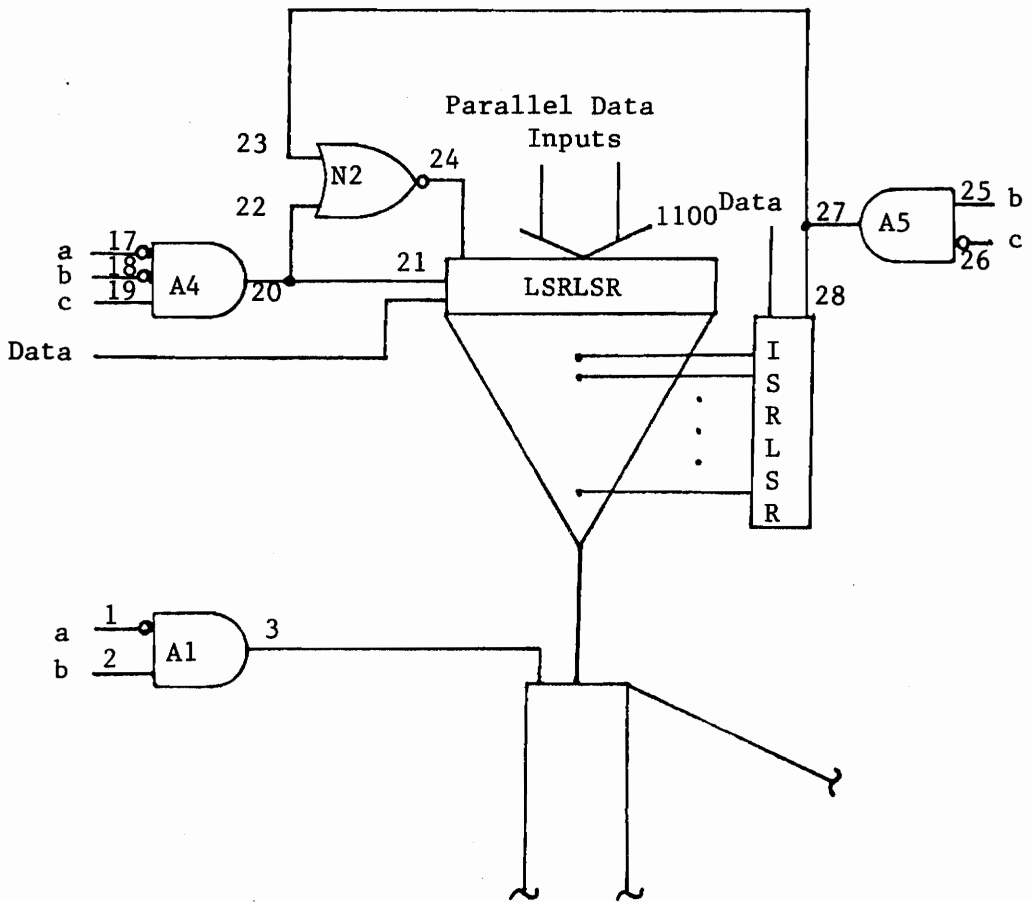


Figure 4.5: CDSTLSR with attendant logic and all combinational lines Numbered.

1) In a fanout free combinational circuit C, any set of tests which detects all s-a-f on primary inputs will detect all stuck faults. [Theorem 2.2 page 57]

2) In a combinational circuit, any test which detects all single (multiple) s-a-f on all primary inputs and all branches of fanout points, detects all single (multiple) faults. The set of primary inputs and branches of fanout points are called checkpoints of the circuit. [Theorem 2.3 page 58]

3) In irredundant two-level combinational circuits, any set of tests which detects all single s-a-f also detects all multiple s-a faults. [page 65]

4) The checkpoints of a combinational circuit consists of all inputs and branches of fanout points. A set of tests which will detect all multiple s-a faults on the checkpoints of a circuit will detect all multiple s-a faults in the circuit. [page 66]

5) In a combinational circuit C, any set of tests which detects all single s-a faults will also detect all multiple s-a faults unless C contains a subcircuit corresponding to the circuit shown within dotted lines in Figure 4.6. [page 66]

In looking at Figure 4.6 it should be immediately noted that this circuit is 3-level logic, whereas the circuits of

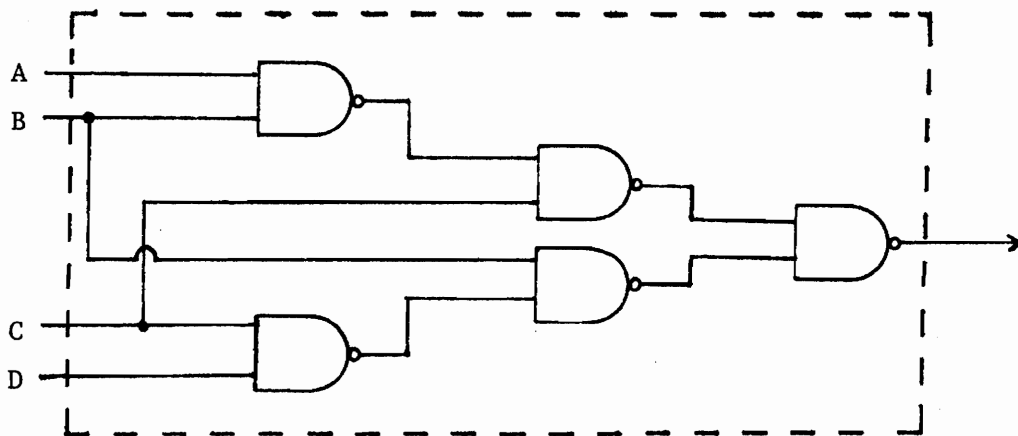


Figure 4.6: Combinational subcircuit which Violates Statement 5.

Figures 4.4 and 4.5 are 2-level logic. The latter two circuits can therefore not correspond to the 3-level circuit of Figure 4.6 and therefore the testing of the combinational logic may be based on a set of tests which will detect all single s-a faults. statement may be used. Perhaps before beginning the test sequence for the combinational logic, it would be best to see if these tests could best be placed within the body of the testing experiment that has been developed so far. It would also be helpful to know what tests are already being conducted on the combinational logic in the normal course of the testing experiment as it presently stands.

It would greatly simplify the testing of combinational faults if it were known that a certain gate or line was active when the code from the control trees activating that gate was applied to said gate or line. Therefore, to begin, an example will be given, for each gate, where the testing algorithm can not work properly (and therefore an error will be detected) if a given gate is not active at the appropriate time. If gate A1 is not activated on codes 010 or 011 then the pattern c can not be shifted into the LSR for the test of the generalized tree. Assuming c is already on the leaves then $\overline{c}$ can not be applied. If gate A2 is not activated when code 101 is applied, the LSRIT will not be able to enter the testing patterns C and $\overline{C}$ at the proper

times. Actually, because of fanout constraints, to perform the actual shifting only line 8 must be active when this code is applied. Still, line 8 must be active when code 010 is applied to A2. Line 16 must be active when code 100 is applied to gate A3 in order to perform the tests on pattern C and $\bar{C}$ in the LSRITs. If line 21 is not active when code 001 is applied then C and $\bar{C}$ can not be entered into the LSRLSR and a fault will be detected. Line 28 must be active on codes 010 and 110 as, if this does not occur, then the tests on patterns C and $\bar{C}$ will fail. A s-a 1 fault on an input to gate N1 will have exactly the same effect as a s-a 0 fault on the output of N1 because of the truth table of this gate. Therefore, if a s-a 0 fault on line 11 will be detected so will a s-a 1 fault on lines 9 and 10. Using the same reasoning on gate N2, s-a 1 faults on lines 22 or 23 will be detected if a s-a 0 fault on line 24 is detected. Looking at the structure for the shift register (Figure 3.4), it can be seen that since the inputs from the broadside load overide the inputs from the d input to the flip-flops, if lines 11 or 24 were s-a 0 then a broadside load could never occur. Line 24 s-a 0 would mean c and $\bar{c}$ could not be entered into the LSR. In either case, a fault would be detected. Let line 24 be s-a 1, then the testing pattern C for the CDSLSR could not be entered (broadside load overides the shifting of data). If the constants 1100

were to be place on leaves 0 through 4, as shown in Figures 4.4 and 4.5, then an error would be detected as the beginning 4 bits of pattern C can not contain this pattern when constructed according to Algorithm 2.1. Similarly, a s-a 1 fault on line 11 would be detected when the CDSTITs were tested. Therefore a s-a p fault (where p is either 1 or 0) will be detected on both lines 11 and 24. From the reasoning given above, if any of the lines 9, 10, 22, or 23 are s-a 1 then a fault will be detected.

By Statement 4, special tests need not be developed for the following output lines: 7, 15, 20, and 27, if all the other checkpoints are tested. Additionally, by the above considerations, lines 3, 8, 11, 16, 24, and 28 s-a 0 will be detected in addition to lines 9, 10, 11, 22, 23, and 24 s-a 1. The rest of this discussion will be based on the assumption that the input to the input shift registers of the CDSTLSR and all the CDSTITs will be 0 when not in use during testing and that the inputs to the leaf registers of all the combinational data steering trees will be 1 when no special sequence is being applied during testing. As will be seen in Experiment 4.2, this assumption will be very helpful in testing the combinational logic. The following Experiment is a combination of all the other experiments and is adapted so as to test the combinatinal logic as well as the trees. The following codes may be found in Tables 3.3

Table 4.1: Codes used in Combinational Logic to enable various circuits.

CODES Tree Number 1	2	3	Abrev. of Register Enabled	Leads that are to be enabled by the given code
0	0	0	none	Must remain unused for testing purposes
0	0	1	LSRLSR	Enable Leaf Shift Register of Combinational Tree attached to Leaf Shift Register of Generalized Tree
0	1	0	ISRLSR LSR	Enable Input Shift Register of Combinational Tree attached to Leaf Shift Register of Generalized Tree Enable Leaf Shift Register of Generalized Tree
0	1	1	LSR	Enable Leaf Shift Register of Generalized Tree
1	0	0	ISRIT	Enable Input Shift Register of Combinational Tree attached to the Input of the Generalized Tree
1	0	1	LSRIT	Enable Leaf Shift Register of Combinational Tree attached to the Input of the Generalized Tree
1	1	0	ISRLSR	Enable Input Shift Register of Combinational Tree attached to the Leaf Shift Register of the Generalized Tree
1	1	1	none	Must remain unused for testing purposes
a	b	c		
Output of Control Trees				

and 4.1.

Experiment 4.2:

(i) Code 110 should be applied for LL time periods while the leaf code of the sequential tree which generates c is shifted into the ISRLSR. (If line 25 s-a 0 or line 26 s-a 1 then this step can not be carried out and a fault will be found. If the sequence in the ISRLSR is not that of the tree which generates c then c can not be entered into the LSR and the test for the generalized tree will detect a fault in step (iv). If the proper generater tree is addressed then $\bar{c}$ can not be entered as the contents of the ISRLSR must be changed to enter $\bar{c}$. Similarly, if line 28 s-a 0 then a fault will be detected in step (iv).)

(ii) Code 011 should be applied for 2^{nk} time periods while c is shifted into the LSR. (If line 26 is s-a 0 then the data path will be switched to the all zero path and the pattern entered into the LSR would be incorrect. This assumes that the LSR is longer than the ISRLSR. It should be remembered that a constant 0 is to be applied to all Input Shift Registers and a constant 1 to

all Leaf Shift Registers of the combinational data steering trees when they are not having any particular sequence entered into them. If lead 1 is s-a 1 or lead 2 is s-a 0 then shifting will not occur and a fault will be detected in step (v) or step (vii). If lines 22 or 23 are s-a 1 or line 24 is s-a 0 then c could not be entered and a fault will be detected at step (iv). If line 28 is s-a 1 then the data path will be switched and c will not be entered, a fault will be detected at step (iv).)

(iii) Code 100 should be applied to the ISRIT for LI time periods while the appropriate codes are being applied to the CDSTITs in order to steer the outputs of the trees generating the test sequence. (If line 12 is s-a 0 or lines 13 or 14 are s-a 1 on any of the CDSTITs then the appropriate generator output will not be selected. The pattern c will not be accessed properly. If 16 is s-a 0 the same effect is obtained. In all of the above cases a fault will be detected in step (iv). If by some chance the contents of the ISRITs are exactly the proper constants to address the leaves which have the proper testing sequence applied then

these s-a 1 faults will not be caught in step (iv). The problem will however be caught when the CDSTITs are required to change the leaf that they are accessing. The latter is done many times throughout the experiment. This condition would be caught in step (xii). The only requirement is that the trees generating the sequence output a constant 1 at that time.)

(iv) Code 000 should be applied for $2^{\mu k}+\mu-1$ time periods while the testing sequence of Experiment 2.2 is applied. Note the output after $\mu-1$ time periods. (If lines 2 or 3 are s-a 1 the the LSR will continue to shift and the pattern c will be modified. Note: make the output of the tree which generates c be a constant after c has been generated. Also some of this pattern (c) will be lost at the last stage of the shift register and could not therefore be accessed to form $\overline{c}$. The error will be detected in this step or step (vii). If line 16 is s-a 1 then the CDSTITs will not stay locked onto the output of proper generator and the testing sequence will not access c properly. This will be detected in this step. If lines 9 or 10 are s-a 1 or line 11 is s-a 0 then a broadside load can not occur

and the testing sequence will again not access c properly. A fault will be detected if any of the above s-a faults are present.)

(v) Code 110 should be applied for 2LL+3 time periods and a sequence of LL+3 zeros concatenated with the leaf number which will steer the output of the LSR to the input of the LSR should be entered into the ISRLSR. (If line 1 is s-a 0 then the LSR is shifted incorrectly and data is lost. A fault will be detected in step (vii).)

(vi) Code 011 should be applied for $2^{\mu k}-1$ time periods so as to enter $\bar{c}$. (If line 22 is not s-a 0 and if 18 is s-a 0 a fault will be detected in step (vii).)

(vii) Code 000 should be applied for $2^{\mu k}+\mu-1$ time periods while the test sequence of Experiment 2.2 is applied to the inputs of the generalized tree. Note the outputs after $\mu-1$ time periods. (This step tests the generalized tree with $\bar{c}$. If no faults are detected then the generalized tree is good.)

(viii) Code 100 should be entered for LI time periods while 0's are shifted into the ISRITs. (This will access a constant 0 on the CDSTITs. If all

the outputs of the CDSTITs are a constant 0 then the zeroth leaf of the generalized tree will be accessed.)

(ix) Code 110 should be applied for LL time periods and 1's should be applied to the ISRLSR.

(x) Code 001 should be applied for 2^{LL} time periods while shifting C, generated from Algorithm 2.1, into the LSRLSR. (If lead 22 is s-a 0 the broadside load will still be enabled and C will be incorrect. It will never be entered and because the constants 1100 have been applied to the parallel inputs of the LSRLSR the output of this register from a broadside load cannot be C. If line 21 is s-a 1 the pattern input will be further changed after entering of data should have ceased and will be incorrect. If line 21 is s-a 0 then C can't be entered into the LSRLSR at all and a fault will be detected in this step. In all cases, a fault will be detected in step (xii). Since checkpoints 21 and 22 can not be s-a p without the fault being detected, a s-a p fault on lead 20 will be detected. If line 17 or line 18 is s-a 1 or line 19 is s-a 0 C will fail to be shifted in unless 20 is s-a 1. Therefore these faults too will be detected in

step (xii).)

(xi) Code 110 should be applied for LL time periods while the LSRLSR is being set to the beginning of the testing sequence. (If line 23 is s-a 0 then the LSRLSR will be broadside loaded and the output of the test will be incorrect in step (xii). The same effect will occur if line 24 is s-a 1. In both cases a fault will be detected. As both checkpoints 23 and 28 s-a p will be detected, so will line 27 s-a p.)

(xii) Code 010 should be entered and the testing sequence should be applied for $2^{LL}-1$ time periods. The outputs of the generalized tree should be examined after $\mu-1$ time periods. [Remember that the all zero data path has been activated by the CDSTITs.]

(xiii) Code 001 should be applied for one time period while one 1 is entered into the LSRLSR. This shifts a 1 into the 0 leaf of the LSRLSR [Pattern C starts with 01....]

(xiv) Apply code 101 for 1 time period. Enter anything into the LSRIT. This should broadside load the LSRLSR. (If line 17 is s-a 0 then the 101 will keep a broadside load of the LSRLSR from occuring and the output of the leaf 0 (of

the LSRLSR) will be 1 instead of 0. The fault may therefore be detected in the next step.)

(xv) Apply code 010 for LL+μ time periods. Enter a constant 0 into the ISRLSR. Examine the output at time LL+μ [this is not LL+μ-1 because there is a 1 time unit delay associated with loading leaf 0]. [Broadside load is inhibited.] Note the outputs of the generalized tree. (If line 17 is s-a 0 then the output will be 1 otherwise it will be 0.)

(xvi) Apply code 000 for 3 time periods. [This should do nothing to any registers.] As the broadside load worked above (or a fault was detected), the LSRLSR will contain ...1100 before this command and the pattern should not change. (If line 19 is s-a 1 then there will be a 1 on leaf 0.)

(xvii) Apply code 010 for μ time periods, while entering 0's into the ISRLSR. Note the output. (If the output of the generalized tree at μ is 1 then a s-a 1 fault in line 19 is detected in this step.)

(xviii) Code 100 should be applied for LL-2 time periods. Zeros should be entered into the ISRIT while the sequence beginning with 1 and followed by 0's is applied to the ISRLSR. (This last

sequence should not enter the latter register.)

(xix) Code 011 should be applied for μ time periods and the output should be examined at time μ. (If line 25 is s-a 1 then the output will be 1 and a fault will be detected as the sequence in the step above was entered into the ISRLSR and so leaf 4 was accessed.) [At this point all the combinational logic for the CDSTLSR has been tested and so may be assumed to be good throughout the remainder of the experiment.]

(xx) Compute b from Algorithm 4.1.

(xxi) Apply code 110 for LL time periods while the leaf code for the tree producing the sequence b is entered into the ISRLSR.

(xxii) Apply code 011 for $2^{\mu k}$ time periods while b is being shifted into the LSR.

(xxiii) Apply code 000 for $\mu-1$ time periods to initialize the tree. The outputs of the CDSTITs should still be all 0's.

(xxiv) Using Algorithm 2.1 with μ replaced by LI, find C and $\bar{C}$.

(xxv) Code 101 should be applied for 2^{LI} time periods while C is being entered into each LSRIT. (If line 9 is s-a 0 then the broadside load is not disabled and C will not be entered. An error

will be detected in step (xxvi). If line 8 is s-a 0 then C will not be shifted in and an error will be detected in step (xxvi). If lines 4 or 6 are s-a 0 or line 5 is s-a 1 the C can not be shifted in and a fault will be detected in step (xxvi).)

(xxvi) Code 100 should be applied for $2^{LI}+LI-1$ time periods while T [the sequence containing all sub-sequences of length LI] is entered. Note the outputs after LI-1 time periods. (If line 10 is s-a 0 then the broadside load is not disabled and C will not be contained in the LSRIT. The fault will be detected. As both line 9 and 10 s-a p will be detected, by Statement 1 used in reference to gate N1 as a subcircuit, line 11 s-a p will be detected. As s-a p faults on lines 10 and 16 will be detected, so will a s-a p fault on line 15.)

(xxvii) If the output string of the generalized tree contains a 1 then at least one CDSTIT is faulty. Stop.

(xxviii) Code 101 should be applied for 2^{LI} time periods while $\bar{C}$ is being entered. [This assumes that $\bar{C}$ is produced by the same tree originally producing C. This must be true as there is only

one input to this register.]

(xxix) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is being entered. Note the outputs after LI-1 time periods.

(xxx) If the output string of the generalized tree contains a 1 then a fault has been detected. Stop.

(xxxi) Code 111 should be applied for 1 time period so as to enter s. (If line 5 is s-a 0 then a broadside load can't occur unless line 7 is s-a 0. As line 9 has already been tested and is not s-a p then line 7 can not be s-a 0 without having been detected in step (xxxii). Therefore line 5 s-a will be detected.)

(xxxii) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is being entered. Note the outputs. (If line 8 is s-a 1 then as 0's are being applied to the LSRIT the pattern s and s' will be partially the same in the high order locations of the register. If line 6 is s-a 1 then the s will be wrong; a shift will again occur. In the above cases, faults will be detected in step (xxxiii).)

(xxxiii) If the output string differs from one generated by a good tree with the same inputs then a fault

has been detected. Stop.

(xxxiv) Code 000 should be applied for 1 time period while s' is being entered into each LSRIT. (If line 8 is s-a 1 then a fault will be found as explained above. As neither line 8 nor line 9 can be s-a p without having a fault detected, by Statement 4 line 7 s-a p will be detected. If line 12 is s-a 1 then the broadside load can not occur and a fault will be detected in step (xxxvi).)

(xxxv) Code 100 should be applied for 2^{LI} +LI-1 time periods while T is being entered into each ISRIT. Note the outputs.

(xxxvi) If the output string contains a 1 then a fault has been detected. Stop.

(xxxvii) Code 000 should be applied for 1 time period in order to broadside load every LSRIT with s'.

(xxxviii) Code 001 should be applied for 2 time periods while 1's are entered into the LSRLSR. (If line 4 is s-a 1 then the broadside load of the LSRIT will be shifted two positions and a 1 will be placed on leaf 0.)

(xxxiv) Code 100 should be applied for LI time periods while 0's are entered into the ISRITs. Note the output at time LI. (The last output should

reflect a broadside load with leaf 0 having a value of 0. If line 4 is s-a 1, the output has the value 1 and an error has been detected.)

(xxxv) Code 100 should be entered for LI-2 time periods while the pattern composed of one 1 followed by all 0's is entered into each ISRIT. Before this occurs the ISRIT contains all 0's. (This should access a leaf with a 1 on it if a broadside load occurs. If the ISRIT is shifted, with 0's applied to the input, it will access a 0.)

(xxxvi) Code 110 should be applied for LL time periods while the string of one 0 followed by one 1 followed by all 0's is entered into the ISRLSR. Note the output of the generalized tree. [The output of the CDSTIT should be a 0 and the output of the generalized tree a 0.] (If line 13 is s-a 0 then a shift will occur and a 1 would be output from the CDSTIT instead of a 0.)

(xxxvii) Code 101 should be applied for 1 time period. This enters a 1 into the LSRIT. Note the output of the generalized tree. (If line 14 is s-a 0 then a shift will occur in the ISRIT and a 0 will be output from the CDSTIT instead of a 1. As an error has occured, line 14 s-a 0 will be detected. The output string of the generalized

tree would contain a 1.)

(xxxviii) Apply code 110 for LL time periods while the string of two 0's followed by one 1 followed by all 0's is entered. [This accesses a leaf with a constant 1 attached.]

(xxxix) Apply code 000 for one time period to broadside load the LSRLSR.

(xl) Apply code 011 for $2^{k(\mu-1)}$ time periods while the sequence of $2^{k(\mu-1)}$ ones is shifted in. A pattern of $2^{k(\mu-1)}$ ones followed by $2^{k(\mu-1)}$ zeros followed by all ones is now in the LSR. An example of this pattern is shown in Figure 4.7. This will cause the output of the generalized tree to be the complement of the input of the CDSTIT closest to the output if all the CDSITs are loaded properly.

(xli) Apply code 000 for $[LI-\mu]+1$ time periods to broadside load the CDSTITs with s'. (So far the following lines have been tested: 16 s-a p; 9, 10, 13, and 14 s-a 1; and 11 and 12 s-a 0. If line 8 is s-a 1 then a fault will be detected in step (xlii) as 0's will be shifted into both the patterns s and s'. If line 12 is s-a 1 then the broadside load can not occur and a fault will be detected. This means that line 12 can not be s-

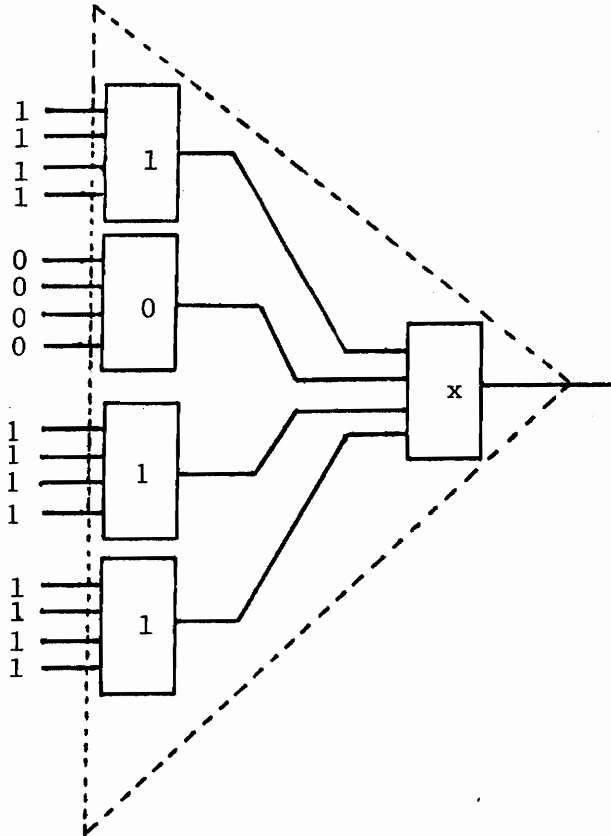


Figure 4.7: Leaf Pattern and Generalized Tree Initialization needed to Test the CDSTIT Closest to the input of the Generalized Tree (μ=2 and k=2 in the Generalized Tree).

a p without detection.)

(xlii) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is being entered into the ISRITs. Note the outputs.

(xliii) If the output string is not s' as accessed by T, then a fault has been detected. Stop.

(xliv) Code 111 should be applied for 1 time period so as to enter s. (If line 5 is s-a 0 then a broadside load can't occur unless line 7 is s-a 0 or line 9 is s-a 0.)

(xlv) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is being entered. Note the outputs. (If line 8 is s-a 1 a fault will be detected in step (xlvi as 1's will be shifted into the pattern s. If line 6 is s-a 1 then 0's are shifted into the register and a fault is detected in step (xlvi).)

(xlvi) If the output string is not s as accessed by T then a fault has been detected. Stop.

(xlvii) Code 101 should be applied for 2^{LI} time periods while C is being entered into the LSRIT closest to the output. (If line 9 is s-a 0 then the broadside load is not disabled and C will not be entered. If line 8 is s-a 0 the C will not be shifted into the LSRIT and an error will be

detected in step (xlviii). If lines 4 or 6 are s-a 0 or line 5 is s-a 1 the C can not be shifted in and a fault will be detected in step (xlviii). At this point it is known that line 8 can't be s-a p without detection.)

(xlviii) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is entered into the CDSTIT closest to the output [0's should be entered in the others]. Note the outputs after LI-1 time periods. (If lead 10 is s-a 0 then the broadside load is not disabled and C will not be contained in the LSRITs an error will be detected in step (xlix). As neither line 9 nor 10 can be s-a p without detection then line 11 can't be s-a p without detection. Additionally s-a p fault detection on lines 8 and 9 implies that line 7 s-a p will be detected.) [All that remains to be checked in so far as combinational logic is concerned is line 4 s-a 1 and lines 13 and 14 s-a 0.]

(xlix) If the output string of the generalized tree is not C as accessed by T then there is a fault. Stop.

(l) Code 101 should be applied for 2^{LI} time periods while $\bar{C}$ is being entered. [This assumes that $\bar{C}$

is produced by the same tree which orginally produced C.]

(li) Code 100 should be applied for 2^{LI}+LI-1 time periods while T is being entered. Note the outputs.

(lii) If the output string of the generalized tree is not $\overline{C}$ as accessed by T then there is a fault. Stop.

(liii) Code 000 should be applied for 2 time periods while 1's are entered into the LSRLSR. (If line 4 is s-a 1 then the broadside load of the LSRIT will be shifted two positions and a 1 will be placed on leaf 0.)

(liv) Code 100 should be applied for LI time periods while 0's are entered into every ISRIT. (The last output should reflect a value of 0 on leaf 0. If line 4 is s-a 1 then the value on leaf 0 will be 1 and the generalized tree will output a 0 instead of a 1. This means lead 4 can not be s-a p without a fault being detected.)

(lv) Apply code 100 for LI-2 time periods while the pattern composed of 1 followed by all 0's is entered into the ISRIT closest to the output. The other ISRITs should have 0's applied. This accesses a constant 1.

(lvi) Apply code 100 for LL time periods while a string consisting of two 0's and one 1 followed by 0's is entered into the ISRLSR. The output of the CDSTIT should be 1. (If line 13 is s-a 0 then a shift will occur in the ISRIT and a 0 will be output instead of a 1. A fault will be detected when the generalized tree outputs a 1 instead of a 0.)

(lvii) Code 101 should be applied for 2 time periods. The first time period enters a 1 into the LSRIT. The second time period is used to transmit the information from the output flip-flop of the generalized tree. (If line 14 is s-a 0 then a shift will occur in the ISRIT and a 0 will be output by the CDSTIT instead of a 1. This means that the generalized tree will output a 1 instead of a 0 and the fault will be detected.)

(lviii) Stop. If no stuck-at faults were encountered then the structure is fault free.

With the completion of this testing experiment all portions of the modified tree structure have been tested for all possible single and multiple stuck-at faults. This means that this new structure can be used to construct a machine capable of complete self testing given enough sets

of trees. It has been assumed that the D flip-flops would have s-a faults only at the inputs and outputs. If this assumption is not strong enough then either pattern c or C (depending on the register) must be reloaded into the various registers and the test must be extended to include these additional testing patterns instead of just patterns $\bar{c}$ and c or C and $\bar{C}$. This modification is rather trivial and will not be included. The "control trees" have not explicitly been tested in this chapter. The reason for this is that these trees are basically no different than any others. The only difference is that the testing sequences should be applied to all the control trees of a given block whenever one of them is to be tested. It should also be noted that the test would be controlled by a set of counters which would switch the appropriate functions on at the proper times. These counters have not been explicitly been included, however each of the steps of the algorithm does state the length of time required to execute it. The actual depths and the interconnection pattern of these counters provides a topic for future study. The only portion of the testing experiment yet to be substantiated is the initialization problem. This topic will be the subject of Chapter 5.

5. Initialization

After loading the LSR of a sequential tree, before it can begin to do useful work, it is necessary that the tree be initialized. There are basically two means of approaching this problem. The first is to extend the leaf shift register so as to include the internal D Flip-Flops. The second method is to enter a special initialization pattern into the LSR and apply a string of K-ary numbers of length μ to the input of the tree. This will initialize the internal D Flip-Flops of the tree. Each method has advantages under certain circumstances. The two methods are however unfortunately not totally independent of one another. If the first approach is taken the user has a choice between using the shift register for initialization and initializing using a subset of the second method (the second may not always be possible). If the second method of initialization is chosen the first may not be used even in part. Both methods will be examined in this chapter and data will be given to help the designer choose between these two alternative methods of initialization.

It is obvious that the first method of initialization will require certain modifications to the structure of the tree. The modification to the D Flip-Flops internal to the

tree can be seen in Figure 5.1. There are two ways in which the shift register can be designed to connect the Flip-Flops internally in the tree structure. These methods are shown in Figures 5.2 and 5.3. The structure of Figure 5.2 will be used in the rest of this discussion. From Figure 5.1 it can be seen that the D Flip-Flop internal to the tree will act as one stage of a shift register only when the signal which enables the LSR is high, In other words, when the LSR is shifting also. At all other times the output will be the input from the multiplexer attached to the input (Figure 1.4).

When initialization is to be performed using this approach, all that is necessary is that the initialization sequence be concatenated onto the front of the leaf pattern. When this combined pattern is shifted in, the initialization will have been performed and the tree will be ready to begin work on the following clock cycle. This structure for the tree requires that the number of Flip-Flops internal to the tree be given by the equation below:

$$\sum_{i=0}^{\mu-1}(2^k)^i \ .$$

From combinatorics this can be shown to be equal to

$$(2^k)^\mu - 1/(2^k-1)$$

This means that the entire initialization sequence will be of the length given above plus $2^{k\mu}$.

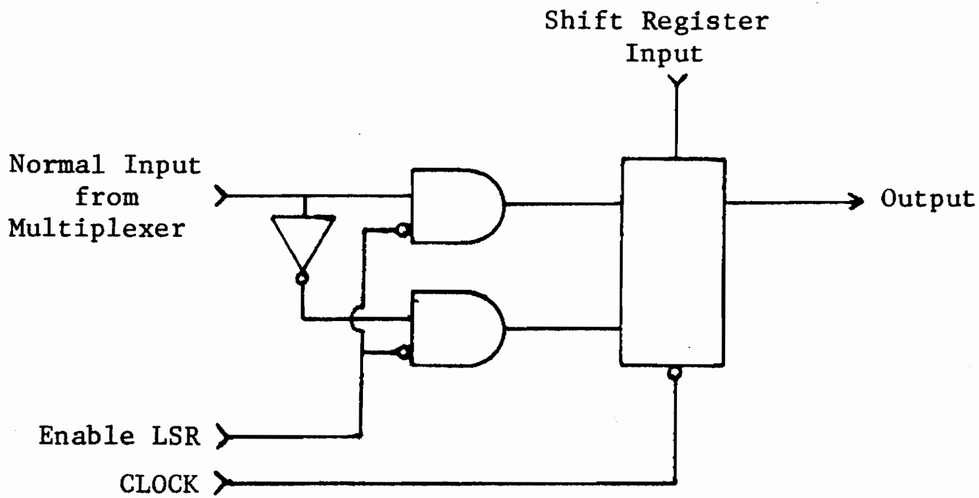


Figure 5.1(a): Structure surrounding D flip-flop in a tree with an Internal Shift Register for Initialization.

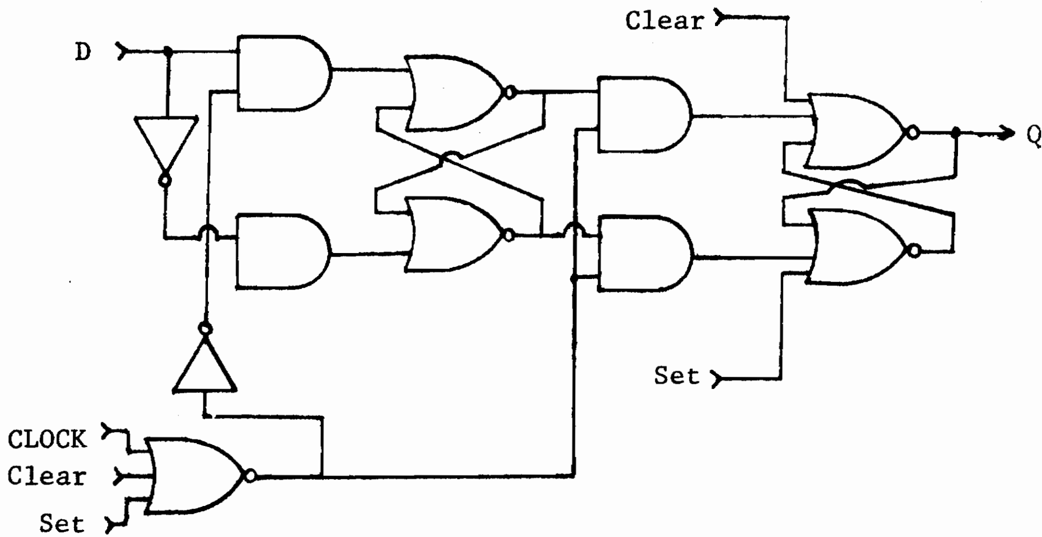


Figure 5.1(b): Internal structure of the flip-flop in 5.1(a).

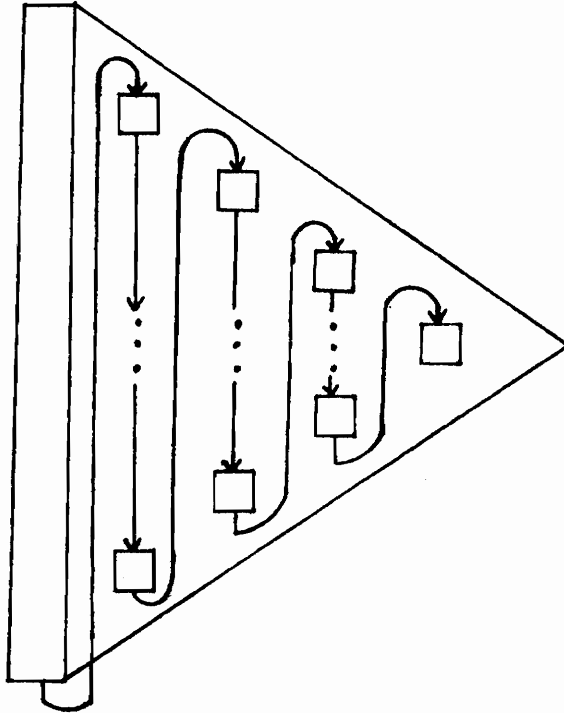


Figure 5.2: First Possible Shift Register Path through the Internal Flip-Flops of the tree.

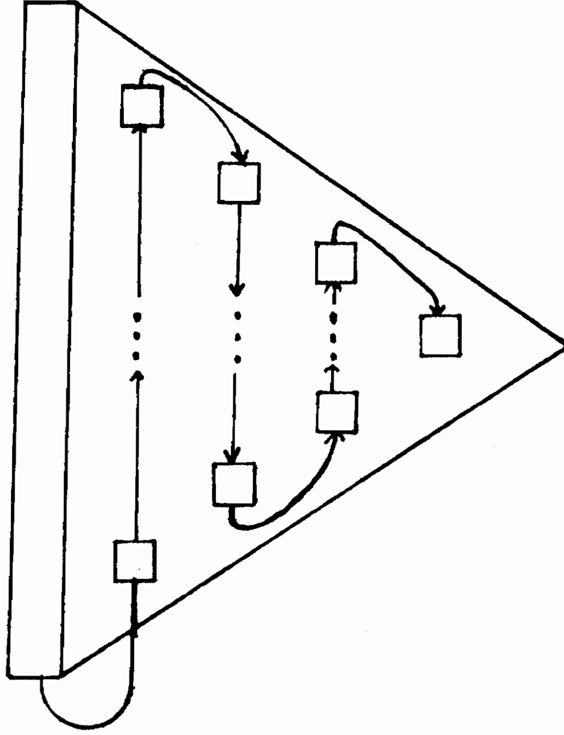


Figure 5.3: Second possible Alternative for Shift Register Path Through the internal Flip-Flops of the tree.

At first glance it would appear that this structure would require that an extra pattern be shifted through the LSR and the internal register of the tree for testing purposes. Actually this is not the case; all that is necessary is that the output of the tree be examined at time periods $(2^{nk}-1)/(2^k-1)$ and $(2^{nk}-1)/(2^k-1)+1$ when the sequence c is being shifted out of the LSR. At these times leaves $2^{nk}-1$ and $2^{nk}-2$ should appear at the output of the tree. This test is sufficient as these two bits of c have different values. If these bits are not correct then there is a s-a-f with in the internal register and the data path is faulty. In this case, this method of initialization will not work and the tree will have to be initialized from the leaves (if this is possible). In all cases the initialization using the first method will take $2^{nk}+(2^{nk}-1)/(2^k-1)$ time periods or more.

The second method of initialization has an advantage over the first in that there is no extra data path in which a fault can occur. Unfortunately, usually the complete initialization will take far longer than with the first method. If the initialization pattern in the tree is such that the normal operation of the tree can cause this pattern to occur within the tree then this method is much faster than the shift register initialization method. If the pattern within the tree is such that it can be initialized

from the leaves then only μ time steps are required to initialize the tree (not including the initialization of the LSR). If the state diagram of the machine at any time can return to the initial state of the machine then the tree realizing this machine can then be initialized from the leaves using the computational control pattern. However if the operational pattern on the leaves can not initialize the tree then $2^{\mu k}+1+\mu$ time steps are required for initialization. In this mode of initialization, an initialization pattern is first shifted into the LSR and the tree is initialized. This takes $2^{\mu k}+\mu$ time periods. Then the operational leaf pattern is shifted into the LSR and the tree is ready to operate. The latter takes another $2^{\mu k}$ time periods. It is unfortunate that the latter method of initialization can not be used if an internal shift register exists within the tree. Example 5.1 shows the manner in which a 1 input (k=1) generalized tree of depth 3 (μ=3) could be initialized with any set of constants.

<u>Example 5.1</u>:

As this is a one input tree label the leaf sequence $a_0b_0a_1b_1a_2b_2a_3b_3$ and place it on the tree as shown in Figure 5.4. Assume that the circled constants with in the tree structure are those that should be used to initialize the tree (as these constants are totally arbitrary this is

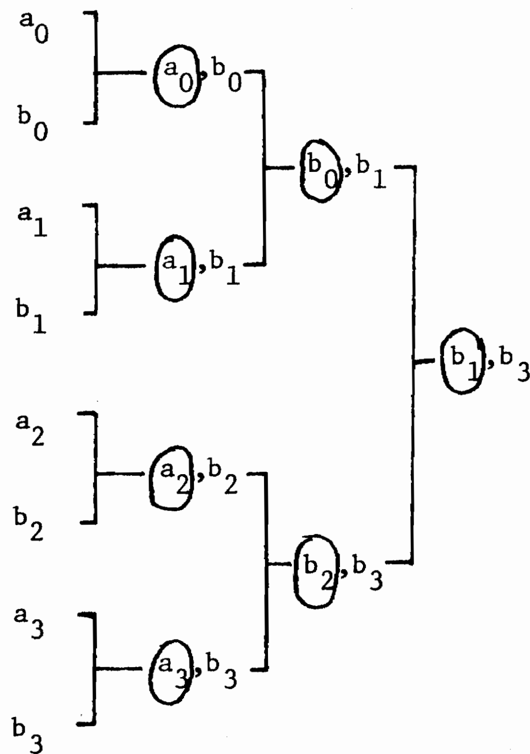


Figure 5.4; Data flow through the tree not including already used (circled) states used as a means of determining which which constants are available for a unique initialization.

certainly possible). Figure 5.5 shows the method in which the tree can be initialized. It begins with the input of a 0 followed by another 0 followed by a third 0 followed by a 1. After each new input is entered the new internal tree pattern is displayed. If k>1, the initialization can be carried out because the same input pattern may be used. It may also be done if $\mu>3$ as extra leaf variables are provided.

Theorem 5.1:

Any sequential tree of depth μ can be uniquely initialized from the leaves in μ time periods using the sequence consisting of $\mu-1$ K-ary ones followed by one K-ary 0.

Proof:

The proof will be inductive.

Basis: Let $\mu=1$.

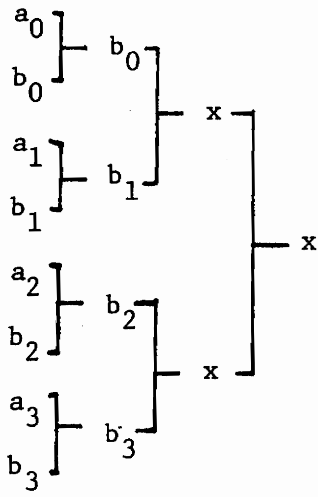
Let the first two leaves of the tree be constants a and b.

Apply the K-ary sequence consisting of no 1's followed by one zero as required by the theorem.

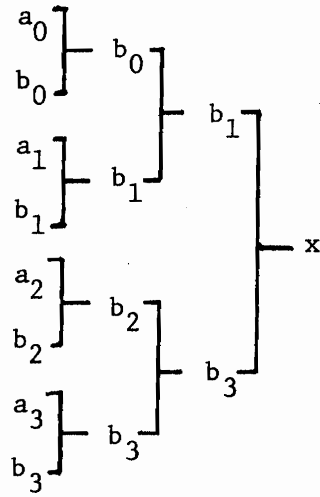
The tree is now initialized with a and the initialization is unique.

Induction Hypothesis: Assume that the sequential tree of

Input Sequence: 1



Input Sequence: 11



Input Sequence: 110

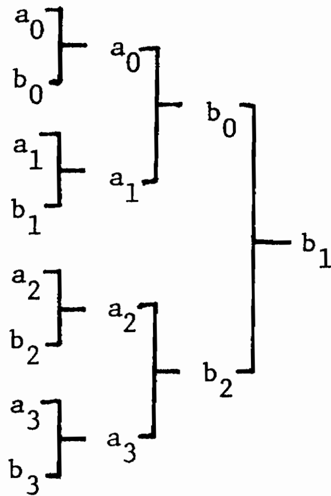


Figure 5.5: Method used to Initialize the Generalized Tree with Depth = 3 and 1 input from the leaves.

depth i can be initialized from the leaves in i time periods by entering the sequence consisting of $\mu-1$ K-ary ones followed by one K-ary 0.

Let the depth of the sequential tree to be initialized be i+1 (i.e. $\mu=i+1$).

Examine each of the subtrees of Figure 5.6 of depth i at time $\mu-1$ (i).

As only (K-ary) 1's have been applied so far, the output of each subtree of depth i will be that of the constant associated with leaf 1 of the subtree. When the zero is applied the unique constant b, at the output of subtree 0, is moved to the output of the tree.

By the inductive hypothesis each subtree of depth i is uniquely initialized.

Therefore all that remains to be shown is that the leaf constant associated with the output of the tree is not used in the initialization of subtree 0. Prasad and Gray [9] have shown that there exixts a unique path from any given leaf of a tree to the output of the tree. This is the data path within the subtree 0 consisting of all K-ary ones. As the last input to the tree is a 0, no b's can be used in the initialization of subtree 0.

Therefore all the variables associated with the internal flip-flops of a sequential tree are unique when the initialization is accomplished using the method in this

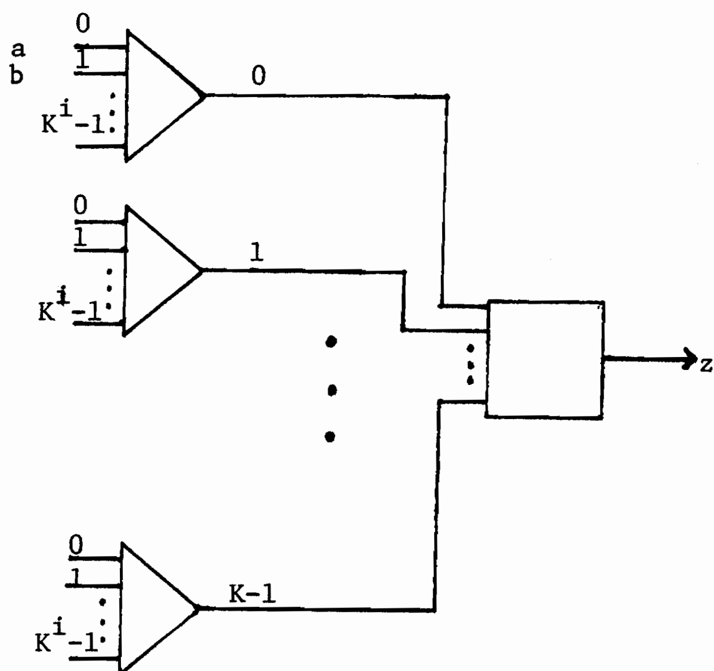


Figure 5.6: Sequential Tree of Depth μ Containing K subtrees of Depth μ-1.

theorem.

When the two methods of initialization are compared it is found that the shift register initialization will take at most 2 times as long as the leaf initialization (this occurs at $\mu=1,k=1$) but that as μ and k increase this factor approaches 1. This assumes that the leaf initialization is done with the same leaf pattern that will be used in computation. Both of these methods may be used on a tree with an internal shift register. If a comparison is made between shift register initialization and that which uses an additional "special" leaf initialization pattern then it is found that the ratio is 1:1 with $\mu=1$ and k=1 and decreases to 2:1 in favor of the shift register initialization as μ and/or k increase (the latter decrease is highly dependent upon the increasing value of the number of inputs, k).

The above results would suggest that the internal shift register be included in the tree structure as the tree may be initialized from either the shift register of the leaves depending upon the initialization states internal to the tree. It is known that the tree structure can be full tested and and can be initialized after testing (and during testing) to the required "state" and that this initialization can be completed in less then $2^{\mu k}+(2^{\mu k}-1)/(2^{k}-1)$ time periods. With this knowledge the testing and initialization algorithm is now complete.

6. Conclusions and Future Directions

It has been shown that the modified tree structure proposed in this work can be tested and that such a structure provides more operational flexibility than the standard generalized modular tree structure. The testing algorithm, although rather long, is not extremely complex. In addition to the implementation of the testing algorithm developed by Prasad and Gray [9] which could be used in testing the following trees; the generalized tree, the CDSTITs, and the CDSTLSR; an algorithm was developed to test the combinational logic associated with the modified tree structure.

It was then proposed that a set of control trees be added to a computational group which would control the combinational logic of a group of trees. With the proper relationship between computational groups, it was determined that these trees were in reality no different than any of the other trees of a computational group except in their normal function. A set of codes was developed which could be used to control a computational group and a set of control trees. The codes 000 and 111 were given as being no-operation codes for testing purposes. This suggests that all the control tree be tested together so as to always produce one of these two codes and thereby not affect the

other computational group. It was also suggested that multiple trees could be tested simultaneously and that the length of time required for testing could therefore be reduced significantly.

Future areas of research might include the following set of questions. Exactly where does the testing signal come from? Where do the inputs which control the control trees come from and exactly how many such signals are necessary? How many counters are necessary for controlling the testing sequence and how should they be arranged to most efficiently perform their control function? How many trees should be placed in a computation group or block? What inputs should be attached to the leaves of the combinational data steering trees? How will all these inputs be set up to provide the patterns s and s'? Where will the inputs to the registers for the combinational data steering trees come from? Can some of the inputs to the registers of the combinational data steering trees be combined so as to reduce the number of inputs to any one computational group? In the light of the above questions, should the testing algorithm be modified and if so, how? Would it be helpful if the trees could in some way be stopped during a calculation for example by disabling the clock to the tree? If stopping the trees would be helpful on what codes should this be done? The questions above are some of those which

must be answered before the testing algorithm can actually be simulated.

BIBLIOGRAPHY

[1]. A. D. Friedman, "Feedback on Synchronous Sequential Switching Circuits", IEEE Transactions on Electronic Computers, Vol. EC-15, pp 354-367, June 1966.

[2]. T. F. Arnold, C. J. Tan, and M. M. Newborn, "Iteratively Realized Sequential Circuits", IEEE Transactions on Computers, Vol. C-19, pp. 54-66, January 1970.

[3]. S. S. Yau and C. K. Tang, "Universal Logic Modules and Their Applications", IEEE Transactions on Computers, Vol. C-19, pp. 141-149, February 1970.

[4]. T. F. Arnold, Universal Structures with Uniform Interconnection Patterns for Synchronous Sequential Circuits, Columbia University, Doctoral Dissertation, 1969.

[5]. F. G. Gray and R. A. Thompson, "Reconfiguration for Repair in a Class of Universal Logic Modules", IEEE Transactions on Computers, Vol. C-23, pp. 1185-1194, November 1974.

[6]. C. Cioffi and E. Fiorillo, "Diagnosis and Utilization of Faulty Universal Tree Circuits", AFIPS Conference Proceedings 1969 Spring Joint Computer Confererce, pp. 139-147.

[7]. Z. Kohavi and I. Berger, "Fault Diagnosis in Combinational Tree Networks", IEEE Transactions on Computers, Vol. C-24, pp. 1161-1166, December 1975.

[8]. S. Ramo and A. R. Smith III, "Fault Detection in Uniform Modular Realization of Sequential Machines", Digest 1972 International Symposium on Fault Tolerant Computing, Newton, Massachusetts, June 1972, pp. 114-119.

[9]. B. A. Prasad and F. G. Gray, Research Initiation: A Theoretical Investigation of Diagnosable Logic Systems, Final Report, National Science Foundation, Washington D.C., October 1973, (Grant GJ-32718), NTIS Accession Number PB 230-441 AS.

[10]. S. W. Golumb, Shift-Register Sequences, Holden-day, San Francisco, 1967.

[11]. M. A. Breuer and A. D. Friedman, Diagnosis and Reliable Design of Digital Systems, Computer Science Press Inc., pp. 57-66, 1976.

VITA

Mark Hoptiak was born on February 26,1954 in Washington D.C. He received his Bachelor of Science degree in Electrical Engineering from Virginia Polytechnic Institute and State University in June of 1976 and expects to receive his Master of Science in Electrical Engineering from VPI&SU in June of 1977.

He is a member of the Electrical Engineering honorary society Eta Kappa Nu. He also belongs to the Institute of Electrical and Electronics Engineers (IEEE) and to the Association for Computing Machinery (ACM).

Mark Hoptiak

IMPLEMENTATION OF A TESTING AND INITIALIZATION ALGORITHM FOR A GENERALIZED TREE STRUCTURE

by

M. Hoptiak

(ABSTRACT)

This work considers a means of implementing a detection algorithm for multiple stuck-at faults occuring within the structure of a generalized tree using, as the testing machines, a set of generalized sequential trees. It is shown that the testing machine can be implemented on a set of trees of the same depth as that of the tree which is being tested. Modifications are made to the generalized tree structure so as to facilitate the testing algorithm and in order to simplify initialization. The testing algorithm is modified so as to test this modified structure. In addition, after testing is complete, two means of reinitialization of the sequential trees are discussed.