EXPLORING LINEAR SPEEDUP IN PARALLEL ATPG THROUGH SPECIAL TOPOLOGY

ZHIMIN SHI

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Exploring Linear Speedup in Parallel ATPG Through Special Topology

By

Zhimin Shi, BSc.

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science

Department of Computer Science

Memorial University of Newfoundland

March 12, 1994

St. John's Newfoundland Canada

ISBN 0-315-91620-6

Abstract

In digital system design, test pattern generation requires a considerable amount of computing time. Using a level-sensitive scan design, test pattern generation can be confined to the combinational circuits. It has been shown that the problem of test pattern generation for combinational circuits is NP-complete. Although many excellent algorithms have been developed to generate test patterns, they still do not keep pace with VLSI technology. Research is ongoing in the development of parallel processing techniques for test pattern generation, but there has been little research into what kind of topology has the greatest potential to speed up test pattern generation.

In this work, simulation software was developed for measurement of the speedup, and three topologies are proposed to explore the parallelism for automatic test pattern generation. These topologies are: modified complete binary tree (MCBTA), autonomous modified complete binary tree (AMCBTA), and square array structure (SQARRAY). The empirical results for these topologies show that a special topology has the potential capability to speed up test pattern generation and super-linear speedup can often result if an autonomous structure is adopted.

Acknowledgements

I wish to express my thanks to my supervisor Dr. Paul Gillard for his guidance, interest, constructive criticism and enthusiasm. Without his contribution, it would be impossible to give this thesis its current quality.

I would like to thank the systems support staff for providing help and assistance while I conducted this research.

I am also very grateful to the Administrative staff who have helped in one way or another in the preparation of this thesis.

In addition, I would like to acknowledge the financial support received from the Department of Computer Science and the School of Graduate Studies.

Special thanks are due to my fellow graduate students and good friends, and in particular to Zhengqi Lu, Xu He, Sun Yongmei, and Chen Hao for their valuable comments and useful suggestions. I would also like to thank Dr. Siwei Lu, Patricia Murphy and Elaine Boone for their help and assistance.

This thesis is dedicated to

my wife, Zhang Min

and my parents

for their support and encouragement throughout

the course of my education

Contents

List of Tables

List of Figures

PartI

Introduction

Chapter 1

Introduction

Generating test patterns for testing digital circuits is a very important aspect of VLSI design. It often consumes a significant portion of the design time. Owing to techniques such as the widely used level-sensitive scan design[5], the problem of test pattern generation is reduced to the problem of generating test patterns for combinational circuits. Even this problem has been shown to be NP complete[17].

There are two basic approaches to solve the automatic test pattern generation (ATPG) problem: algorithmic test pattern generation and statistical, or pseudo-random, test pattern generation. In the algorithmic approach, a specific ATPG algorithm is used to generate a test for each fault in the circuit. Most of these algorithms can be proved to be complete; that is, they are guaranteed to find a test for a fault — as long as a test exists. However, this may involve searching the entire solution space, which is computationally expensive.

Statistical test pattern generation, on the other hand, selects test patterns at random, or by using some heuristic, and uses fault simulation to determine the faults detected by the pattern. Test patterns are selected and added to the test set if they detect any previously undetected faults, until some required fault

coverage measure or computation time limit is reached. This method finds tests for the easy-to-detect faults quickly but becomes less and less efficient as the easy-to-detect faults are removed from the fault list and only the hard-to-detect faults remain. In many cases, the required fault coverage cannot be achieved without excessive computation times.

An efficient combined method for solving the ATPG problem uses statistical methods to find tests for the easy-to-detect faults on the fault list and switches to an algorithmic method to find tests for the remaining hard-to-detect faults. In either the combined or the purely algorithmic method, a significant portion of the computation time will be spent generating tests for the hard-to-detect faults algorithmically. Therefore, finding a method to speed up this process should reduce the overall computation time considerably.

Much research has gone into increasing the efficiency of algorithms for ATPG. However, the overall gains achieved through these improvements have not kept pace with increasing circuit size, and computation time is still excessive. Another approach to reducing computation time is simply to use a faster machine. Parallel-processing machines are becoming available for general use and are helping to solve other problems in computer-aided design.

Much research has been done to parallelize the test pattern generation problem. Most of this work concentrates on how to use existing multi-processor systems, such as the Intel iPSC/2, Network of Sun workstations, or the Links-1 Z8000 based systems, to effectively generate test patterns.

Parallel techniques for ATPG problem can be classified into five major categories[8]:

1. fault partitioning,

2. heuristic parallelization,

3. search-space partitioning,

4. functional (algorithmic) partitioning,

5. and topological partitioning.

Although some promising results have been shown, much work still remains.

As the development of microelectronics technology progresses, massively powerful processors will be used to form special parallel architectures to generate test patterns. New architectures for the interconnection of processors have to be studied so as to design a very efficient multi-processor system for ATPG. Therefore, it is natural to investigate a good interconnection network to speed up the ATPG process, when many processors are available. This thesis discusses this problem by proposing several special parallel architectures, and examining the empirical results through simulation. These special architectures are the modified complete binary tree (MCBTA), the autonomous modified complete binary tree (AMCBTA), and the square array architecture. With these special architectures, the parallel algorithms discussed in this work were found to achieve linear, and sometimes superlinear speedup. The empirical results also show that AMCBTA is the best one among these special architectures.

This report is arranged as follows: Chapter 2 and chapter 3 give a survey of automatic test pattern generation. Chapter 4 shows our experimental results about the faults. Chapter 5 discusses the model for measuring the performance

of a parallel automatic test pattern generation system. Chapter 6 describes all algorithms used in our simulation software to simulate parallel automatic test pattern generation systems and evaluate their performance. Chapter 7 briefly discusses 4 connected structures and isomorphism of two graphs. Chapter 8 and 9 demonstrate our three architectures designed to solve test pattern generation problem in parallel. In the conclusion, the results are summarized and some future work are discussed.

Chapter 2

Conventional ATPG Algorithms

In this chapter, we will review 3 widely used algorithms, the D-algorithm, the Podem algorithm, and the Fan algorithm. Before this review, the stuck-at model and the testing problem are described.

2.1 Stuck-at Fault Model & Testing Problem

This section first introduces the stuck-at fault model followed by a discussion of the testing problem.

2.1.1 Stuck-at Fault Model

Logic gates are realized by transistors, normally either bipolar transistors or metal oxide semiconductor field-effect transistors (MOSFET, or simply MOS). The technology families based on bipolar transistors are transistor-transistor logic (TTL), emitter-coupled logic (ECL), and so forth. Some logic families based on MOSFET are p-channel MOSFET (p-MOS), n-channel MOSFET (n-MOS), and complementary MOSFET (CMOS). Although ECL and TTL are important for high-speed applications, their integration sizes are limited by the heat generated by their heavy power consumption and by large gate sizes. In contrast, the MOS logic

families are well suited for LSI or VLSI, because higher integration can be obtained than with bipolar logic families. Most LSI and VLSI circuits of today are implemented with MOS.

A *fault* in a circuit is a model at the logic level of the effect of a physical defect of one or more of its components. Faults can be classified as logical or parametric. A *logical fault* is a defect that causes the logic function of a circuit element or an input signal to be changed to some other logic function; a *parametric fault* alters the magnitude of a circuit parameter, causing a change in some factor such as circuit speed, current, or voltage.

Circuit malfunctions associated with timing are due mainly to circuit delays. Those faults that relate to circuit delays such as slow gates are called *delay faults*. Usually, delay faults only affect the timing operation of the circuit, which may cause hazards or critical races.

Faults that are present in some intervals of time and absent in others are *intermittent faults*. Faults that are always present and do not appear, disappear, or change their nature during testing are called *permanent faults* or *solid faults*. Although many intermittent faults eventually become solid, the early detection of intermittent faults is very important to the reliable operation of a circuit. However, there are no reliable means of detecting their occurrence, since such a fault may disappear when test is applied. In this thesis, we will consider mainly logical and solid faults.

When an input or output of a logic gate is always a fixed voltage, either high or low, it is said to have a *stuck-at fault*. For positive logic, if a node is low, it is said to be *stuck-at 0*; when it is always a high voltage, it is said to be *stuck-at 1*.

The most popular fault model used in gate level simulation is the stuck-at fault. The stuck-at-fault model was originally used as a means of describing faults in early electromagnetic relay computers. However, the model was also found to be applicable to diode transistor logic (DTL); this led to its use in small scale integration (SSI) and medium scale integration (MSI) fault modeling. Thus the model became a standard widely used in the integrated circuit industry[31]. When Roth [32] developed the D-algorithm in 1966, to automatically generate test sets based on the stuck-at-fault model, its continuation was assured. However, as failure modes in modern VLSI circuits are better understood, its applicability and usefulness are being challenged [34]. The CMOS stuck open fault model is an example.

2.1.2 Testing Problems

To ensure the proper operation of a system, we must be able to detect a fault when one has occurred and to locate it or isolate it to a specific component — preferably an easily replaceable one. The former procedure is called *fault detection*, and the latter is called *fault location*, *fault isolation*, or *fault diagnosis*. These tasks are accomplished with tests. A *test* is a procedure to detect and/or locate faults. Tests are categorized as fault-detection tests or fault diagnostic tests. A fault-detection test tells only whether a circuit is faulty or fault-free; it tells nothing about the identity of a fault if one is present. A fault diagnostic test provides the location and the type of a fault and other information. The quantity of information provided is called the *diagnostic resolution* of the test; a fault-detection test is a fault diagnostic test of zero diagnostic resolution. If a test not only detects a fault

but also locates the fault, it is a fault diagnostic test of high diagnostic resolution.

Logic circuits are tested by applying a sequence of input patterns that produce erroneous responses when faults are present and then comparing the responses with the correct (expected) ones. Such an input pattern used in testing is called a *test pattern*. In general, a test for a logic circuit consists of many test patterns. They are referred to as a *test set* or *test sequence*. The latter term, which means a series of test patterns, is used if the test patterns must be applied in a specific order. Test patterns, together with the output responses, are sometimes called *test data*.

If there exists only one fault in a circuit, it is said to exhibit a *single fault*. If there exist two or more faults at the same time, then the circuit exhibits *multiple faults*. Here, we are only concerned with a single fault in a circuit. For a circuit with k lines, there are at most $2k$ possible single stuck-at faults since each line has at most 2 possible faults: stuck-at-0 and stuck-at-1.

The testing of logic circuits is performed in two main stages: generating test patterns for a circuit under test (the *test generation* stage) and applying the test patterns to the circuit (the *test application* stage). Thus, the generation of test patterns is important; however, it is very difficult for large circuits, so most of the effort of the past 20 years in this field went into research and development of efficient and economical test generation procedures.

The quality of a test (a set or a sequence of test patterns) depends much on the fault coverage as well as the size or length of the test. The *fault coverage* (or *test coverage*) of a test is the fraction of faults that can be detected or located within the circuit under test. The fault coverage of a given test is determined by

a process called *fault simulation*, in which every given test pattern is applied to a fault-free circuit and to each of the given faulty circuits, each circuit behavior is simulated, and each circuit response is analyzed to find what faults are detected by the test pattern. Fault simulation is also used to produce *fault dictionaries*, in which the information needed to identify a faulty element or component is gathered.

2.2 D-algorithm

The first algorithm for ATPG that was proved complete is the D-algorithm introduced by Roth in 1966[4]. The D-algorithm includes a notation and a calculus with which a single stuck-at fault can be detected at a node in the circuit and propagated to a primary output of the circuit. This algorithm uses a five-valued logic, which consists of the logic value 0 and 1, an unknown value X, and two additional values D and $\overline{D}$. A D value signifies a value of 1 in the good circuit and 0 in the faulty circuit, and a $\overline{D}$ value represents a value of 0 in the good circuit and 1 in the faulty circuit.

Each gate in the circuit has two D-cubes associated with it, the *primitive D-cube* of a fault (pdcf) and a *propagation D-cube* (pdc). A pdcf is the set of inputs that produces an error signal on the output of that gate if it contains any fault of the particular type. A pdc specifies the input values necessary to propagate an error signal on an input of a gate to the output.

The D-algorithm's basic operation is the repeated intersection of the D-cubes necessary to perform the tasks required to test for a specific fault. These tasks consist of three processes: fault sensitization, fault propagation, and justification.

Fault sensitization is the process by which the circuit node presumeed to exhibit the fault is made to produce an erroneous value as a result of the fault. Sensitization is accomplished by specifying an input combination for the circuit element containing the fault, using the pdcf's, such that the node presumed to exhibit the fault holds the complement of the fault value.

The list of circuit elements closest to the primary outputs that have a D or $\overline{D}$ on the output is called the *D frontier*. The objective of fault propagation is to advance the D frontier to the primary outputs. This process sensitizes all possible paths from the fault site to the primary outputs. This multiple-path sensitization is necessary for the D-algorithm to guarantee completeness.

During fault sensitization and fault propagation, certain circuit nodes are required to take on specific values. Establishing this value, or goal, on the node by placing values on the primary inputs is called *justification*. The primary inputs that can be used to justify a goal are usually determined by backtracking through the circuit topology from the node in question to the primary inputs. A value is chosen for one of these inputs, and a forward simulation-like process, called forward implication, is performed to see if this input assignment is consistent with satisfying the goal. If it is not, a different value is chosen and the process is repeated. A test is finally generated when the fault sensitized, a path for the fault to be observed at the primary outputs in sensitized, and all of the goals are justified.

As an example of the D-algorithm, consider the circuit under test shown in Figure 2.1. Assume that a test is being generated for a stuck-at 1 fault on node J. The first step is to fill in an initial test cube with a $\overline{D}$ on node J, as shown in test

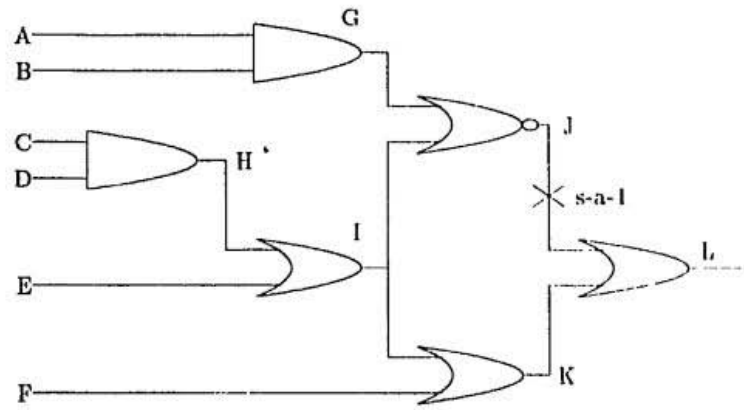


Figure 2.1: A Circuit for D-algorithm

Test Cube	A	B	C	D	E	F	G	H	I	J	K	L
0										$\overline{D}$		
1							1		X	$\overline{D}$		
2	1	1					1		X	$\overline{D}$		
3	1	1				0	1		0	$\overline{D}$	0	
4	1	1			0	0	1	0	0	$\overline{D}$	0	$\overline{D}$
5	1	1	0	X	0	0	1	0	0	$\overline{D}$	0	$\overline{D}$

Table 2.1: Test Cubes for Justification First

cube 0 in Table 2.1. This value is then sensitized using a pdcf for the NOR gate (test cube 1). Next, all values implied on other circuit nodes by the previous step are filled in (test cube 2). The next step is to advance the D frontier by setting node K to 0. This implies values on nodes I and F (test cube 3). The 0 value on node I in turn implies 0 values on nodes E and H (test cube 4). The final step is to justify the 0 value on node H by setting input C to a 0 value (test cube 5).

If the values shown in test cube 1b, Table 2.2, were chosen when selecting the

Test Cube	A	B	C	D	E	F	G	H	I	J	K	L
0b										$\overline{D}$		
1b							X		1	$\overline{D}$		
2b							X		1	$\overline{D}$	1	1

Table 2.2: Test Cubes for Propagation First

pdcf for the initial fault, the implications of that choice would have caused a test to be impossible, as test cube 2b shows. This problem would have caused the algorithm to backtrack to the last point a choice was made, pick the alternate choice, and proceed from there. In the D-algorithm, choices are available at many internal nodes in the circuit, and more than two choices can be present if there are gates in the circuit with more than two inputs. This fact greatly increases the size of the algorithm's search space and makes backtracking more complex. The D-algorithm can be implemented as recursive routine that pushes or pops test cubes off a test cube stack as required for forward progress or backtracking.

Note that justification of two separate node assignments cannot be undertaken simultaneously, because if an inconsistency occurs, it will not be possible to determine which unique assignment caused it. Also, the original D-algorithm does not specify which process — fault sensitization or fault propagation — is to be undertaken first or whether justification is to be done in intermediate steps or deferred until the process ends. These details are left to the implementation. Unfortunately, the efficiency with which a test can be generated for a specific fault depends heavily on the order of these operations, and the most efficient order is determined by the circuit topology. For example, if in generating a test for ./

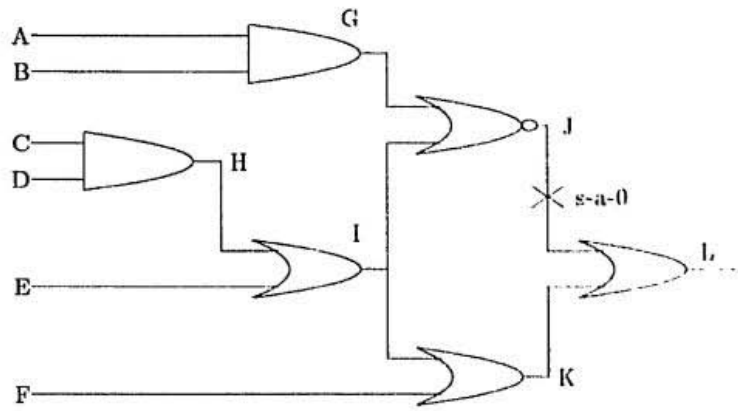


Figure 2.2: A Podem Example

stuck-at-1 in the circuit of Figure 2.1, fault propagation is done before selecting a pdcf, the unique value of 0 required on node *I* will be discovered and the pdcf for the faulty gate will be fixed. Test generation can then proceed without the possibility of backtracking. Finding the most efficient order for operations and detecting inconsistencies early in the process is the focus of most subsequently developed algorithms.

2.3 PODEM Algorithm

The Podem (Path-Oriented DEcision-Making) algorithm is an attempt to reduce the size of the solution space that must be searched. Recall that the D-algorithm tries to assign a value to each circuit node. Conflicts can arise when values assigned to different nodes cannot all be justified. Podem tries to eliminate these hidden conflicts by assigning values only to the primary inputs.

The algorithm begins by trying to justify the D or $\overline{D}$ at the node under test,

similar to the D-algorithm. This justification is done by assigning values to primary inputs that affect the node in question. These primary inputs are again found by backtracking through the circuit topology. When an input assignment is made, a simulation-like process, called forward implication, is run to find all of the node values implied by the assignment. If this new input assignment is incompatible with the goal, the complementary value is tried. If the complementary value assignment also conflicts, the algorithm backtracks efficiently to the previous input assignment. This process results in an orderly search methodology that will implicitly search the entire input space.

This search methodology can be represented by a binary search tree, as Figure 2.3 shows. After the value at the faulty node is justified, subsequent objectives are set up to propagate the D frontier along a path or paths to some primary output. The exact order in which this process occurs is again implementation dependent. The important point is that this strategy of assigning values only to primary inputs orders the search space. This procedure lets the search methodology prune the search tree implicitly and increase efficiency.

Consider, for example, the Figure 2.2, a representation of the binary search space for a J stuck-at-0 fault in the circuit under test. This search space was constructed using the simple heuristic of always first trying the logic value 1 on a primary input. Since assignment of the value 1 for node B in the left-hand subtree is inconsistent, all solutions that live below B in that part of the solution space can be pruned from the search tree. This ordering of the search space also allows it to be divided into disjoint sections so that work on the different sections can proceed simultaneously. Note that the processor must have access to the entire

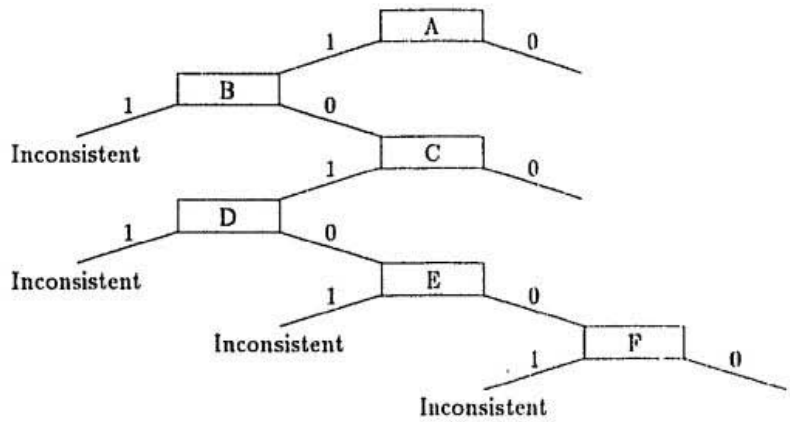


Figure 2.3: Podem Search Space Diagram

circuit topology and that only one goal may be justified at a time, as with the D-algorithm.

2.4 FAN Algorithm

The Fan algorithm is similar to Podem but includes improvements to increase its efficiency. The major goal of Fan is to reduce the number of backtracks in the search tree. This is accomplished using several techniques, including the consideration of fan-out branches in the circuit as a special case, hence the name Fan.

To examine this concept, we must define several terms. A *freeline* is a circuit node that has no predecessors that are part of a fan-out loop. As such, freelines may have a uniquely assigned value. In Figure 2.4, lines A through I are examples of freelines. A *bound line* is the opposite of a freeline. Nodes J and K are bound lines and cannot have unique (independent) values assigned to them. *Headlines*

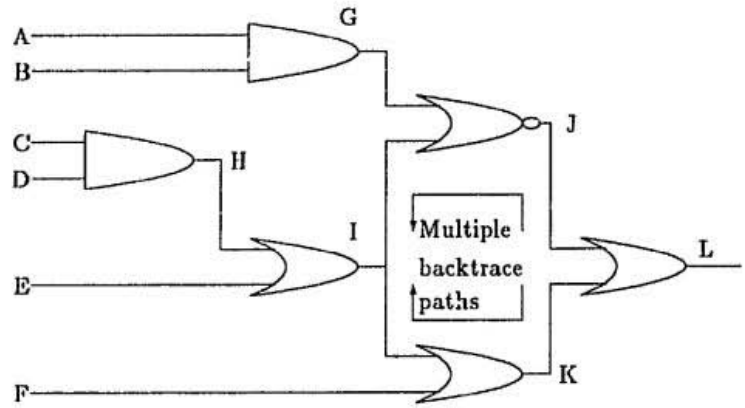


Figure 2.4: A Fan Example

are freelines that drive a gate that is part of a reconvergent fan-out loop. Node I in the figure is a headline. By definition, headlines can also be assigned values arbitrarily because they are freelines and can always be independently justified. They can therefore be treated as primary inputs in the justification process.

Identification of these nodes makes reconvergent fan-out loops much easier to handle. Once a test is found by treating headlines as primary input, the values on them can be justified at the end of the test generation process. Fan also uses a multiple-backtrace procedure for reconvergent fan-out branches buried in the circuit to reduce the number of backtracks that must be made in the search. For example, if a certain value is necessary at node L in the figure, and this circuit is part of some larger circuit, a single backtrace could be made along the path $L \rightarrow J \rightarrow G \rightarrow A, B$. Values for inputs A and B could be chosen so that the goal is satisfied with a unique value on nodes I and K. Then if the value on K cannot be achieved with the value chosen for I, a significant amount of backtracking in

the search tree can result. First, with a multiple-backtrace both the $L \rightarrow J \rightarrow I$ and $L \rightarrow K \rightarrow I$ paths can be used to determine the value needed at I to satisfy the goal. This value would then be set as a requirement for the justification of the value at node L. This process can increase the Fan algorithm's efficiency significantly in a circuit with numerous buried reconvergent fan-out loops.

Three conventional automatic test pattern generation algorithms were discussed. They can often be used to generate test patterns for very hard-to-detect faults. Parallelization is one of the methods used to speed up this procedure.

Chapter 3

Taxonomy of Parallel ATPG Algorithms

As mentioned earlier, techniques to parallelize ATPG can be classified into five major categories according to Robert's contribution in [8]: 1) fault partitioning; 2) heuristic parallelization; 3) search-space partitioning; 4) functional (algorithmic) partitioning; and 5) topological partitioning. Tables 3.1, 3.2, 3.3, 3.4 and 3.5 in the following section are taken from this reference. The following sections will give an overview for each category, and present its advantages and disadvantages, the type of parallel machine it has been implemented on, and a brief summary of the reported results.

3.1 Fault Partitioning

Fault partitioning is the simplest way to parallelize the ATPG problem. It first divides the fault set F into several subsets F_i, $i = 1, \cdots, m$, $F = \cup_{i=1}^{m} F_i$, and $F_i \cap F_j = \emptyset$ where $i \neq j$. Every processor P_i is assigned to generate test patterns to the fault set F_i. This scheme is called *static fault partitioning*.

Static fault partitioning results in each processor having a completely separate

Table 3.1: Summary of Fault Partitioning

Researchers	Parallel Machine	Scalability	Major Results
Srinivas Patil, Prith Banerjee	Intel iPSC/2	Nearly linear speedup for up to 8 processors; speedup falls off after that.	Demonstrated that ATPG with fault simulation is more efficient than ATPG alone, even in parallel environment. Fault partitioning shows good speedup for up to 8 processors.
Hideo Fujiwara, Tomoo Inoue	Network of Sun 3/50 workstations	Nearly linear speedup for up to 5 processors.	Verified analysis of optimal grain size for fault-partitioning system with experimental results.
Susheel J.Chandra, Janak H.Patel	Network of Sun 3/50 workstations	Uniform partitioning: nearly linear speedup for 5 processors.	Introduced concept of heuristic parallelization and developed two methods : uniform and concurrent heuristics. Demonstrated uniform partitioning produces better speedup.

task in that it performs the entire test generation procedure on its own. If the fault set is divided carefully, each processor will have roughly the same amount of work and will finish in about the same time. If this is the case, the communication cost is low. In practice, it is very difficult to get such a partition prior to executing the ATPG algorithms, so dynamic scheduling is used. In dynamic scheduling, each processor requests a new fault from a master scheduler when it is idle. Dynamic scheduling increases the communications overhead because of requests from idle processors for new faults.

In ATPG, one test pattern can test several faults at the same time. This implies that the time to generate test patterns for those other faults can be saved if a test pattern is found for one fault and at the same time the test pattern can detect those other faults. In static fault partitioning, if one test pattern for fault $f_i \in F_i$ is found and after fault simulation, $f_{i_1}, \cdots, f_{i_k}$ are found to be detectable by the same test pattern, $f_{i_1}, \cdots, f_{i_k}$ should be removed from F. If $f_{i_j} \in F_i$, it is easy to remove it since the processor itself can do without any communication with other processors. If $f_{i_j} \in F_l$ $(i \neq l)$, communication between P_i and P_l is necessary to remove f_{i_j} from F_l. This communication increases the parallel system's communication overhead and reduces the possible speedup.

Table 3.1 summaries the current research work using fault partitioning. The best scalability to date is 8 processors.

3.2 Heuristic Parallelization

As we know heuristics can be used to guide the algorithm to generate test patterns. Research has indicated that many heuristics will produce a test for a given fault

within some computation time limit when other heuristics have failed to do so[9]. We can use complementary heuristics to speed up the ATPG in multiprocessor systems.

Suppose there are k different heuristics. k processors are used to generate test patterns, and each processor uses a different heuristic. All the processors compute a test for the same fault. Once a processor succeeds in generating a test for the fault, it sends a message to other processors to notify them to stop working. Then all processors begin to work for a new fault.

Heuristic parallelization has the potential to achieve greater speedups than the uniform-partitioning method because of possible anomalies in the ordering of the heuristics for different faults. For example, suppose the time limit for each of five heuristic in the uniform-partitioning method is 10 seconds and only the last heuristic on the list can generate a test for a specific fault within the time limit, say in 5 seconds. Then the processing time for the uniform-partitioning method will be 45 seconds. However, the concurrent heuristic method will find a test for the same fault in only 5 seconds.

In the heuristic parallelization method, there is no way to ensure that the search space of each processor is disjoint. That is, even though the heuristics used by the processors differ, they might all lead the ATPG algorithm down similar paths to a non-solution and a test may not be found in the allotted time, even though one exists. This means that the heuristic techniques cannot be guaranteed to make all processors work efficiently together to find a test for a single hard-to-detect fault which takes a large amount of computation time.

Table 3.2 gives a summary of heuristic parallelization. It shows that the

Table 3.2: Summary of Heuristic Parallelization

Researchers	Parallel Machine	Scalability	Major Results
Susheel J.Chandra, Janak H.Patel	Network of Sun 3/50 workstations	Concurrent heuristics : less than linear speedup for only 5 processors	Introduced concept of heuristic parallelization and developed two methods : uniform and concurrent heuristics. Demonstrated uniform partitioning produces better speedup.

speedup is less than linear speedup, for up to at least 5 processors.

3.3 Search-space Partitioning

Search-space partitioning is a technique to make all processors work efficiently together to find a test pattern for a single fault.

The search space is divided into sub-search spaces. Given a circuit with N_{pi} primary inputs, there are $2^{N_{pi}}$ possible input patterns. The search space is the set which contains all these patterns. A sub-search space is a subset of the search space. A processor searches one of the sub-search spaces. The sub-search spaces for the processors are disjoint and are spread as far as possible across the solution space to maximize the area of the current search. This organization increases the chances of finding a valid solution quickly.

The following is an example which shows one way to partition the whole search space. Suppose there are 2^k processors, the number of primary inputs for a given circuit is N_{pi} ($N_{pi} \geq k$). Then the whole search space is $2^{N_{pi}}$. We can divide it into 2^k sub-search spaces if every processor has an identifier i ($0 \leq i < 2^k - 1$).

From N_{pi} inputs, k inputs are selected. These k inputs can work as processor identifiers, or identifiers of sub-search spaces, since 2^k different values can be used to represent 2^k processors. The whole search space is divided by these 2^k values. Without losing generality, the selected k bits are the first k bits in N_{pi} bits

$$a_0 \cdots a_{k-1} \underbrace{x \cdots x}_{N_{pi}-k}$$

where $a_i \in \{0,1\}$, $i = 0, \cdots, k-1$ and x is a unspecified value forming a sub-search space. This space is assigned to processor P_s, where $s = \sum_{i=0}^{k-1} a_i 2^i$. For each P_{s_1}, P_{s_2} $(s_1 \neq s_2)$, $s_1 = \sum_{i=0}^{k-1} a_{1_i} 2^i$, $s_2 = \sum_{i=0}^{k-1} a_{2_i} 2^i$, there is at least one i, $a_{1_i} \neq a_{2_i}$. Otherwise, we have

$$s_1 = \sum_{i=0}^{k-1} a_{1_i} 2^i = \sum_{i=0}^{k-1} a_{2_i} 2^i = s_2$$

This contradicts that $s_1 \neq s_2$. Therefore the sub-search spaces are disjoint.

It is impossible to search the whole space within limited time, for large problems, because the search space increases exponentially, so a backtrack limit still must be specified. When the number of backtracks exceeds the limit, the algorithms will give up the search and consider this fault as a hard-to-detect fault.

[13] makes the following observations: First, increasing the backtrack limit on the uniprocessor implementation does not yield better results on hard-to-detect faults, and the parallel algorithm yields better results for an equal number of backtracks. The results are better because the parallel algorithm searches a larger portion of the solution space. Second, the parallel algorithm runs much faster than the uniprocessor implementation and exhibits early linear speedup in most cases for up to 16 processors.

Table 3.3 shows the current research for search space partitioning. One result

Table 3.3: Summary of Search Space Partitioning

Researchers	Parallel Machine	Scalability	Major Results
Srinivas Patil, Prith Banerjee	Intel iPSC/2	Nearly linear speedup for up to 16 processors. Superlinear speedup in some cases.	Introduced efficient search space partitioning using Podem algorithm.
Akira Motohara, Kenji Nishimura, Hideo Fujiwara, Issao Shirakawa	Links-1 Z8000-based system	Averaged linear speedup for up to 50 processors during search space phase.	Demonstrated good speedup is possible for large numbers of processors using search space partitioning.

shows that linear speedup can be had for up to 16 processors. Another shows that linear speedup can be had for up to 50 processors. These results are much better than the results in previous subsections.

3.4 Functional (algorithmic) Partitioning

An algorithm can be divided into independent subtasks that can then be executed on separate processors in parallel. This method is referred to as functional partitioning.

Motohara[11] uses a type of functional partitioning to remove the easy-to-detect faults from the fault list. This procedure is done before the parallel method for hard-to-detect faults presented in the previous section is run. The method begins by dividing the fault list into groups of related faults. Typical related faults include those along the same path between a fault site and a primary output. After the fault list is divided into groups, each group is sent to a cluster of processors that

Table 3.4: Summary of Algorithmic Partitioning

Researchers	Parallel Machine	Scalability	Major Results
Srinivas Patil, Prith Banerjee	Intel iPSC/2	Nearly linear speedup for up to 8 processors; speedup falls off after that	Demonstrated that ATPG with fault simulation is more efficient than ATPG alone, even in parallel environment. Fault partitioning shows good speedup for up to 8 processors.
Akira Motohara, Kenji Nishimura, Hideo Fujiwara, Issao Shirakawa	Links-1 Z8000-based system	Linear speedup for up to 10 processors during algorithmic phase.	Introduced combination of algorithmic and search space partitioning systems.

includes a test generator and a fault simulator. The test generator takes the first fault and generates a test for it using a Podem algorithm with a limited number of backtracks. If a test for a fault is not generated within the backtrack limit, it is considered a hard-to-detect fault and is processed later. If a test is found, it is sent to a fault simulator node. This node runs a version of a concurrent fault simulator[33] to determine which other faults the test pattern detects. These faults are then removed from the fault list.

So far most serial ATPG algorithms developed are difficult to parallelize functionally. In order to efficiently use functional partitioning, a new algorithm for ATPG must be designed.

Table 3.4 shows that the current algorithmic partitioning systems can reach linear speedup for up to 10 processors.

Table 3.5: Summary of Topological Partitioning

Researchers	Parallel Machine	Scalability	Major Results
Fumiuasu Hirose, Koichiro Takayama, Nobuaki Kamato	Special purpose simulation processor	No results available speedup falls off after that	Demonstrated topological partitioning for simulation portion of ATPG process.
Glenn A. Kramer	Connection Machine	Linear speedup for circuits with up to 15 - 18 inputs. Speedup falls off rapidly after that.	Employs topological partitioning by mapping one circuit element to each Connection Machine processor. Only current algorithm demonstrated on massively parallel machine.

3.5 Topological Partitioning

All parallel algorithms discussed so far require each processor to access to the entire circuit database. This may be a problem for large circuits because each processor may not have enough memory to hold the entire circuit database. Topological partitioning tries to divide a circuit into separate partitions and instantiate each partition on a different processor. Each processor only processes a partition of circuit therefore less memory is needed. Since it is a difficult task to partition circuits so as to parallelize the ATPG algorithm, no ideal method has been reported so far. Further work is needed.

Table 3.5 shows the summary of current research systems using topological partitioning. It is clear that the results are not satisfactory.

So far, most of the techniques for parallel-processing ATPG use one of the commercially available networks to provide communication between processors. Tables 3.1 to 3.5 summarize the previous research work in parallel processing ATPG. For example, [14] and [13] used the Intel iPSC/2; a network of Sun3/50 workstations were used by [10] and [12]. As the number of processors increases, it is unavoidable that network communication load becomes heavier and heavier. As the limited capacity of the communication network is provided, traffic jams appear, computation time decreases and the network saturates. Therefore, in order to permit the computing time to decrease linearly, ATPG requires a communication network which can avoid communication conflicts.

Chapter 4

Hard-to-detect Faults

Since massively parallel machines with hundreds or thousands of processors will be available in the future, can these machines be used to efficiently solve the automatic test pattern generation problem? What is the problem which should be concentrated on? Some experiments may give us some hints.

4.1 Data from Experiments

Let us first do several experiments.

If we use the PODEM algorithm (implemented by ourselves) with heuristics, which will be discussed later, we can try to discover what the relationship is between backtrack limits and what percentage of faults are *solved faults*. Here, a *solved fault* is a fault for which a test pattern is found or its redundancy is proved.

It is reasonable to take the number of primary inputs as a unit of backtrack limit. For example, C432[23] has 36 primary inputs, the backtrack limit is assigned as 36, 2×36, etc. The following explains why it is reasonable.

1. Different circuits have different sizes. A test set for a circuit with only 10 logic gates can be generated within a constant backtrack limit k. For a

Table 4.1: Statistical Data for Circuits

Circuit	PI	Gates	Faults	Solved Faults		Percentage(%)	
				Step1	Step 2	Step1	Step 2
C422	36	160	524	471	471 + 45	89.88	98.47
C499	41	202	758	254	254 + 496	33.50	98.94
C880	60	383	942	872	872 + 44	92.56	97.23
C1355	41	546	1574	350	350 + 1152	22.23	95.42
C1908	33	880	1879	1369	1369 + 459	72.85	97.28
C2670	233	1193	2747	2678	2678 + 0	97.48	97.48
C3540	50	1669	3428	3180	3180 + 96	92.70	95.50
C5315	178	2307	5350	5259	5259 + 8	98.20	98.40
C6288	32	2416	7744	6274	6274 + 1457	81.01	99.83
C7552	207	3512	7550	7073	7073 + 12	93.68	93.84

circuit with 1000 logic gates, within the same backtrack limit k, no test pattern is likely to be found even for one fault. Hence, it is not a good idea to take a constant value as a backtrack limit for all circuits, without considering the difference between their sizes.

2. In combinational circuits, the size of a circuit is strongly related to the number of primary inputs. In other words, to some extent, the number of primary inputs represents the size of the circuit.

We assign the backtrack limit as the number of primary inputs (one unit) and double the number of primary inputs (two units), respectively, and observe the number of solved faults and percentage of faults. For 10 typical circuits[23], we obtain the following 10 groups of data, which are represented by Table 4.1.

In those tables, every unit is the number of primary inputs of the circuit under

test. For example, circuit C499 has 41 primary inputs. Its unit is 41. Here, 499 represents the number of connecting lines in the circuit. The number of faults is a reduced equivalent fault set based on equivalence fault collapsing[17]. The number of faults to be tested can be reduced by combining, for example, indistinguishable faults into a single set[30]. "Indistinguishable faults", are faults such that there is no test to distinguish between them. Therefore, when generating a test for an n-input AND (OR) gate only $(n+2)$ rather than $(2n+2)$ faults of the gate need to be tested. A systematic approach that reduces the number of faults that have to be tested is based on the idea of fault equivalence classes, i.e. such faults that are covered by a single test set[30].

4.2 Inference from Experiments

In the previous section, some experimental results about automatic test pattern generation were discussed. Now, we will see what kind of conclusion we can deduce.

Our data tells us that over 93% faults are solved faults if the backtrack limit is double the number of primary inputs. For the circuit C6288, over 99% of the faults are solved faults. This fact implies that in a system which contains massive processors, fault partitioning can work very efficiently to generate test patterns for most faults. (Why? because most faults are easy to detect). The next stage is the time to concentrate on how to coordinate all processors to solve the remaining faults, which are called *hard-to-detect faults*.

If there is a system which consists of massive processors, the fault partitioning method can be used to efficiently solve the test pattern generation problem for

most of the faults (93% or over), namely, the easy-to-detect faults. There is no communication problem among processors after each processor is assigned a subset of faults. The backtrack limit is double the number of primary inputs, which is a small integer. This implies that there is no big difference between the amount of work done by each processor. The test set may, however, contain many redundant test patterns.

After this first step, the remaining faults are hard-to-detect faults. Any one of them may require several hours, days or even weeks if one conventional ATPG algorithm and one processor are used. It was also shown that current parallel systems are still unsatisfactory for the solution of the automatic test pattern generation problem.

Therefore, it is time to think about designing a special structure to interconnect many processors in order to make a group of processors solve the automatic test pattern generation problem more effectively, for those hard-to-detect faults.

PartII

Simulation Environment

Chapter 5

Model for Measurement

Some kind of a measurement model is important to measure the quality of a system, or to allow meaningful comparisons of different systems. The measurement model must encapsulate the essential functionality of the system in a quantifiable manner.

5.1 Model for Measurement

If there is a system which consists of lots of processors and has a specific topology to connect these processors, can the quality of this system be measured? Here quality means the quality of the system for parallel processing of automatic test pattern generation.

One criterion for evaluating the quality of a parallel solution to a problem is how well it scales. There are two aspects which play important roles in the quality of a parallel solution. One is the algorithm, the other is the topology of a multi-processor system.

So far, most research has concentrated on the design of parallel algorithms, which can be executed by a specific multi-processor system. The measure of the

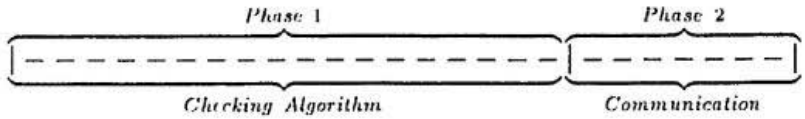


Figure 5.1: Virtual 2 Phase Clock

quality of a parallel solution is determined by how well the algorithm scales. An algorithm scales well if the computation time decreases linearly, or nearly so, with an increase in the number of processors in the system. The speedup of a given parallel algorithm is defined as the ratio of the time taken by the fastest sequential algorithm running in an equivalent uniprocessor to the time taken by the parallel algorithm on the parallel machine. The goal is to have the algorithm's speedup scale linearly with the number of processors.

Since our goal is to design a multi-processor system which has a high performance when generating test patterns, the quality of a parallel solution is how well the *system* scales. Analogous to the case for a parallel algorithm, a parallel system scales well if the computation time decreases linearly, or nearly so, with an increase in the number of processors in the system. The speedup of a system is defined as the ratio of the time taken by the algorithm running in one processor to the time taken by the algorithm on the multi-processor machine. The goal is to have the system's speedup scale linearly with the number of processors.

A multi-processor system is a tuple (P, C, p_i), where P is a set of processors, and every processor is identical. C is a set which contains interconnection information for the processors. All processors execute their own algorithms synchronously under the control of a virtual two phase clock (2PC). Figure 5.1 shows the dia-

gram of 2PC. It is also required that these checking algorithms have the same time complexity. This is very important since all processors work in synchronization. The period of 2PC depends on the lowest speed processor. Since all processors in P are the same, the greater the time complexity, the lower the speed. A low speed will cause other processors to wait.

$p_i \in P$, p_i is the master processor of the system. The master processor is in charge of coordinating the system. For example, it receives tasks from outside, and is the first processor to begin to generate test patterns. It decides whether it is time to stop working because one test pattern has been found to detect a given fault, or time has run out, or a redundant fault is found. It also should report the result to the outside. C determines the topology of a multi-processor system since a topology depends on the connections among processors. If there is a connection between two processors, it means that there is a wire between them on the physical level. Limitations are needed for C because it is impractical to have many wires to input or output data for each processor. Later, we will show that all our multi-processor systems have a 4 connected structure, two for inputs and two for outputs. This results in a simple and natural layout.

In order to measure the quality of a system, the number of two phase clock (2PC) steps is used. The number of 2PC steps (N2PC) is counted to record how many N2PC are used to find a test pattern, or a redundant fault, or a hard-to-detect fault. If a multi-processor system can generate all possible 2^k values for any given integer k, a test pattern can be found eventually — as long as it exists, or a conclusion of redundancy can be reached. Hard-to-detect faults are those whose test pattern has not been found within the specified time limit.

With an increasing number of processors, that is $\mid P \mid \rightarrow \infty$, every N2PC is recorded. These data are analyzed to determine whether a multi-processor system scales well, which represents the quality of the system.

5.2 Parallel Speedup in Test Pattern Generation

In 1864, the philosopher Charles Babbage said:

> It is impossible to construct machinery occupying unlimited space; but it is possible to construct finite machinery, and to use it through unlimited time. It is this substitution of the infinity of time for the infinity of space which I have made use of to limit the size of the engine and yet to retain its unlimited power.

We may call this Babbage's thesis. This thesis states that time and space complexity are related and can be traded for one another. As hardware technology develops, we can employ the converse of Babbage thesis: use a very large number of processors to solve the test pattern generation problem. That is to say, we can use space to gain invaluable time.

The central issue in parallel and concurrent processing using a large number of processors is the design of multi-processor systems and parallel algorithms whose performance can be somehow related to the time complexity of the single-processor sequential algorithm, T_1. Ideally, we require that a parallel algorithm which takes a problem and uses N processors in time T_N is related to T_1 by the relation $T_N = T_1/N$. In other words, we hope that a multi-processor system with N

independent processors should be able to compute the solution of a problem N times faster than a single processor. This is called *ideal speed-up*. However, in practice, this speed-up ratio T_1/T_N often turns out to be far less than N for the following reasons:

1. Processors competing for the same communication paths with other processors or to a shared memory can slow down because of the non-availability of paths.

2. Since simultaneous reading and writing from a file can cause conflicts, the processors are forced to wait for mutual exclusion.

3. Processors need to be conditionally synchronized when different tasks are to be coordinated.

4. The sequential component in an algorithm limits the speed of the total process; in other words, if T_s and T_p are respectively the time spent on serial and parallel components of an algorithm in a single processor, then the maximum speed-up S_N that can be achieved using N processors in parallel for the parallel component is given by:

$$S_N \leq \frac{T_s + T_p}{T_s + \frac{T_p}{N}} = \frac{1}{f + \frac{(1-f)}{N}}$$

where $f = \frac{T_s}{T_s+T_p}$ and $0 \leq f \leq 1$.

We can find that f is the fraction of computations performed sequentially. For example, if $f = \frac{1}{k}$, where $k > 1$, then $S_N \leq k$, even if N is very large; obviously, for $f = 0$, $S_N = N$. This is called *Amdahl's law*.

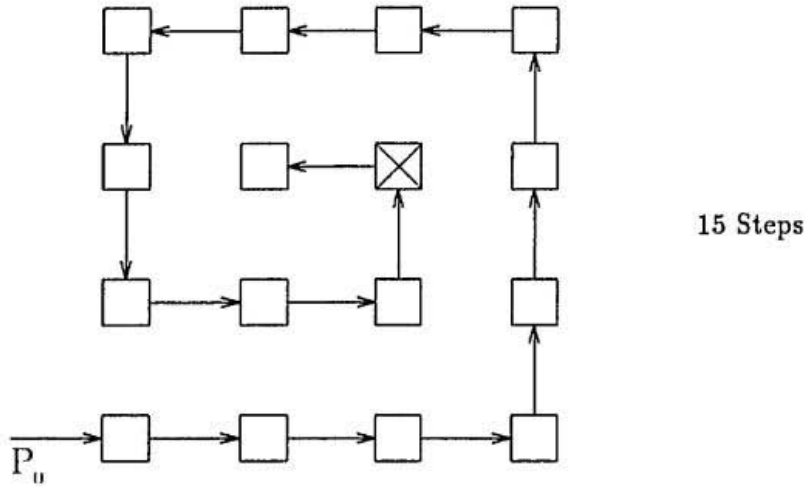


Figure 5.2: One Processor Searches 16 Elements

However, the test pattern generation problem is anomalous. To find a test pattern for a given fault can be considered equivalent to searching a space. N cooperating processors may reach the goal much faster than one processor, even more than N times faster.

Suppose there is a space which has 16 elements. One processor P_0 searches the space according to some heuristics. The goal element can be reached after 15 steps, as shown in Figure 5.2. If four processors, P_0, P_1, P_2 and P_3, take part in the search according to the same heuristics, each will search a sub-space, as shown in Figure 5.3. The goal element is found by P_3 after one step. T_1/T_4 is 15, which has greater than 4. This means that there is greater than linear speedup, called *superlinear speedup*.

This anomaly can also be seen from another point of view. In general, a problem contains n sub-problems. In order to solve this problem, the processing

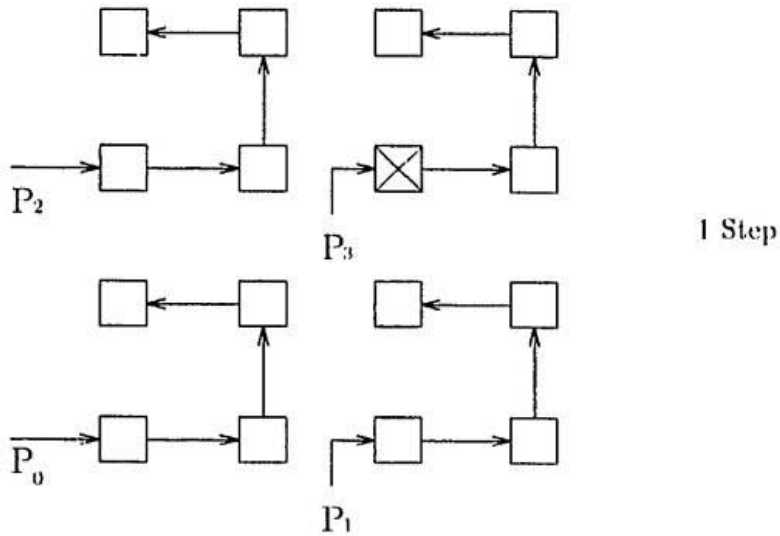


Figure 5.3: 4 Processors Search 16 Elements

elements in a parallel system have to cooperate with each other to solve all of these n sub-problems. But for the test pattern generation, if a test pattern is found, all of the remaining computation can be omitted. Therefore, not all of the sub-problems must be done.

If a circuit has N_{pi} primary inputs, the test pattern generation problem for this circuit can be divided into $2^{N_{pi}}$ sub-problems according to the representive at its primary inputs. Every sub-task is to solve one sub-problem, which is to check whether the given pattern in primary inputs can detect the given fault. If one of the sub-tasks is done and a test pattern is found, all of the remaining unfinished sub-tasks can be ignored. Again, consider the examples in Figure 5.2 and Figure 5.3. In Figure 5.2, when a test pattern is found, 15 sub-tasks have been done, which occupies 93% of all sub-tasks. But in Figure 5.3, only 25% of all sub-tasks

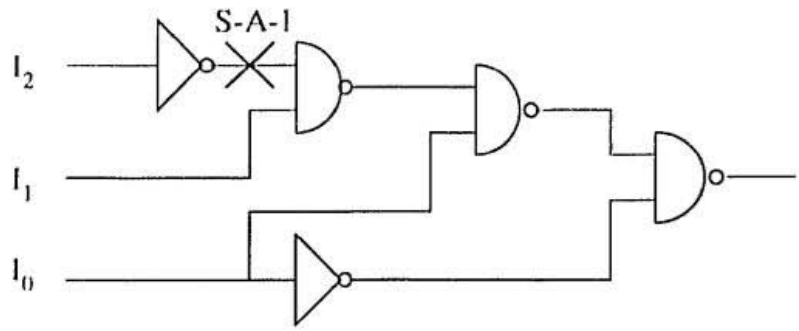


Figure 5.4: A Circuit with One Fault

Step	I_2	I_1	I_0	Status
1	X	X	X	potential
2	0	X	X	cannot
3	1	X	X	potential
4	1	0	X	cannot
5	1	1	X	potential
6	1	1	0	cannot
7	1	1	1	cannot

Table 5.1: A Process of the Proof of a Redundant Fault

are done when a test pattern is found. Although only part of the sub-tasks are done, the test pattern generation problem is solved. This is one of the anomalous characteristics of the problem.

How about the redundant fault? Should all sub-tasks be done in order to prove its redundancy? The answer is "No", again.

Step	I_2	I_1	I_0	Status
1	X	X	X	potential
2	X	X	0	cannot
3	X	X	1	cannot

Table 5.2: A Process of the Proof of a Redundant Fault

For example, consider the circuit in Figure 5.4. Table 5.1 and Table 5.2 are two examples which show the process to prove the redundancy of the fault. They clearly show that: it is possible for the proof of a redundant fault to do only part of the sub-tasks, if an unknown value X is used as one of the primary input's value. These two tables also show that the different order of assigning values to primary inputs may cause a different number of sub-tasks to be done. Table 5.1 arranges I_2, I_1, I_0 as the order of assignment. Seven sub-tasks are done to prove the redundancy. But Table 5.2 selects I_0 as the first primary input to have its value assigned. It requires only 3 sub-tasks to prove the redundancy. In general, this order of primary inputs is dependent on what heuristics are adopted. Hence, it is not necessary for the proof of a redundant fault to do all of the sub-tasks. Moreover, the heuristics play an important role in deciding how many sub-tasks should be done.

So far, there is no standard method to measure the speedup of parallel systems for test pattern generation. T_1/T_N is the method widely used [13][8].

For the automatic test pattern generation problem, we may expect a multi-processor system to generate a test pattern for a given fault very quickly if the test pattern exists and there are enough processors which are interconnected. For example, there is a circuit with N primary inputs. To detect a possible stuck-at fault, there are 2^N different patterns which can be fed to the circuit. These 2^N different patterns form a pattern set. This pattern set is called the *search-space* because ATPG algorithms always try to search this set so as to find a pattern which can detect the given fault. If there is a multi-processor system which contains $2^{N+1} - 1$ processors, which are connected to form a complete binary

tree, then, within N steps, a test pattern can be found if it exists, or a redundant fault can be proved if there is no test pattern for the given fault.

In practice, it is impossible to have such a system since 2^N is a huge number when N is a little bit large, say $N > 20$. Therefore, the problem is how to use limited resources, or processors, to find a test pattern in the search-space as quickly as possible. More exactly, the problem of exploring linear speedup is to design a topology to connect given N processors and a protocol to make these N processors communicate with each other so as to find a test pattern or prove its redundancy N times faster than when one processor is used.

Chapter 6

Algorithms

This chapter introduces all the algorithms used in our simulation software. Their time complexities are also discussed.

6.1 Parser Construction

To generate test patterns, first of all, circuits have to be analyzed. The description of circuits is written in a netlist format[23]. This section discusses the grammar rules of the netlist format; the format is described in detail in Byron [23]. Based on the grammar rules, a parser can be developed directly.

6.1.1 The Grammar Rules

The description of the netlist format from Bryan[23] is a list of descriptions of logic gates. The description of a logic gate is called a *node* since each gate, or primary input, or fanout branch is considered as a *node*. Using the form of YACC[36], we can use the following grammar rules to represent these:

```
circuit   : node_list
```

```
node_list : node
          | node_list node
```

A *circuit* is a list of nodes, denoted by *node_list*. A *node_list* is described in a recursive way. A *node* forms a *node_list*. A *node_list* followed by a *node* also forms a *node_list*.

From the netlist format, *nodes* can be classified into three types: primary inputs, fanout branches, and logic gates. They have different formats.

Primary inputs have the format:

```
address name INPT fanout ZERO faults
```

Fanout branches have the format:

```
address name FROM name faults
```

Logic gates have the format:

```
address name type fanout fanin faults fanin_line
```

Here, *type* represents the type of the gate, for example, AND, NAND, OR, etc.

Using grammar rules, a *node* can be written as:

```
node      : address name INPT fanout ZERO faults
          | address name FROM name faults
          | address name type fanout fanin faults fanin_line
```

Since we adopted the reduced equivalent fault set, which is based on equivalence fault collapsing[23], there may be some nodes labeled no fault, some labeled stuck_at_0, some stuck_at_1, some labeled both. Therefore, the grammar rules of *faults* can be described as

```
faults    :
          | S_A_0
          | S_A_1
          | S_A_0 S_A_1
          | S_A_1 S_A_0
```

The complete grammar rules can be listed as follow:

```
circuit   : node_list

node_list : node
          | node_list node

node      : address name INPT fanout ZERO faults
          | address name FROM name faults
          | address name type fanout fanin faults fanin_line

address   : integer

name      : STREAM
          | N_ZERO
          | ZERO

type      : AND
          | NAND
```

```
          | OR
          | NOR
          | XOR
          | NXOR
          | BUFF
          | NOT

fanout    : integer

fanin     : N_ZERO

faults    :
          | S_A_0
          | S_A_1
          | S_A_0 S_A_1
          | S_A_1 S_A_0

fanin_line   : address_list

address_list : address
             | address_list address

integer   : ZERO
          | N_ZERO
```

Here, the *address* should be an *integer*. The *integer* is a zero or a non_zero value. The *name* can be a zero, or non_zero value, or a string of characters. The *fanin* and the *fanout* are integers. The *fanin* cannot be zero since each logic gate must have at least one input. A logic gate has several inputs, whose addresses are put in the *fanin_line* field.

With each grammar rule, actions may be associated to be performed each time the grammar rule is recognized in the input process[36]. Then the netlist format can be analyzed, and the needed data structure can be constructed.

6.2 Compiler Driven Simulation

In a multi-processor system, there are a lot of processing elements, we call them checking processing elements (CPE), which simultaneously do the same task, checking whether the given trial test pattern can, or cannot, or is possible to detect a given fault. In order to do this job, every CPE does its work in two steps:

1. simulate the logic circuit

2. check the result

In simulation, there are two basic classes of simulators, compiler driven and table-driven event-directed. The earliest simulators were of the former type, but most modern ones are of the latter type since they allow for more versatility in handling delays as well as a reduction in simulation time. In our case, the compiler driven method is adopted. In our systems, all processors work synchronously under the control of the virtual 2 phase clock (2PC). For the worst case, the table-driven event-directed method has to simulate all logic gates in the circuit

since every gate is active. This situation has to be considered when we calculate the time period of virtual 2 phase clock. In a synchronous system, the period of 2PC is the time for the worst case. If table-driven event-directed method is used, processors in the best cases have to wait for processors in the worst cases. In other words, the saved time is wasted because the faster processors are idle in order to wait. If the compiler driven method is adopted, it is much easier to estimate the time period of 2PC since every processor runs the same executable code. Therefore, the compiler driven method is more suitable for our case.

Compiler driven simulation first translates the description of circuit into a list of logic gates, which is called *the machine executable gate list*, which is arranged according to *the machine executable order*. This machine executable order guarantees that the circuit simulation can be done by simulating each logic gate in the list one by one according to their order in the list. This subsection first introduces the circuit levelizing algorithm which translates the description of circuit into a machine executable ordered list. Then the simulation algorithm will be discussed.

Forming machine executable gate list

In any logic circuit, each logic gate can be assigned a level value. The level value decides when the logic gate can be simulated. For example, there are two logic gates G_i and G_j, which have level values k_{G_i} and k_{G_j}, respectively. The order of simulation for G_i and G_j satisfies the following restrictions:

1. If $k_{G_i} < k_{G_j}$, G_i must be simulated before G_j.

2. If $k_{G_i} = k_{G_j}$, G_i and G_j can be simulated in any order.

3. If $k_{G_i} > k_{G_j}$, G_i must be simulated after G_j.

It is clear that the level value of each logic gate imposes a partial order on the simulation.

The following circuit levelizing algorithm assigns each logic gate a level value:

1. Assign all primary input lines x and feedback lines y the level value 0.

2. For any element not yet assigned a level value, assign this element and its output lines a level value as defined by

$$k_e = 1 + max(k_{i_1}, k_{i_2}, \cdots, k_{i_j})$$

where k_i is the level value of element i, and element e has inputs from elements i_1, i_2, $\cdots$, i_j.

To implement this levelizing algorithm, the following algorithm was designed. For convenience, each primary input is considered as a special logic gate.

1. Find all primary inputs, assign a level value 0, and put them into an assigned queue;

2. While (the assigned queue is not empty)

   (a) Get one logic gate G from the assigned queue;

   (b) For (each logic gate G_o driven by G) do

      i. If (G_o already has been assigned one level value) Skip;

      ii. If (at least one of the logic gates which drive G_o has not been assigned a level value) Skip;

iii. Otherwise, all logic gates which drive G_o have been assigned a level value

A. Assign $k_{G_o} = 1 + max(k_{i_1}, k_{i_2}, \cdots, k_{i_j})$;

B. Put G_o into the assigned queue;

Suppose the number of logic gates in a circuit is m, the maximum number of fanin is f_i, and the maximum number of fanout is f_o. Now the complexity of the algorithm can be analyzed.

step 1: In the worst case, after m steps, all the primary inputs can be found. Therefore, the time complexity is $O(m)$.

step 2: Each logic gate will be entered, only once, into the assigned queue, so this while statement will be executed m times.

step (a): Constant time is required for it.

step (b): for statement will run f_o times.

step i: It needs Constant time.

step ii: f_i time is required to do this judgment.

step iii: It is clear that it is constant time.

step A: $O(f_i)$ time is needed.

step B: It is constant time.

The total time needed is:

$$O(m) + m(C + f_o(C + f_i + C + f_i + C)) = O(m) + O(f_i f_o m) = O(f_i f_o m)$$

For all circuits, each logic gate has a limited number of fanin and fanout. It can be assumed that they are less than a constant C. Hence, the time complexity is $O(m)$.

After levelization, any sorting program can be used to put all the logic gates in a special order. That is, if $k_{G_i} < k_{G_j}$, G_i is before G_j.

In VLSI circuits, there are many logic gates. It is worth using an economic sorting method to rearrange these gates. *Quicksort* is one of the widely used methods since it has a best-case time $O(n \log n)$ [35]. It also has a worst-case time $O(n^2)$ [35]. *Heapsort* can sort these data within time $O(n \log n)$ [35], but it needs a little bit more space. Either of them can serve the purpose.

After sorting, an ordered gate list is formed. This list is called a *machine executable gate list.*

Simulation Algorithm

Based on the machine executable gate list, the circuit simulation becomes easy and direct.

From head to tail, for each logic gate in machine executable gate list, do

1. Get all its input values;

2. Simulate the logic gate based on the five value logic $\{0, 1, X, D, \overline{D}\}$.

Suppose the maximum fanin among all logic gates is k. To simulate each logic gate, k steps may be needed to fetch input signals. Since the machine executable gate list contains m logic gates, the time complexity of the simulation algorithm is $O(km)$. In practice, the fanin of a logic gate is a limited value, say 4 or a little

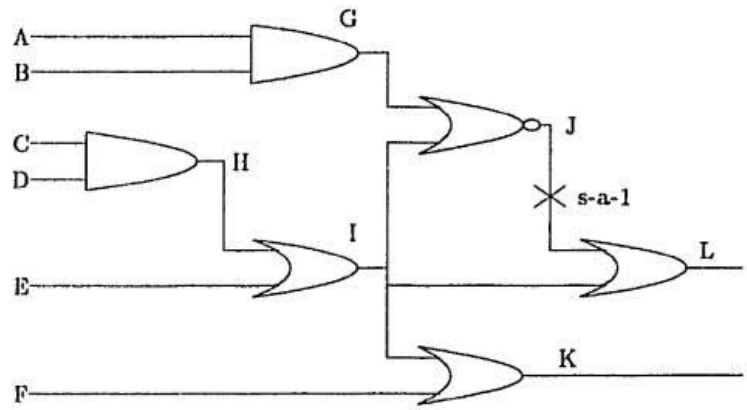


Figure 6.1: Ineffective Logic Gates

more. The maximum fanin can be considered as a constant value. Therefore, the time complexity of simulation algorithm is $O(m)$.

This machine executable gate list can be made smaller if a fault, which should be detected, is given. In a circuit, some logic gates do not affect the test pattern generation for a given fault.

For example, the logic gate K in Figure 6.1 does not affect the test pattern generation for the fault shown in the figure. Therefore, gate K can be deleted from the machine executable gate list, making the list smaller. Hence, simulation time can be saved.

To reduce this list, another algorithm is needed. Here, we describe it in natural language, it is trivial to code it in a programming language. First, two concepts are defined.

A *fault transferring gate* is a logic gate such that, if at least one of its inputs carries a faulty signal, then its output is also a faulty signal.

A useful gate is a logic gate, which drives at least one useful signal and its inputs are all useful signals.

At the faulty point, the faulty signal must be propagated forward and useful signals propagated backward.

Based on these definitions, the algorithm can be described. Beginning from the faulty point, a fault transferring signal can be propagated forward. All fault transferring gates can be labeled. Also beginning from the faulty point, a useful signal can be propagated backward. All useful gates can be found. Those gates, which are neither fault transferring gates nor useful gates, can be deleted from the machine executable gate list since they do not affect the test pattern generation for the given fault.

The penalty is that the changing of a fault causes the changing of the executable gate list. As we know, our goal is to solve the problem for hard-to-detect faults. This may necessitate that the simulation be executed many times. Therefore, it is still worth doing so because simulation becomes quicker.

6.3 Checking Trial Test Patterns

After the simulation, we use the following checking algorithm to check whether the trial test pattern can detect the fault.

1. If all primary outputs are 0 or 1, it cannot detect the given fault.

2. If at least one primary output is D or $\overline{D}$, a test pattern for the given fault is found.

3. Else, there is no D either $\overline{D}$. And there is at least one X in the primary outputs. Do

   (a) Set an empty set which is used to contain any logic gate which is found can possibly propagate the fault to primary outputs;

   (b) If the logic gate which has a stuck-at fault, has the value D, $\overline{D}$ or X, put it into the set;

   (c) While the set is not empty, Do

      i. Get one logic gate from the set

      ii. If the logic gate is a primary output gate with value X, return a POTENTIAL_TEST_PATTERN flag, which means that the input pattern has the possibility to detect the fault. As discussed before, the input pattern is a potential test pattern.

      iii. If the logic gate is not a primary output gate, put into the set all those logic gates which are driven by the logic gate, have value D, $\overline{D}$ or X, and have not been in the set before.

   (d) If the set is empty, return a NOT_TEST_PATTERN flag, which means that this input pattern is impossible to detect the fault.

Suppose the maximum fanout in the circuit is k, and a circuit contains m logic gates. We can analyze the time complexity of the checking algorithm.

step 1: In the worst case, all gates are primary outputs, m gates are needed to be checked. Therefore, the time complexity is $O(m)$.

step 2: To check the result, m gates may have to be scanned. The time complexity is still $O(m)$.

step 3: After step 1 and step 2, the result is obvious. It can be thought as constant time.

step (a): Constant time is required for it.

step (b): It is still constant time.

step (c): Since every gate may be put in the test set, the **While** statement may be executed m times.

step i: It needs constant time.

step ii: Clearly, it is constant time.

step iii: k steps are needed because the logic gate may drive k gates.

step (d): Constant time.

According to the structure of the algorithm, the total time is:

$$O(m) + O(m) + C + C + C + m(C + C + k) + C = O(km)$$

In practice, the fanout of a logic gate is a limited value. Therefore, we can take k as a constant. Hence, the algorithm has $O(m)$ time complexity.

6.4 Heuristics

Heuristics are very useful to speed up the test generation since the test pattern generation problem is NP-complete in general. Here, we use the testability mea-

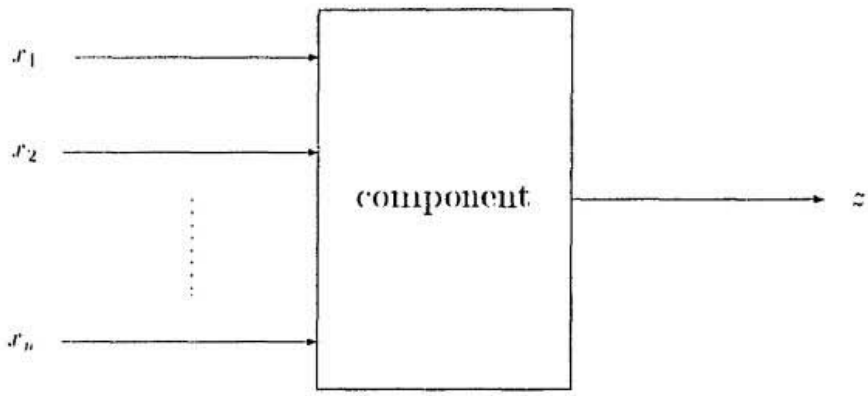


Figure 6.2: Diagram for a Component

sure of Stephenson and Grason[17] as the heuristics to generate a test pattern for a given fault.

A register transfer level circuit can be assumed to be a network of components (e.g., adders, registers, multiplexors, controllers) interconnected by unidirectional links. In general, a link may be many conductors carrying more than one bit of information; however, to simplify our discussion we assume here that every link has a single conductor. A link is a signal line carrying logic values 0 and 1.

A controllability value $CY(s)$ and observability value $OY(s)$ ranging from 0 to 1 are assigned to each signal line s.

Consider the component illustrated in Figure 5.2. The expression used to calculate CY for output z is

$$CY(z) = CTF \times \frac{1}{n} \sum_{i=1}^{n} CY(x_i)$$

where CTF is the controllability transfer factor of the component and n is the

number of inputs of the component.

The concept of the CTF is used to account for the potential diminishing of control information as it is propagated through the circuit. The CTF of a component must represent the ability to control output of the component by applying input values. It is defined by the following equation, depending only on the input-output relation of the component:

$$CTF = 1 - \frac{| N_z(0) - N_z(1) |}{2^n}$$

where $N_z(0)$ and $N_z(1)$ are the numbers of input values for which output z has output value 0 and 1, respectively. The CTF of a component ranges between 0 and 1. It takes the maximum value 1 when the component has a uniform input-output relation, and decreases to 0 as the degree of uniformity decreases. For example, the CTFs for a NOT gate and an XOR gate are 1, since $N(0)$ and $N(1)$ are equal. On the other hand, the CTF of an n-input NAND gate is $\frac{1}{2^n}$.

Consider again the component diagrammed in Figure 5.2. The expression used to calculate OY's for each input x_i is

$$OY(x_i) = OTF \times OY(z)$$

where OTF is the observability transfer factor of the component. Note that each input observability is assigned the same value.

The OTF of a component must represent the case of propagating a fault value through the component. It is expressed as

$$OTF = \frac{1}{n} \sum_{i=1}^{n} \frac{NS_i}{2^n}$$

where NS_i is the number of input values for which output resulting from changing the input value of x_i are different. NS_i also means the number of input values that can sensitize a path from x_i to the output of the component. The OTF measures the probability that a faulty value at any input of the component will propagate to its output. The values of OTFs also vary between 0 and 1.

As discussed before, controllability transfer factor (CTF) is used to account for the potential diminishing of control information as it is propagated through the circuit. The CTF of a component represents the ability to control the output of the component by applying input values. The observability transfer factor of the component represents the ease of propagating a fault value through the component. It also measures the probability that a faulty value at any input of the component will propagate to its output.

So to forward a faulty value to output, we can use the observability measure to get the "most potential" logic gate which makes the fault be propagated to primary outputs as early as possible. In the backward propagation of supporting values, the controllability measure is used to gain the "most potential" primary input which makes the test pattern generation terminate as early as possible.

From the discussion of testability, we can find that it is suitable to select a logic gate which has the greatest observability in the D frontier as the element with the "most potential", and to select a primary input located on *the most controllable path* from the selected logic gate. From a given logic gate, we always select a logic gate which has the greatest controllability among its input gates. All these gates form a path. This path will terminate at a primary input. This path is called *the most controllable path.*

From another point of view, the selection of primary inputs can also depend on *the worst controllable path*. Similar to the most controllable path, the worst controllable path begins from a logic gate and ends at a primary input. Each gate on the path is chosen because it has the least controllability among all logic gates which drive a gate, which is already in the path.

The worst controllability path is the complement of the most controllability path. Since there is no way to guarantee the most controllability path is really the best or the quickest path to generate a test or prove a redundant fault, the complementary method may be better for some situations. They may rule out some fruitless search space quickly. This quickness can also accelerate the test pattern generation. The selection of this least controllability becomes complementary heuristics. Later, these complementary heuristics will be used in our autonomous architecture to generate test patterns.

6.5 Expansion of Trial Test Patterns

To generate a test pattern for a given fault, first all primary inputs are assigned unknown value Xs, which means that it may be a value 0 or 1. This pattern works as the first *trial test pattern*. A recursive method can be used to define a trial test pattern:

Suppose a circuit has N_{pi} primary inputs $I_1 I_2 \cdots I_{pi}$

1. $X_1 X_2 \cdots X_{pi}$ is a trial test pattern, where X_i means that the primary input I_i has the unknown value X.

2. Suppose $a_1 a_2 \cdots a_{j-1} X_j a_{j+1} \cdots a_{pi}$ is a trial test pattern, where a_i means that the primary input I_i has a value a and $a \in \{0, 1, X\}$. After simulation and checking algorithms, it is found that this pattern is a potential test pattern. Then, we can say that

$$a_1 a_2 \cdots a_{j-1} 0 a_{j+1} \cdots a_{pi}$$

and

$$a_1 a_2 \cdots a_{j-1} 1 a_{j+1} \cdots a_{pi}$$

are two trial test patterns.

A potential test pattern $a_1 a_2 \cdots a_{pi}$ may contain several X values. The task of our expansion algorithm is to select one and assign it 0 and 1, therefore two trial test patterns are generated.

1. Set a temporary set empty;
2. If the logic gate which has a stuck-at fault, has the value D, or $\overline{D}$ put it into the set; otherwise, it is X according to the conclusion of the checking algorithm. We take this gate as the most potential forward node PFN.
3. While the temporary set is not empty, Do
   (a) Get one logic gate from the temporary set
   (b) For all of the logic gates driven by it,
      i. If the logic gate outputs D or $\overline{D}$, put it into the set if it has not been there before.

ii. If the logic gate outputs 0 or 1, skip.

iii. If the logic gate outputs X, compare it with the gate labeled by PFN.

A. If it is more suitable according to the observability value, change the PFN flag to this gate.

B. Otherwise, keep the PFN flag unchanged.

4. Beginning from the gate labeled PFN, DO

(a) If the gate is a primary input, this input is assigned 0 and 1 separately. Therefore, two trial test patterns are formed.

(b) If it is not a primary input, select the most suitable gate in all the gates which drive it according to their controllability value. Goto step 4.(a).

It is not difficult to show that the time complexity of this algorithm is $O(m)$ if the maximum fanin and the maximum fanout are considered as constants. Step 3 may be executed m times. Step 3.(b) may be run M_{fanin} times. Step 4.(a) and 4.(b) may be executed M_{fanout} times as well. Therefore, the time complexity of this algorithm is $O(m)$.

6.6 Detection of Redundant Faults

If a fault is redundant, it can be detected after whole search space is searched. In a multiprocessor system, a fault is proven redundant if all processors are idle in the same phase clock. If in any clock phase, some processors are idle and some are busy, then some subsearch space is being searched. Therefore, it is possible

that there are some test patterns. If all processors are idle during the same phase clock, each processor has no potential test pattern in its local memory, nor in its input ports; there are no potential test patterns to be generated. Therefore, all processors are idle during next phase clock. This implies that full space has been searched.

To detect whether all processors are idle during one clock phase, a special processor is designated. When this processor is idle, it sends *an idle detection signal* to its children. The idle detection signal contains the time when it is sent and *flag space* for processor. Suppose there are n processors, the flag space contains n flags. Each processor corresponds to one flag in the flag space.

When a processor receives an idle detection signal, it checks whether it has been idle since the time specified in the idle detection signal. If it is idle, it sets the flag in the flag space. If it is not idle, it resets the flag. Then it sends this idle detection signal to its children.

After a period of time, this idle detection signal is received by the designated processor. If all the flags in the signal are set, it knows that all processors have been idle since that time, and consequently the fault is found redundant. This is the protocol we uses to prove the redundancy of a fault.

This chapter discussed all of the algorithms relating to the simulation in detail. These algorithms have been used in the code for the simulations described in the later chapters. They lay the foundation for the methods described in the following chapters.

PartIII

ATPG and 4 Connected Architecture

Chapter 7

Four Connected Topology

This chapter defines the 4 connected structure, and discusses some of its characteristics.

7.1 4 Connected Structure and Examples

A 4 connected structure is a graph. If a node is denoted by v, its four ports are represented by v_0, v_1, v_2, and v_3. v_0 and v_1 are two input ports And v_2 and v_3 are two output ports. Each input port is fed from one output port of a node, and each output port is connected to only one input port of a node. A 4 connected structure can be defined formally. In following definition, we use V to represent the node set, V' the port set, and E the connection set. If $v \in V$, v_i denotes one port of the node v. Sometimes, there are several nodes, for example, x, y, z. In order to distinguish their ports, we use x_{i_x}, y_{i_y}, and z_{i_z} to denote one of x's port, one of y's, and one of z's.

Definition Given a finite node set V, a 4 connected structure (4CS) is a directed graph derived from V, $G = (V', E)$ where V' and E are derived from the

finite node set V.

$$V' = \{v_0, v_1, v_2, v_3 \mid v \in V\}$$

$$E \subseteq V' \times V'$$

E satisfies the following conditions:

1. If $v \in V$, then there exist four nodes $x, y, z, w \in V$, (some of them may be the same node). The ports of v have following relations: $(v_2, x_{i_x}) \in E$, $(v_3, y_{i_y}) \in E$, $(z_{i_z}, v_0) \in E$, and $(w_{i_w}, v_1) \in E$. Here $i_x, i_y \in \{0, 1\}$, and $i_z, i_w \in \{2, 3\}$.
2. If $(v_k, x_{i_x}) \in E$ and $(v_k, y_{i_y}) \in E$, then $x_{i_x} = y_{i_y}$.
3. If $(z_{i_z}, v_k) \in E$ and $(w_{i_w}, v_k) \in E$, then $z_{i_z} = w_{i_w}$.

Condition 1 says that each $v \in V$ has 2 inputs, v_0, v_1, and 2 outputs, v_2, v_3. Each input port, v_0 or v_1, is fed by one and only one output port, and each output port, v_2 or v_3, feeds one input port. Condition 2 guarantees that each output is connected to only one input. Condition 3 ensures that each input is fed by only one output. Followings are two examples of four connected structures.

Example 1 A directed graph $G_1 = (V', E)$ is given by the sets

$$V = \{0\}$$

$$V' = \{0_0, 0_1, 0_2, 0_3\}$$

$$E = \{(0_3, 0_0), (0_2, 0_1)\}$$

It is clear that G_1 is a 4 connected structure (4CS). Figure 7.1 is a diagram, which represents the graph G_1.

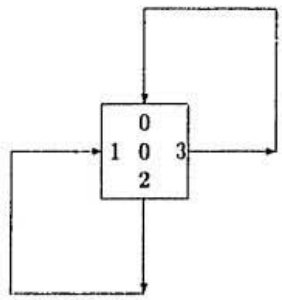


Figure 7.1: The Diagram for Graph in Example 1

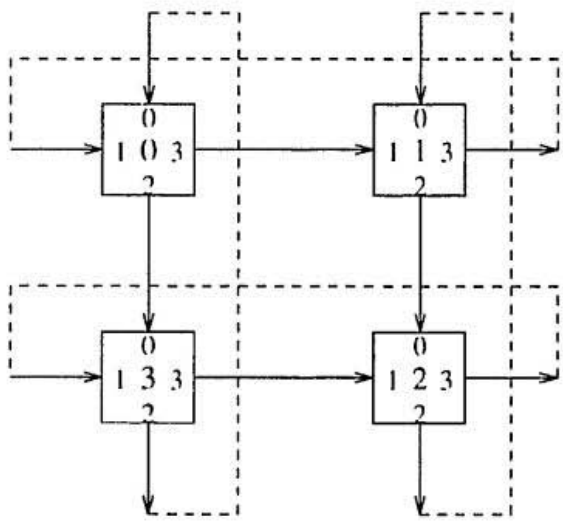


Figure 7.2: The Diagram for Graph in Example 2

Example 2 A graph $G_2 = (V', E)$ is given by the sets

$$V = \{0, 1, 2, 3\}$$

$$V' = \{0_0, 0_1, 0_2, 0_3, 1_0, 1_1, 1_2, 1_3, 2_0, 2_1, 2_2, 2_3, 3_0, 3_1, 3_2, 3_3, \}$$

$$E = \{(0_2, 3_0), (0_3, 1_1)(1_2, 2_0), (1_3, 0_1)(2_2, 1_0), (2_3, 3_1)(3_2, 0_0), (3_3, 2_1)\}$$

It is not difficult to verify that G_2 is a 4CS. Later, we will find that G_2 forms a square array architecture. The diagram of G_2 can be represented by Figure 7.2.

7.2 Characteristics of 4 Connected Topology

7.2.1 4CS naturally supports parallel ATPG

As discussed before, to generate a test pattern for a given fault, we first check whether a trial test pattern can detect the fault. If it can detect the fault, a test pattern is found. If it is shown that the test pattern cannot detect the fault, it will lose its status as a potential test for the fault and be abandoned by our ATPG parallel algorithms. If we cannot decide whether this trial test pattern can or cannot detect the given fault, it becomes a potential test pattern. We will expand this potential test pattern to two new trial test patterns by guessing the best potential primary input and setting the input 0 and 1, respectively, as discussed in the previous chapter. Thus, during one virtual 2 phase clock period, each processing element may accept one trial test pattern and generate two trial test patterns. Two output ports of a processing element provide the throughways for these patterns. In this model, these two potential test patterns can flow out of the node without any delay or traffic jam. Since a traffic jam is avoided, no situation arises where some processors cannot send potential test patterns out to

other idle processors because the bus which delivers these patterns is busy. A network with this property is said to be *saturation-free*.

Here, we note another characteristic. Each processor has two input ports, but in each cycle, a node can only process one input. Therefore, the other input has to be stored in somewhere. Every node has its own memory to store these unprocessed inputs. This memory provides a *self-balance* characteristic. In our algorithm, a trial test pattern may be aborted because it is impossible to generate a test pattern after applying the checking algorithm. Therefore, there is no potential test pattern sent out. This may make some of the PEs in some subtree idle. The topology itself has several ways to make them busy again: it can

1. fetch trial test patterns from the memory of the PE.

2. receive trial test patterns from PEs connected to its input ports.

These characteristics are very useful. All our designed architectures are 4 connected structures. Therefore, they have the properties of saturation-free and self-balance. These properties will be shown in later chapters.

7.2.2 Isomorphic 4CS systems

In drawing a diagram for 4 connected multi-processor systems, we have complete freedom to draw them in arbitrary positions or shapes. There are no restrictions on the size of the vertices or on the length or even the shape of the edges. These drawings, although constrained by the connectivity of the nodes, are very much free-form. We are also free to choose an entirely different representation for the graph. But this freedom presents us with some other difficulties. If a graph

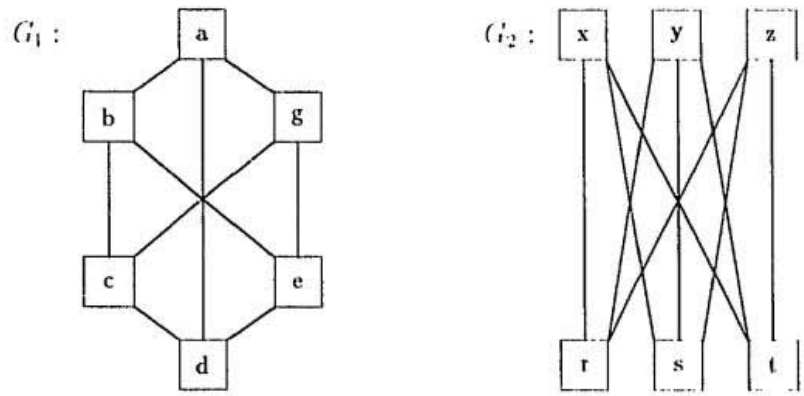


Figure 7.3: Two Isomorphic Graphs

is presented in different ways, how can we determine if the presentations really represent the same graph, or the same topology in our 4 connected multi-processor system?

Mathematicians use the term *isomorphism* to mean the "fundamental equality" of two objects or systems. That is, the objects really have the same mathematical structure, and only nonessential features like object names might be different. For graphs, "fundamentally equal" means the graphs have essentially the same adjacencies and nonadjacencies. To formalize this concept further, we use the following definition to define when two graphs G_1 and G_2 are *isomorphic*:

Definition Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic when there is a bijection $\alpha : V_1 \rightarrow V_2$, such that $(\alpha(x), \alpha(y)) \in E_2$ if and only if $(x, y) \in E_1$. The bijection α is said to be an isomorphism.

For example, consider the two graphs shown in Figure 7.3. An isomorphism

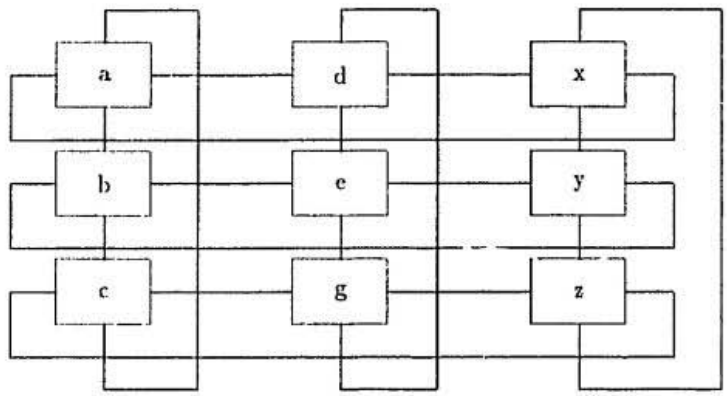


Figure 7.4: A Non-planar 4-connected Graph

between G_1 and $G2$ is determined by the function $\alpha : V(G_1) \longrightarrow V(G_2)$ where

$$\alpha(a) = x, \qquad \alpha(b) = r, \qquad \alpha(c) = y,$$

$$\alpha(d) = s, \qquad \alpha(e) = z, \qquad \alpha(g) = t.$$

It is clear that α is a one-to-one and onto function. An isomorphism from G_2 to G_1 is given by α^{-1}, the inverse of α.

Unfortunately, the graph isomorphism problem, that is: *given two graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$, *are* G_1 *and* G_2 *isomorphic, i.e., is there a one-to-one and onto function* $f : V_1 \longrightarrow V_2$ *such that* $\{u, v\} \in E_1$ *if and only if* $\{f(u), f(v)\} \in E_2$? is an open problem[27]. It means that, so far, there is no efficient algorithm to solve the graph isomorphism problem. This problem remains open even if G_1 and G_2 are restricted to regular graphs, bipartite graphs, line graphs, comparability graphs, chordal graphs, or undirected path graphs[27]. However, it is solvable in polynomial time for planar graphs[29]. Here it is worth

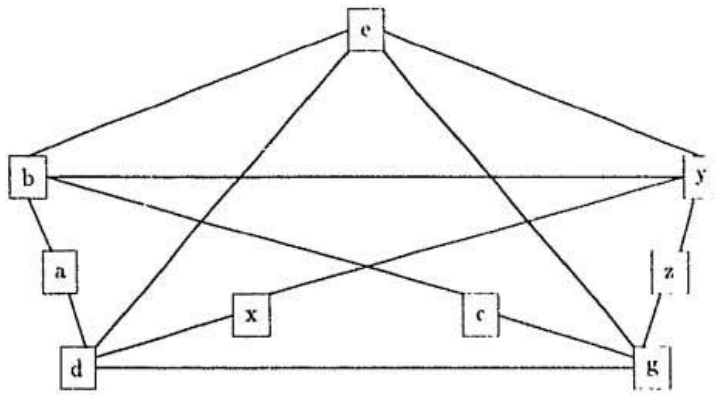


Figure 7.5: A Subdivision of K_5

mentioning that **NOT** all 4 connected systems have planar structure. For example, Figure 7.4 shows an array structured 4 connected multi-processor system. The subgraph of this structure can be drawn as a subdivision of K_5, as shown in Figure 7.5. According to Kuratowski's Theorem[28], *a graph G is planar if and only if G contains no subgraph homeomorphic with K_5 or $K_{3,3}$* , this array structured 4 connected multi-processor is not planar. Kuratowski's theorem also implies that a planar graph and a unplanar graph are not isomorphic.

Therefore, so far, we have no efficient algorithm to detect whether two 4 connected multi-processor systems are isomorphic or not. When we design a parallel system, we should pay attention to this problem so as to avoid designing isomorphic configurations.

The concept of 4 connected structure and some characteristics have been discussed. Their properties will be helpful in the design of special multi-processor

systems to solve the automatic test pattern generation problem. From the next chapter, we will begin to discuss several special structures designed for automatic test pattern generation.

Chapter 8

ATPG Using MCBTA

In this chapter, we first introduce the Modified Complete Binary Tree Architecture and its parallel algorithm. The data from experiments with this architecture, showing their speedup, are presented. After that, an autonomous MCBTA, and the results of some experiments with it are also discussed.

8.1 MCBTA Architecture and Parallel Algorithm

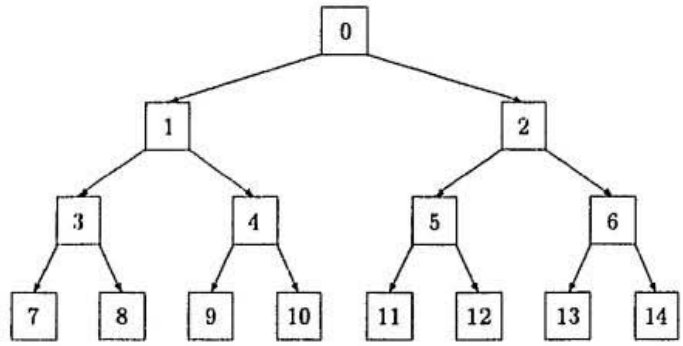
8.1.1 Architecture and Algorithm

CBTA and Parallel Algorithm

In a complete binary tree with height k, every internal node has exactly two children, a left subtree and a right subtree. The distance from the root node to any leaf node is k. Figure 8.1 shows a 3 *complete binary tree*, denoted 3 CBT.

We construct a processing architecture, called the complete binary tree architecture(CBTA). Suppose every node in k CBT is a processing element (PE) which is also often called *processor*, every edge is a connection between two processors, one output line to the input of another. Every PE has two output lines and

Figure 8.1: Complete Binary Tree Architecture



one input line, which are denoted as O_0, O_1 and I, respectively. All PEs work synchronously. Data from each PE are sent to both its left and right subtrees. Parallel algorithm 1 can generate all possible k bit values in k CBTA within k steps.

Parallel Algorithm 1:

set root PE store vector $X_1 X_2 \cdots X_k$;

set all other non_root PEs to contain no vectors;

EVERY PE DOES SIMULTANEOUSLY

EVERY STEP DO

input vector = vector from I;

if (input vector is non_data) do nothing in this step;

/* otherwise input vector has the form $a_1 \cdots a_i X_{i+1} \cdots X_k$ */

if ($i == k$) $a_1 \cdots a_i$ is one possible value;

else

send $a_1 \cdots a_i 0 X_{i+2} \cdots X_k$ to PE connected to O_0;

send $a_1 \cdots a_i 1 X_{i+2} \cdots X_k$ to PE connected to O_1;

It is obvious that this parallel algorithm which runs on k CBTA can generate all possible k bit values with k steps since each step determines one bit.

Modified CBT (MCBTA) and Parallel Algorithm

It is impractical for a circuit with many inputs, say m inputs, to use m CBTA to generate all possible values since $2^{m+1} - 1$ PEs have to be used. It is necessary to use a fixed height k CBTA to generate all possible values for m bits ($m > k$). We modify CBTA to get a modified CBTA (MCBTA) with these properties. First, every PE is expanded to have a local memory and two inputs, I_0 and I_1. Second, two output lines of every leaf processor are connected to one internal processor and itself, we designate such output lines as feedback lines or feedback connections. For example, *leaf node 11* in Figure 8.2 is connected to *internal node 2* and itself. In a later section, an algorithm is discussed to generate such connected MCBTA structures. Figure 8.2 shows the MCBTA for the CBTA shown in Figure 8.1. This network is 4-connected, and is obviously planar, making it an excellent candidate for VLSI layout. Figure 8.3 is its layout using the H-tree algorithm[6].

To generate test patterns using this architecture, parallel algorithm 1 must be modified slightly; we designate this as parallel algorithm 2.

Parallel Algorithm 2:

set root PE store vector $X_1 X_2 \cdots X_m$;

set other non_root PEs empty;

EVERY PE DOES SIMULTANEOUSLY

Figure 8.2: Modified Complete Binary Tree Architecture

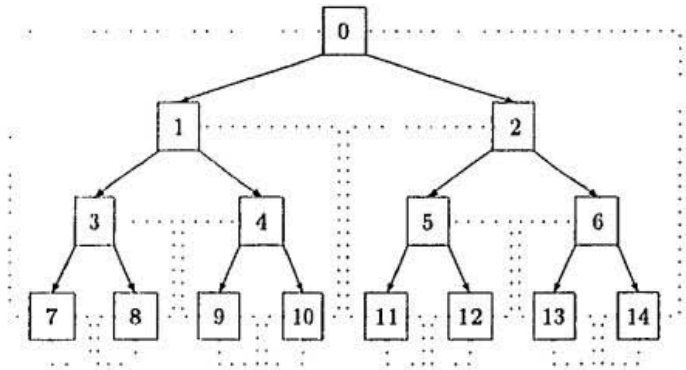
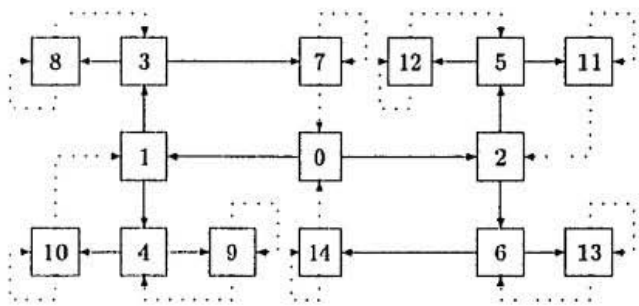


Figure 8.3: Layout of MCBTA using H-tree



EVERY STEP DO

receive vectors from I_0 and I_1;

put them into memory according to some strategy[1];

input vector = get one vector from its memory;

if (input vector is non_data) do nothing in this step;

/* otherwise input vector has the format $a_1 \cdots a_i X_{i+1} \cdots X_m$ */

if ($i == m$) $a_1 \cdots a_i$ is one possible value

else

send $a_1 \cdots a_i 0 X_{i+2} \cdots X_m$ to the processor connected to O_0

send $a_1 \cdots a_i 1 X_{i+2} \cdots X_m$ to the processor connected to O_1

One of the drawbacks of using the MCBTA configuration is that the output can be a bottleneck for the whole system. That is, when several PEs generate their possible values at the same step, how can these data be output? In general, one PE is specified to take charge of communicating with the outside. Therefore, all results have to be sent to this PE. Then, this PE will send them outside step by step. This PE forms a bottle-neck of the system. Fortunately, for automatic test pattern generation, this is not the case since one suitable pattern is sufficient for ATPG.

[1]For every PE, some strategy has to be adopted to store data. Since there are two input lines, I_0 and I_1, two inputs may possibly be fed at the same time. Since at each step, every PE can only process one input vector, one vector has to be stored in memory to be processed later. The strategy used to store these vectors decides the size of local memory. If a queue structure is used, MCBTA is similar to using a breadth-first algorithm to generate all possible values. Consequently, a large (possibly huge) amount of memory is needed for every PE to store potential values (test vectors with undetermined values, X). If a stack structure is used, MCBTA is similar to using the depth-first algorithm, local memory can be greatly saved. From this consideration, it is most preferable to adopt the stack structure.

An Algorithm to Generate MCBTA Structure

To design an algorithm to generate the topology of a MCBTA, each node needs an identifier, called *label*. Each node is labeled based on following rules:

1. Each node is located on a special layer l which is defined as the distance from the root node to the node.

2. Root node is labeled as 0.

3. The left-most node on layer l is labeled as $i+1$ if the right-most node on layer $l-1$ has label i.

4. Each node except the left-most node on layer l is labeled as $i+1$ if its left node has label i.

According to these rules and characteristics of modified complete binary tree, a node i located at jth position of l layer has the relationship,

$$i = 2^l - 1 + j \qquad (j = 0, 1, \cdots, 2^l - 1)$$

In order to connect these nodes, following rules can be used:

1. If node $i = 2^l - 1 + j$ is not a leaf node, its two outputs are connected to node k and node $k+1$, where $k = 2^{l+1} - 1 + 2j$.

2. The left-most leaf node and the right-most leaf node have labels $2^{L-1}-1$ and 2^L-2, where L is the number of layers in the MCBTA. Both of them connect one of their output ports to one of their own input ports, and connect their another output port to the root node.

3. For any leaf node which is neither the left-most nor the right-most leaf node, one of its output ports is connected to one of its own input ports. Its another output port is connected to one node according to the rule: For $l = 1, 2, \cdots, L-1$ and $j = 0, 2, 4, \cdots, 2^l - 2$, node $2^l - 1 + j$ is fed by leaf node $2^{L-1} + 2^{L-l-1} + (j-1)2^{L-l}$, and node $2^l - 1 + j + 1$ is fed by leaf node $2^{L-1} + 2^{L-l-1} + (j-1)2^{L-l} + 1$.

ATPG Algorithm using MCBTA

In this section, MCBTA is used to generate a test pattern. We modify algorithm 2 to parallel algorithm 3. The time complexity of each step depends on the algorithm for circuit simulation, the algorithm for checking whether the tried test pattern can detect a given fault, and the algorithm for expanding a potential test pattern. As discussed in chapter 6, all of these algorithms have time complexity of $O(N)$, where N is the number of gates in a circuit. Hence, the time complexity of this parallel algorithm in each step is $O(N)$.

Parallel Algorithm 3:

set root PE store vector $X_1 X_2 \cdots X_m$ with undecided-flag;

set other non_root PEs empty;

EVERY PE DOES SIMULTANEOUSLY

EVERY STEP DO

receive vectors from I_0 and I_1;

if (any of them has a detected-flag)

// assume this vector has the form $a_1 \cdots a_i X_{i+1} \cdots X_m$

set detected-flag; // the test pattern is already found.

send $a_1 \cdots a_i X_{i+1} \cdots X_m$ to processors connected by O_0 and O_1

else

put them into memory according to some strategy[2];

input vector = get one vector from its memory;

if (input vector is non_data) do nothing in this step; /* idle status */

/* otherwise input vector has the form $a_1 \cdots a_i X_{i+1} \cdots X_m$ */

simulate the faulty circuit with the input vector[3];

check whether the fault can be detected by $a_1 \cdots a_i X_{i+1} \cdots X_m$ [4];

switch(result of checking)

case DETECTED:

set detected-flag;

send $a_1 \cdots a_i X_{i+1} \cdots X_m$ to processors connected by O_0 and O_1

break;

case UNDECIDABLE:

if ($i == m$) send nothing to O_0 and O_1 [5];

else

use heuristics to select the primary input

with the most potential ability [6];

/* suppose X_{i+1} is this primary input */

send $a_1 \cdots a_i 0 X_{i+2} \cdots X_m$ to the processor connected by O_0;

send $a_1 \cdots a_i 1 X_{i+2} \cdots X_m$ to the processor connected by O_1;

[2]Refer to the explanation in the parallel algorithm 2.

[3]Refer to the simulation algorithm in chapter 6

[4]Refer to the section about checking trial test patterns in chapter 6

[5]In general, there should not be the UNDECIDABLE case since all primary inputs have a specified value 0 or 1. For the program completeness, it is still considered a possibility.

[6]Refer to the expanding algorithm in chapter 6.

break;

case CAN_NOT_DETECTED:

send nothing to O_0 and O_1;

break;

In order to output the result of ATPG, one output port should be defined for the whole architecture, which can report whether the MCBTA has generated a test pattern, whether it is generating a test pattern, or whether it has found that the fault is undetectable. We accomplish this by expanding the function of the root node, denoted by I/O PE. I/O PE has the following functions; it:

1. receives commands from outside the network.

2. sends commands to internal PEs.

3. outputs computing results to outside.

These duties make it a special PE.

MCBTA is a 4 connected structure. Following from the discussion in chapter 7, this structure has the self-balance property and is saturation-free.

We can also claim that MCBTA has $O(\log^2 n)$ output time delay.

Suppose the number of processors is n, then the height of the MCBTA is $\log n$. If one PE finds that the fault can be detected, it will send the result to the root node, which can then output the result. In the worst case, $O(\log^2 n)$ steps are needed to propagate this test pattern. This can be shown as follows:

Before the proof, we introduce the concept of *level*. Every processor can be labeled with a value which is called its *level*. This value is defined by the following rules:

1. The root processor has *level 0*.

2. A processor has *level l* + 1, if its parent processor in the complete binary tree has *level l*.

The longest path from a processor to the root processor determines the output time delay. MCBTA itself has one important characteristic: every path from an inner processor to the root processor contains one of two feedback connecting lines:

1. the left-most leaf processor $\longrightarrow$ the root processor

2. the right-most leaf processor $\longrightarrow$ the root processor

The longest path in the architecture begins at a leaf processor. Each time to reach a lower level inner processor, h steps have to be taken, where h is the height of this subtree rooted from this lower level inner processor. For example, in Figure 8.2, one of the longest paths is

$$\underbrace{PE_9 \longrightarrow PE_4}_{1} \underbrace{\longrightarrow PE_{10} \longrightarrow PE_1}_{2} \underbrace{\longrightarrow PE_3 \longrightarrow PE_7 \longrightarrow PE_0}_{3}$$

The length of this path is

$$6 = 1 + 2 + 3$$

from PE_9 to PE_4, 1 step; from PE_4 to PE_1, 2 steps; from PE_1 to PE_0, 3 steps.

In general, the length of the longest path in a MCBTA from one processor to the root processor has

$$\sum_{h=1}^{k} h = \frac{1}{2}k(k-1) = O(k^2) = O(\log^2 n)$$

edges. Each edge causes one time delay. Here, it is worth mentioning that the parallel algorithm 3 first checks whether patterns in two input ports contain a test pattern. If there is a test pattern, it will be sent directly to two output ports O_0 and O_1. This guarantees that each edge causes only one time delay and the test pattern can reach at the I/O PE as quickly as possible. Therefore, MCBTA has $O(\log^2 n)$ output time delay.

8.1.2 Empirical Results and Analysis

To evaluate the performance of MCBTA, every processor uses a parallel algorithm with the same heuristics as discussed before. At first, MCBTA contains only one processor. The speed of this MCBTA forms a basis for speed comparison. As more and more processors are put into MCBTA, the ratio of this basis speed and the current MCBTA's speed determines the speedup of the current MCBTA.

MCBTA is a complete binary tree. The number of processors in MCBTA is $2^{k+1} - 1$, where k is the height of tree. For $k = 0$, 1, 2, 3, and 4, we can construct 5 MCBTA multi-processor systems.

Five very hard-to-detect faults in the circuit C432 were submitted to each system. Empirical results were gained and are shown in the Figure 8.4 and Figure 8.5.

Four of these very hard-to-detect faults are proved to be redundant by these multi-processor systems[7]. Their speedup curves are shown in Figure 8.4.

One of the 5 very hard-to-detect faults was found to be detectable, and its test pattern was generated. Figure 8.5 shows the speedup curve for the fault. It

[7]the method to prove the redundancy was discussed in Chapter 6, section 6.6

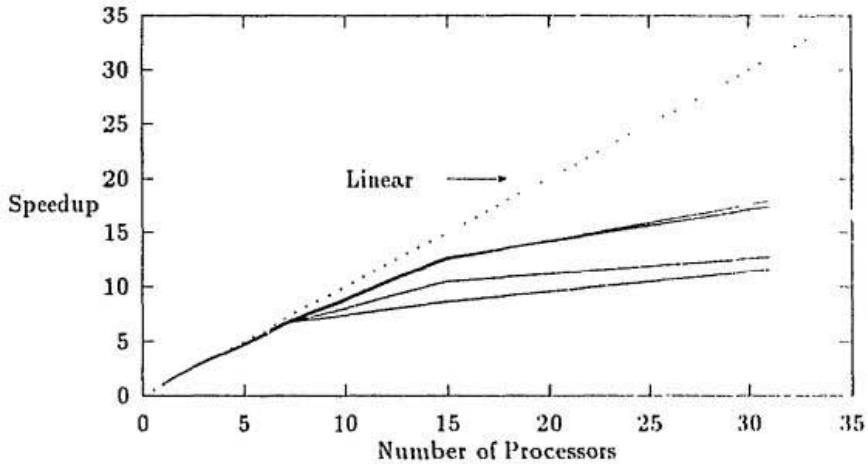


Figure 8.4: Speedup for 4 Redundant Faults in MCBTA

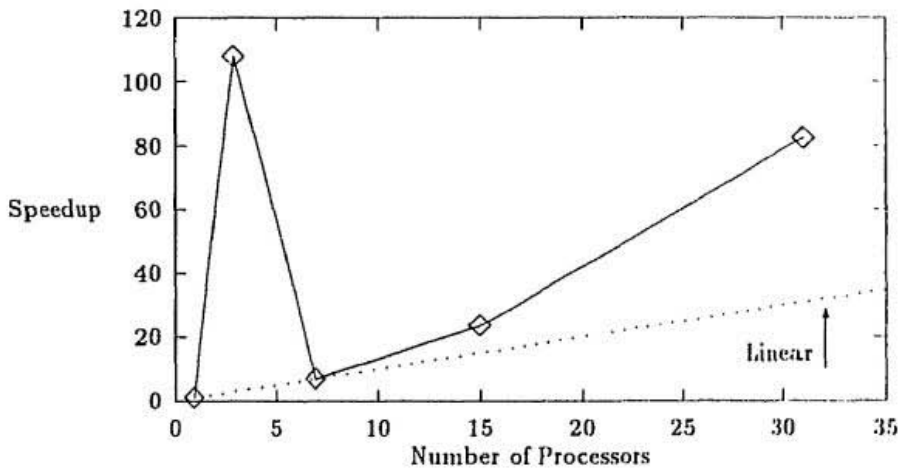


Figure 8.5: Speedup for an Irredundant Hard-to-detect Fault in MCBTA

is very interesting to note that there are super-linear speedups, that is, speedup is greater than the number of processors, when the number of processors is 3, 15, and 31, respectively.

8.2 Autonomous MCBTA Architecture

All processors in MCBTA use the same heuristics to guess the "best" undecided input so as to find a test as early as possible. We call such a MCBTA a *pure* MCBTA. Testability is one of the most widely used heuristics since it is considered to be an inherent property of a circuit, and is determined entirely by its structure [17]. This allows estimation of circuit testability before test generation. Because of the approximate nature of the analysis, most testability analyses results have poor accuracy.

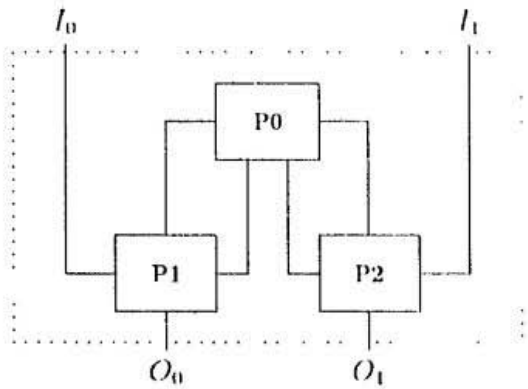
If some complementary heuristics are also used by some processors, a mixed heuristic MCBTA can be formed. It is quite possible that a test pattern can be found much faster than a pure MCBTA.

8.2.1 Architecture

If several processors form a pure MCBTA, one I_0, one I_1, one O_0, and one O_1 are opened to outside. We call such a MCBTA a pure MCBTA module, or autonomous MCBTA module. Figure 8.6 shows one pure MCBTA module which consists of 3 processors.

It is called an autonomous module because it has two characteristics. First, one module has only one policy to select the most potential undecided input. Second, there are cycles within the module. This property has the potential ability to find

Figure 8.6: A Pure MCBTA



a test pattern early, or to get a stop conclusion quickly, in case the selecting policy is the most suitable one for a given fault.

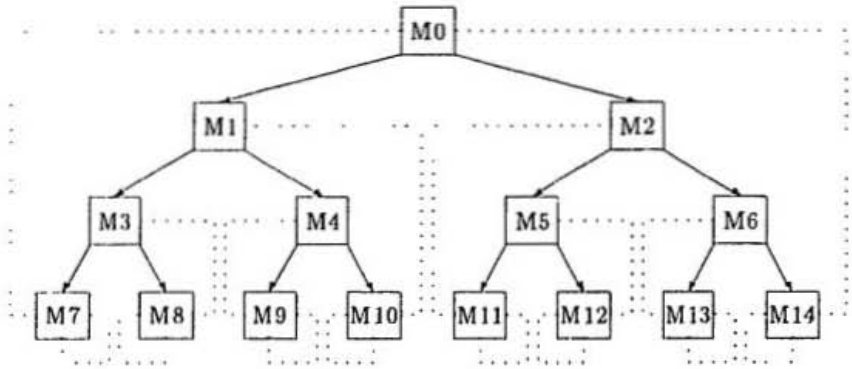
Figure 8.7 is an autonomous MCBTA (AMCBTA). From the overview, it has the same topology as a pure MCBTA. The difference is that it consists of autonomous modules instead of processors and each module uses one of the two heuristics as discussed in section 6.5.

8.2.2 A Parallel Algorithm

The parallel algorithm for AMCBTA is exactly the same as the parallel algorithm for MCBTA, as discussed in last section, *parallel algorithm 3* in section 8.1.1.

The output time delay of AMCBTA is $O(\log^2 n)$. Suppose n is the number of processors in the system. Since every module contains three processors, we can use a k AMCBTA ($k = \log \frac{n}{3}$) to generate test patterns. If one PE finds that the fault can be detected, it will send the result to the host module, which can

Figure 8.7: Autonomous MCBTA



inform the outside. In the worst case, $O(k^2)$ steps are needed to propagate this test pattern. The explanation is similar to that for MCBTA discussed previously.

8.2.3 Empirical Results and Analysis

To evaluate the performance of an autonomous MCBTA, we use the same simulating method as for MCBTA. At the first, AMCBTA contains only one module. The speed of MCBTA with only one processor is still a basis speed. As more and more modules are put into AMCBTA, the ratio of basis speed to the current AMCBTA's speed determines the speedup of the current AMCBTA.

AMCBTA is a complete binary tree. The number of processors in MCBTA is $3 \times 2^{k+1} - 1$, where k is the height of tree. For $k = 0$, 1, 2, and 3, we can construct 4 AMCBTA multi-processor systems, which contain 3, 9, 21, and 45 processors, respectively.

Five very hard-to-detect faults in the circuit C432 were submitted to each

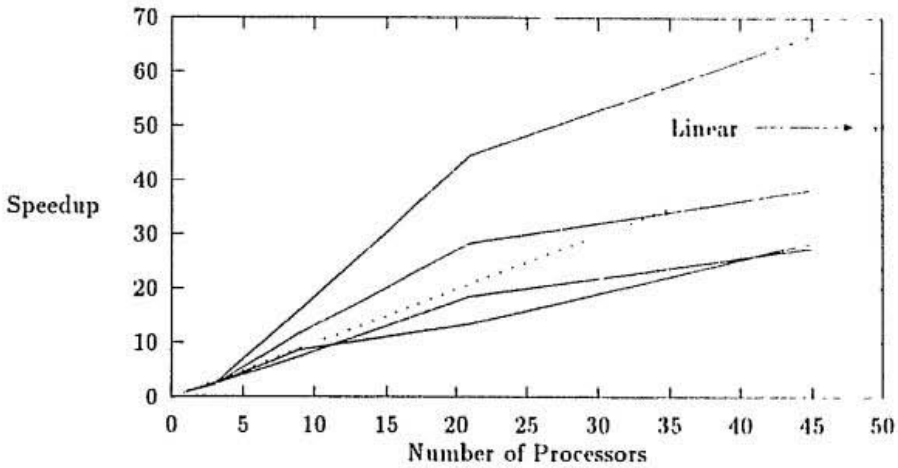


Figure 8.8: Speedup for 4 Redundant Faults in AMCBTA

system. Empirical results were gained and are shown in the Figure 8.8, 8.9, and 8.10.

Again, four of these very hard-to-detect faults are proved to be redundant by these multi-processor systems[8]. Their speedup curves are shown in Figure 8.8. The anomalous phenomenon[9] of the automatic test pattern generation problem appears on only one fault.

One of the 5 very hard-to-detect faults is found to be detectable, and its test pattern is generated. Figure 8.9 and 8.10 are the speedup curves for the fault. It is amazing that there are super-linear speedups too, when the number of processors is 9, 21, and 45, respectively. And they are even better than the results in Figure 8.5. Since complementary heuristics[10] are used, they eliminate the fruitless search space quickly. The redundancy is proved much more quickly than in MCBTA.

[8]refer to Chapter 6, section 6.6
[9]refer to Chapter 5.
[10]refer to Chapter 6, section 6.4

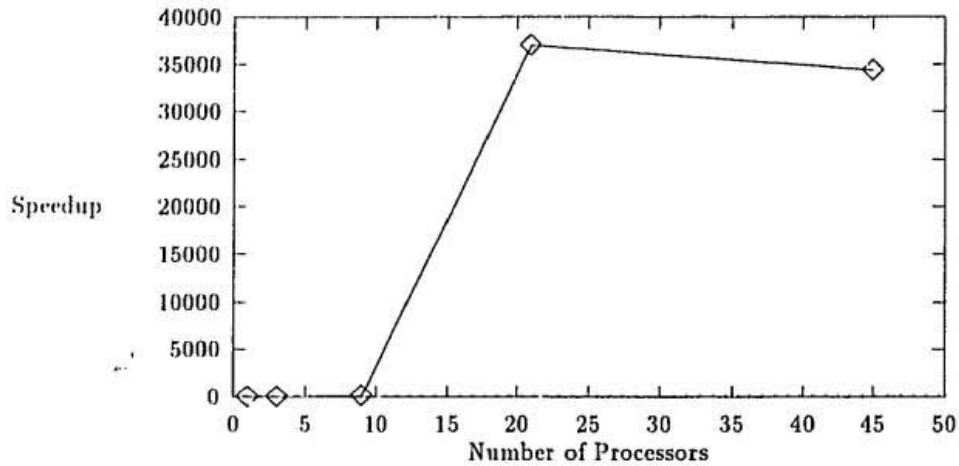


Figure 8.9: Speedup for an Irredundant Hard-to-detect Fault in AMCBTA

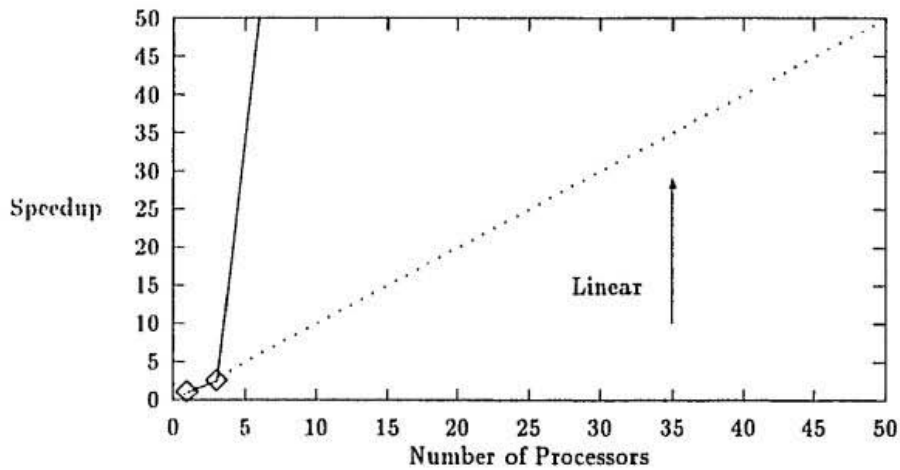


Figure 8.10: Scaled Diagram for Figure 8.9

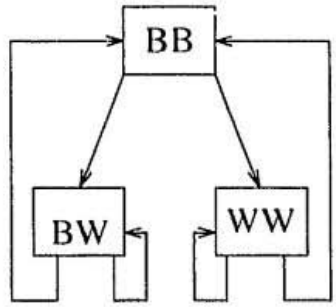


Figure 8.11: Experiment With 1 Module

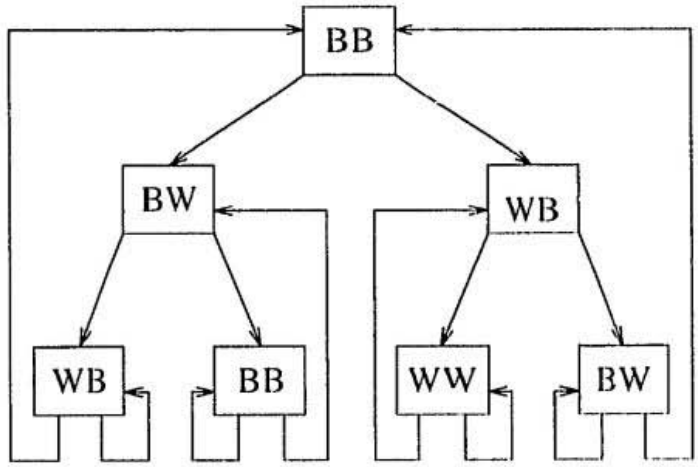


Figure 8.12: Experiment With 7 Modules

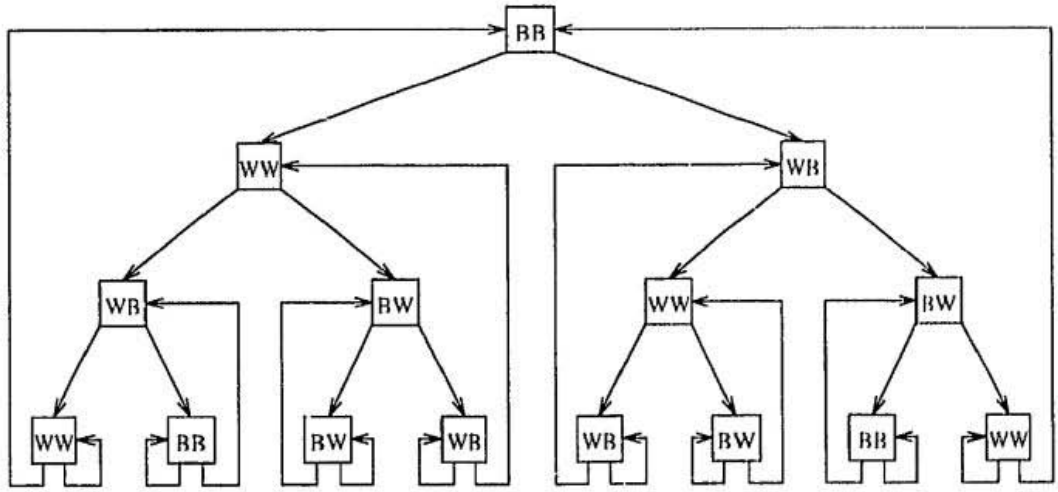


Figure 8.13: Experiment With 15 Modules

Figure 8.11, 8.12 and 8.13 show the mapping between modules and heuristics used in our experiments. There four notations are defined:

BB -- Best controllability and Best observability

BW -- Best controllability and Worst observability

WB -- Worst controllability and Best observability

WW -- Worst controllability and Worst observability

Each module is assigned one heuristic at random.

Comparing experimental results between MCBTA and AMCBTA, we can conclude that AMCBTA has better linear speedup than MCBTA, and it also has greater super-linear speedup than MCBTA has. Therefore, at least for this circuit, AMCBTA is much more attractive than MCBTA.

Chapter 9

ATPG Using Square Array

This chapter introduces another system: square array architecture. Experimental results will also be presented for this architecture.

9.1 Square Array and Its Parallel Algorithm

9.1.1 Square Array Architecture

A square array system consists of n^2 processors, called n^2 SQARRAY. In an n^2 SQARRAY, each row or column contains n processors. Each processor has two input ports (I_0 and I_1), and two output ports (O_0 and O_1).

Figure 9.1 shows the symbol for one processor. Every output O_0 is connected to its right neighbor's input port I_0. The right-most processor will connect its O_0 to the I_0 of the left-most processor on the same row. Every output O_1 is connected to the input port I_1 of its neighbor below. The lowest processor in a column will connect its output O_1 to the input I_1 of the processor at the top of the column. Figure 9.2 is a square array system, which consists of 3^2 processors.

Here, we may ask whether SQARRAY and MCBTA are isomorphic. We say, they are not. As we discussed in chapter 7 section 7.2.2, a SQARRAY is a non-

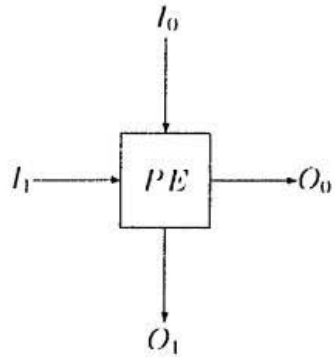


Figure 9.1: The Symbol for a Processor in SQARRAY

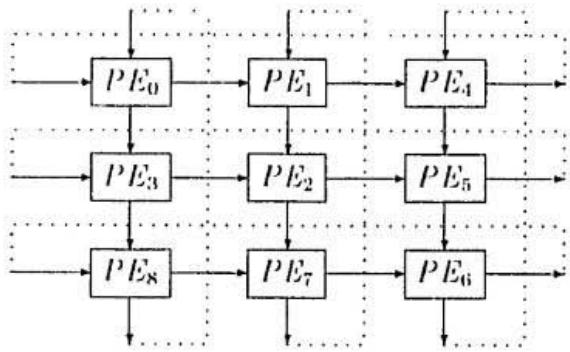


Figure 9.2: SQARRAY with 9 Processors

planar 4-connected graph. It is clear that MCBTA is planar since it can be easily floorplaned on a plane without any intersection. A non-planar graph and a planar graph are not isomorphic[1].

9.1.2 Completeness of Square Array

Theorem Given any integer n, n^2 square array system can generate all 2^k possible values, where k is an integer.

Proof From the topology of a square array system, if there is a binary value $b_{k-1}, \cdots, b_{k-i}, X_{k-i-1}, \cdots, X_0$ through a horizontal connecting line from processor PE_i to PE_j, $b_{k-1}, \cdots, b_{k-i}, 0, X_{k-i-2}, \cdots, X_0$ can be generated at the PE_j. If the connecting line is a vertical line, $b_{k-1}, \cdots, b_{k-i}, 1, X_{k-i-2}, \cdots, X_0$ can be generated at the PE_j.

For convenience, in the same row, we call the left-most processor as the neighbor of the right-most processor. In the same column, the top-most processor is considered as the neighbor of the lowest processor. Each processor is therefore connected directly to all of it nearest neighbors.

For any v, $0 \leq v < 2^k$, v can be represented by a k bit binary number v_{k-1}, $v_{k-2}, \cdots, v_0$. Then, we can find a path from PE_0 to PE_v ($0 \leq v < n^2$). We say that $PE_0, PE_{i_{k-1}}, PE_{i_{k-2}}, \cdots, PE_{i_0}$ is a path which generates $v_{k-1}, v_{k-2}, \cdots, v_0$, where $PE_{i_{k-1}}$ is the PE to the right of PE_0 if $v_{k-1} = 0$. Otherwise, $PE_{i_{k-1}}$ is the PE below PE_0 if $v_{k-1} = 1$. For $PE_{i_{k-j}}$, $(j = 2, \cdots, k)$, if $v_{k-j} = 0$, $PE_{i_{k-j}}$ is the PE to the right of $PE_{i_{k-j+1}}$; if $v_{k-j} = 1$, $PE_{i_{k-j}}$ is the PE below $PE_{i_{k-j+1}}$.

[1] refer to chapter 7, section 7.2.2

For example, in Figure 9.2, PE_0, PE_1, PE_2, PE_7, PE_6 is a path from PE_0 to PE_6, which generates 0110. □

This theorem guarantees that SQARRAY will find a test pattern for a given fault if it exists.

9.1.3 A Parallel Algorithm

In this section, SQARRAY uses the same parallel algorithm as MCBTA and AM-CBTA, as discussed in chapter 8. The time complexity of this parallel algorithm in one step is $O(N)$, since the checking algorithm, the simulation algorithm, and the expanding algorithm all have $O(N)$ time complexity.

In SQARRAY, the output time delay is $O(\sqrt{n})$. Suppose n processors are used, we construct a $(\sqrt{n})^2$ SQARRAY. If one PE finds that the fault can be detected, it will send the result to the host processor, which can tell the outside. If any PE finds a test pattern, this pattern can be propagated to the left-most processor within $\sqrt{n}$ steps since there is a path to connect all processors on the same row. Similarly, this pattern can also reach the top-most processor within $\sqrt{n}$ steps. Therefore, within $\sqrt{n} + \sqrt{n}$ time step, the test pattern can arrive at the corner of left and top, which is the processor in charge of communicating with outside. Hence, we say that $O(\sqrt{n})$ steps are needed to propagate this test pattern.

9.1.4 Empirical Results and Analysis

To evaluate the performance of a square array system (SQARRAY), First, we let the system contain only one processor. The speed of this SQARRAY forms a

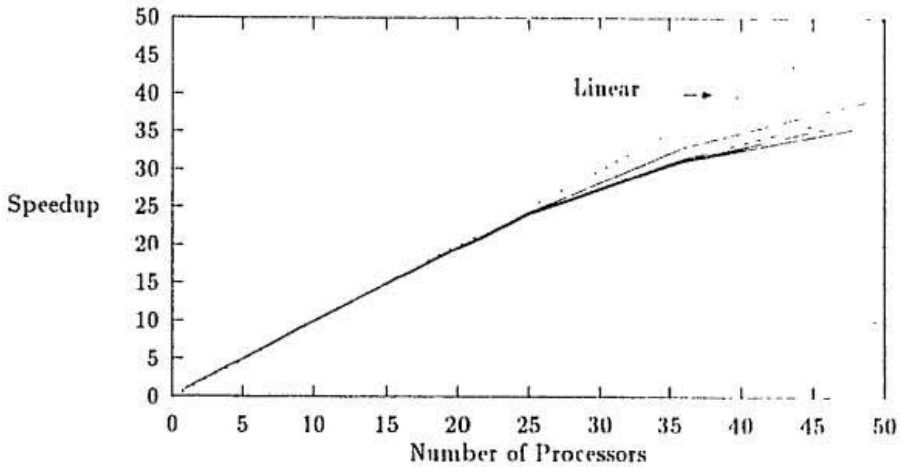


Figure 9.3: Speedup for 4 Redundant Faults in SQARRAY

basis speed. As more and more processors are put into SQARRAY, the ratio of basis speed and current SQARRAY's speed determines the speedup of the current SQARRAY.

When the size of SQARRAY was assigned as $1(1^2)$, $4(2^2)$, $9(132)$, $16(4^2)$, $25(5^2)$, $36(6^2)$, and $49(7^2)$, we got the speedup curves shown in Figure 9.3.

Again, five very hard-to-detect faults in the circuit C432 were submitted to each system. Empirical results were gained and are shown in the following curves.

Again, four of these very hard-to-detect faults were proved to be redundant by these multi-processor systems. Their speedup curves are shown in Figure 9.3. They are quite close to linear speedup.

One of the 5 very hard-to-detect faults is found to be detectable, and its test pattern was generated. Figure 9.4 is the speedup curve for the fault. There are super-linear speedups when the number of processors is 4, 16, 36, and 49, respectively. Compared with Figure 8.5 and Figure 8.9, Figure 9.7 is better than

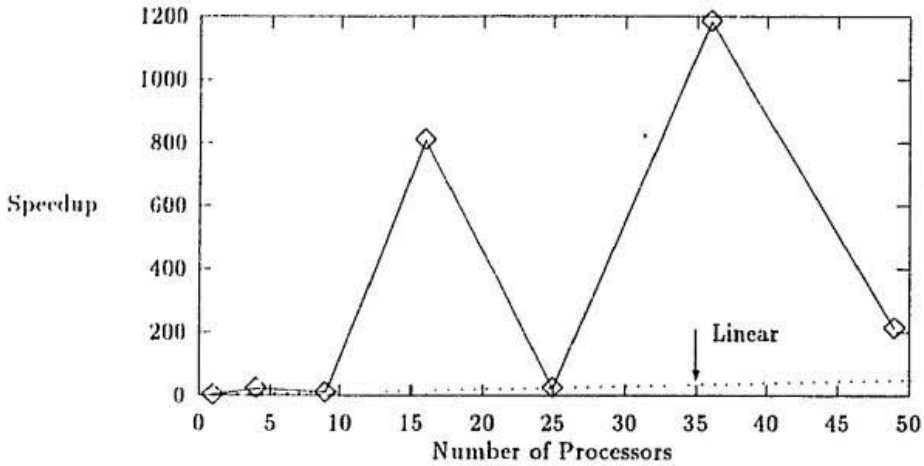


Figure 9.4: Speedup for an Irredundant Fault in SQARRAY

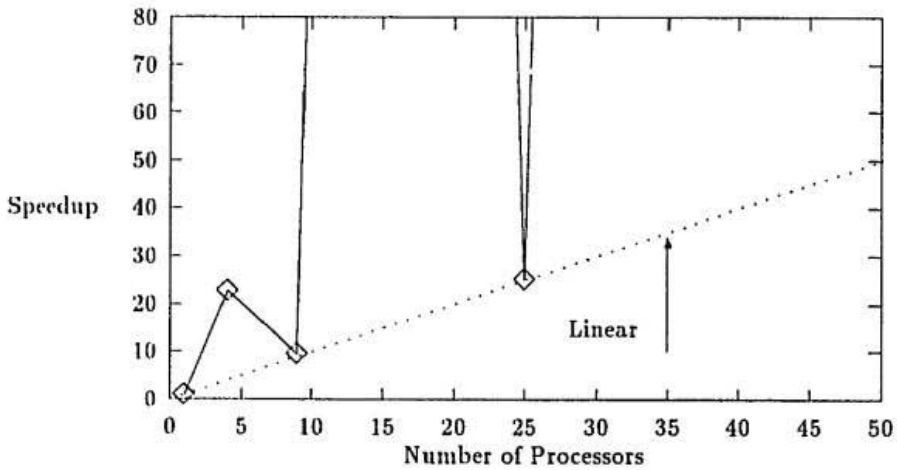


Figure 9.5: Scaled Speedup for the Irredundant Fault in SQARRAY

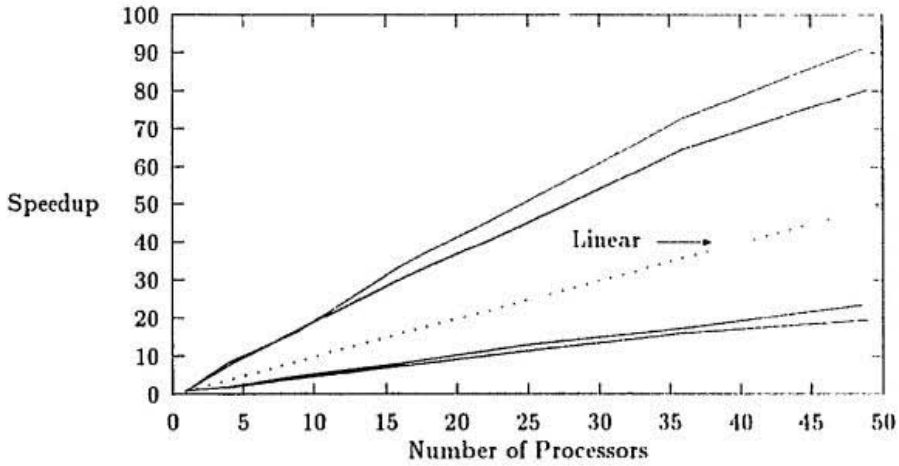


Figure 9.6: Speedup in Complementary SQARRAY (4 redundant faults)

the curve in Figure 8.5 but is inferior to the curve in Figure 8.9.

If we compare the results with MCBTA, we can find that SQARRAY has better speedup than MCBTA, and it also has superior super-linear speedup relative to MCBTA, at least.

For this example, if we use complementary heuristics for each processor, can the performance be improved?

Curves in Figure 9.6 are obtained from the simulation.

These curves tell us that complementary heuristics can speed up the processing for some faults, and they can also produce extreme super-linear speedup relative to the use of a single heuristic.

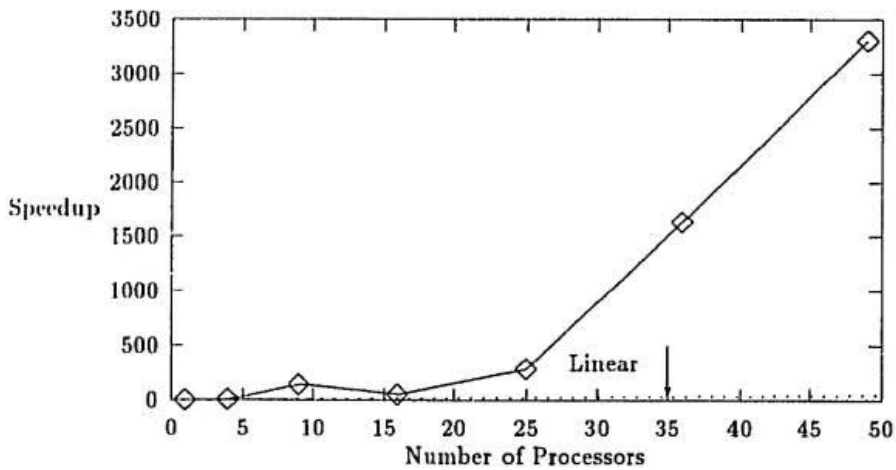


Figure 9.7: Speedup in Complementary SQARRAY (an irredundant fault)

PartIV

Conclusion and Discussion

A major barrier to the full exploitation of the capabilities offered by VLSI is the problem of the increased cost of testing the complex devices immediately after fabrication. The need to test devices results from imperfections in the fabrication process producing a wide range of defects in the devices; for example, pin-holes in the gate oxide, shorted or open interconnect lines (polysilicon, diffusion and metal), contact hole defects, crystalline defects on the wafer, etc. There may also be some design faults, such as a gate output having insufficient drive capability for its output capacitance, which may not be identified by the simulator, unless a post layout simulation is performed; although simulation may have been used extensively many design faults may go undetected since simulation is an incomplete process based on an abstract model.

Attempts to reduce the costs of testing have been made by developing more sophisticated gate level test generation algorithms, by performing test generation at higher levels of abstraction, by exploring parallel processing techniques, etc. The results of these work still show that developments in the solution of the testing problem do not keep up with the pace of the development of VLSI devices. This implies that much more work should be done in this field.

One method is to study the types of architecture which can powerfully support ATPG algorithms, or study what kind of connection among processors can most efficiently speed up automatic test pattern generation.

In this report, we proposed three interconnection methods to speed up test pattern generation. The experiment results show that for a redundant fault, the square array structure has more linear speedup than MCBTA if the same heuristics are used. For an irredundant fault, SQARRAY more likely reaches super-linear

speedup than MCBTA does. If autonomous MCBTA is used, even for redundant faults, super-linear speedup can occur. For an irredundant fault, the speedup reaches incredible values. For example, when the number of processors is 21, the speedup is greater than 3500, a factor of about 170. These results are even better than SQARRAY, showing that autonomous methods are more attractive than pure methods.

Of course, we have no way to guarantee super-linear speedup for every fault. Complementary heuristics can often enhance super-linear speedup. In an architecture, what kind of combination of heuristic information most likely reaches super-linear speedup? This is one of the interesting problems which will be investigated in the future. Is it possible to implement these parallel processing systems on a single VLSI chip? If so, how? What are the bounds for area, time, and area time squared? These are exciting problems to be researched. The key to these problems is how to design a very elegant checking algorithm so as to implement it on a small area, and how to solve the storage problem for circuit description since a VLSI circuit contains so many logic gates. Once these problems are solved, automatic test pattern generation will be much less expensive than nowadays. As well, similar algorithms can be developed for many other NP-complete problems.

Bibliography

[1] Zhimin Shi, and Paul Gillard, *A parallel Processing Architecture & Algorithm for ATPG*, Canadian 7th Annual High Performance Computing Conference, June, 1993

[2] Zhimin Shi, and Paul Gillard, *Quantitative Approach for Redundancy Identification In ATPG*, The Third International Conference for Young Computer Scientists, July, 1993, Beijing, P. R. China

[3] Zhimin Shi, and Paul Gillard, *Using Square Array Structures in Parallel ATPG*, 1993 Canadian Conference on Electrical and Computer Engineering, September, 1993, Vancouver, Canada

[4] J.P.Roth, *Diagnosis of Automata Failures: A Calculus and a Method*, IBM J.Research and Development, Vol.10, July 1966

[5] Eichelberger, E.B. and Williams, T.M., *A Logic Design Structure for LSI Testability*, 14th Design Automation Conference Proceedings, June, 1977

[6] Jeffrey D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, ISBN 0-914894-95-1, 1983

[7] Melvin A. Breuer, Arthur D. Friedman, *Diagnosis & Reliable Design of Digital System*, Computer Science Press, Inc., ISBN 0-914894-57-9, 1976

[8] Robert H. Klenke, Ronald D. Williams, and James H. Aylor, *Parallel-Processing Techniques for Automatic Test Pattern Generation*, IEEE Computer, January 1992

[9] S.J. Chandra and J.H. Patel, *Experimental Evaluation of Testability Measures for Test Generation*, IEEE Trans. Computer-Aided Design. Nol. 8. No. 1, Jan. 1989

[10] S.J.Chandra and J.H.Patel, *Test Generation in a Parallel Processing Environment*, Proc. IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, CS Press, Los Alamitos, Calif., Order No872, 1988, pp.11-14

[11] Akira Motohara, Kenji Nishimura, Hideo Fujiwara, and Isar Shrakawa, *A Parallel Scheme For Test Pattern Generation*, ICCAD-86, 1986

[12] H.Fujiwara and T.Inoue, *Optimal Granularity of Test Generation in a Distributed Syste* IEEE Trans. Computer-Aided Design, Vol.9, No.8, Aug., 1990

[13] S. Patil and P. Banerjee, *A Parallel Branch-and-Bound Algorithm for Test Generation*, Proc. 26th ACM/IEEE Design Automation Conf., June 1989

[14] S. Patil and P. Banerjee, *Fault Partitioning Issues in an Integrated Parallel Test Generation/Fault Simulation Environment*, Proc. 1989 Int'l Test Conf., CS Press, Los Alamitos, Calif., Order No. 1962, 1989, pp. 718-726

[15] Tracy Larrabee, *Test Pattern Generation Using Boolean Satisfiability*, IEEE Trans. on COMPUTER-AIDED DESIGN, Vol. 11, No. 1, January 1992

[16] Kazuo Iwama, *Complementary Approaches to CNF Boolean Equations*, Proc. of the Japan-US Joint Seminar on Discrete Algorithm and Complexity, June, 1986, Kyoto, Japan

[17] Hideo Fujiwara, *Logic Testing and Design for Testability*, The MIT Press, 1985

[18] Miron Abramovici, et al., *Dynamic Redundancy Identification in Automatic Test Generation*, IEEE Trans. on COMPUTER-AIDED DESIGN, Vol. 11, No. 3, March 1992

[19] S.A.Cook, *The complexity of theorem proving procedures*, Proc. Third Annual ACM Symp. Theory of Computing, 1971

[20] H. Fujiwara and T. Shimono, *On the acceleration of test generation algorithms*, IEEE Trans. Comput., Vol. C-31, 1983

[21] P. Goel, *An implicit enumeration algorithm to generate tests for combinational logic circuits*, IEEE Trans. Comput., Vol. C-31, 1981

[22] M.H. Schulz, et al, *Socrates: A highly efficient automatic test pattern generation system*, IEEE Trans. Computer-Aided Design, Vol. 7, Jan., 1988

[23] David Bryan, *The ISCAS'85 benchmark circuits and netlist format*, email: bryan@mcnc.org, 9-30-88

[24] Vishwani D. Agrawal and Sharad C. Seth, *Tutorial : Test Generation for VLSI Chips*, The Computer Society, ISBN 0-8186-8786-X

[25] Gordon Russell and Lan L. Sayers, *Advanced Simulation and Test Methodologies for VLSI Design*, Van Nostrand Reinnold (International), ISBN 0-7476-0001-5

[26] Kwang-Ting Cheng, Vishwani D. Agrawal, *Unified Methods for VLSI Simulation and Test Generation*, Kluwer Academic Publishers, ISBN 0-7923-9025-3

[27] Michael R. Garey, David S. Johnson, *COMPUTERS AND INTRACTABILITY, A Guide to the Theory of NP-Completeness*. W.H.FREEMAN AND COMPANY, ISBN 0-7167-1044-7, 1979

[28] Kuratowski, G., *Sur le Probleme des Courbes Gauches en Topologie*. Fund. Math., 15(1930), 271-283

[29] Hopcroft, J.E., and J.K.Wong, *Linear time algorithm for isomorphism of planar graphs (Preliminary report)* Proc. 6th Ann ACM Symp. on Theory of Computing, New York, 172-184

[30] Parchomenko, P.P. (ed.) *Technical Diagnostic Fundamentals*, Energiya, Moskva.

[31] Williams, T.W. and Parker, K.P. *Design for testability — A survey*, Proc. IEEE, 71(1), 98-113, 1983

[32] Roth, J.P. *Diagnosis of automata failures: a calculus and a method*, IBM J. Res. and Dev. 10, 278-81, 1966

[33] Ulrich,E.G. and Baker, E.T, *Concurrent simulation of nearly identical digital networks*, Computer, April, 39-44, 1974

[34] Burgess, N., Damper, R.I., Shaw, S.J. and Wilkins, D.R.J. *Faults and fault effects in NMOS Circuits — impact on DFT*, Proc. IEE, Pt.G, 132(3), 82-9, 1985

[35] Jeffrey D.Smith, *Design and Analysis of ALGORITHMS*, PWS-KENT Publishing Company, 1989, ISBN 0-534-91572-8

[36] Sun microsystems, *Programming Utilities & Libraries*, Part Number: 800-3847-10, Revision A of 27 March, 1990