# Clustering of Test Cubes: a Procedure for the Efficient Encoding of Complete Test Sets Based on the Intelligent Reseeding of LFSRs

Ronald Alleyne

McGill University, Montreal

May 10, 1994

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering.

Name Alleyne, Ronald Marc

*Dissertation Abstracts International* is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided

Engineering, Electronics and Electrical

SUBJECT TERM

| 0 | 5 | 4 | 4 |
|---|---|---|---|

SUBJECT CODE

U·M·I

## Subject Categories

## THE HUMANITIES AND SOCIAL SCIENCES

### COMMUNICATIONS AND THE ARTS

| | |
|---|---|
| Architecture | 0729 |
| Art History | 0377 |
| Cinema | 0900 |
| Dance | 0378 |
| Fine Arts | 0357 |
| Information Science | 0723 |
| Journalism | 0391 |
| Library Science | 0399 |
| Mass Communications | 0708 |
| Music | 0413 |
| Speech Communication | 0459 |
| Theater | 0465 |

### EDUCATION

| | |
|---|---|
| General | 0515 |
| Administration | 0514 |
| Adult and Continuing | 0516 |
| Agricultural | 0517 |
| Art | 0273 |
| Bilingual and Multicultural | 0282 |
| Business | 0688 |
| Community College | 0275 |
| Curriculum and Instruction | 0727 |
| Early Childhood | 0518 |
| Elementary | 0524 |
| Finance | 0277 |
| Guidance and Counseling | 0519 |
| Health | 0680 |
| Higher | 0745 |
| History of | 0520 |
| Home Economics | 0278 |
| Industrial | 0521 |
| Language and Literature | 0279 |
| Mathematics | 0280 |
| Music | 0522 |
| Philosophy of | 0998 |
| Physical | 0523 |
| Psychology | 0525 |
| Reading | 0535 |
| Religious | 0527 |
| Sciences | 0714 |
| Secondary | 0533 |
| Social Sciences | 0534 |
| Sociology of | 0340 |
| Special | 0529 |
| Teacher Training | 0530 |
| Technology | 0710 |
| Tests and Measurements | 0288 |
| Vocational | 0747 |

### LANGUAGE, LITERATURE AND LINGUISTICS

| | |
|---|---|
| Language | |
| General | 0679 |
| Ancient | 0289 |
| Linguistics | 0290 |
| Modern | 0291 |
| Literature | |
| General | 0401 |
| Classical | 0294 |
| Comparative | 0295 |
| Medieval | 0297 |
| Modern | 0298 |
| African | 0316 |
| American | 0591 |
| Asian | 0305 |
| Canadian (English) | 0352 |
| Canadian (French) | 0355 |
| English | 0593 |
| Germanic | 0311 |
| Latin American | 0312 |
| Middle Eastern | 0315 |
| Romance | 0313 |
| Slavic and East European | 0314 |

### PHILOSOPHY, RELIGION AND THEOLOGY

| | |
|---|---|
| Philosophy | 0422 |
| Religion | |
| General | 0318 |
| Biblical Studies | 0321 |
| Clergy | 0319 |
| History of | 0320 |
| Philosophy of | 0322 |
| Theology | 0469 |

### SOCIAL SCIENCES

| | |
|---|---|
| American Studies | 0323 |
| Anthropology | |
| Archaeology | 0324 |
| Cultural | 0326 |
| Physical | 0327 |
| Business Administration | |
| General | 0310 |
| Accounting | 0272 |
| Banking | 0770 |
| Management | 0454 |
| Marketing | 0338 |
| Canadian Studies | 0385 |
| Economics | |
| General | 0501 |
| Agricultural | 0503 |
| Commerce Business | 0505 |
| Finance | 0508 |
| History | 0509 |
| Labor | 0510 |
| Theory | 0511 |
| Folklore | 0358 |
| Geography | 0366 |
| Gerontology | 0351 |
| History | |
| General | 0578 |
| Ancient | 0579 |
| Medieval | 0581 |
| Modern | 0582 |
| Black | 0328 |
| African | 0331 |
| Asia, Australia and Oceania | 0332 |
| Canadian | 0334 |
| European | 0335 |
| Latin American | 0336 |
| Middle Eastern | 0333 |
| United States | 0337 |
| History of Science | 0585 |
| Law | 0398 |
| Political Science | |
| General | 0615 |
| International Law and Relations | 0616 |
| Public Administration | 0617 |
| Recreation | 0814 |
| Social Work | 0452 |
| Sociology | |
| General | 0626 |
| Criminology and Penology | 0627 |
| Demography | 0938 |
| Ethnic and Racial Studies | 0631 |
| Individual and Family Studies | 0628 |
| Industrial and Labor Relations | 0629 |
| Public and Social Welfare | 0630 |
| Social Structure and Development | 0700 |
| Theory and Methods | 0344 |
| Transportation | 0709 |
| Urban and Regional Planning | 0999 |
| Women's Studies | 0453 |

## THE SCIENCES AND ENGINEERING

### BIOLOGICAL SCIENCES

| | |
|---|---|
| Agriculture | |
| General | 0473 |
| Agronomy | 0285 |
| Animal Culture and Nutrition | 0475 |
| Animal Pathology | 0476 |
| Food Science and Technology | 0359 |
| Forestry and Wildlife | 0478 |
| Plant Culture | 0479 |
| Plant Pathology | 0480 |
| Plant Physiology | 0817 |
| Range Management | 0777 |
| Wood Technology | 0746 |
| Biology | |
| General | 0306 |
| Anatomy | 0287 |
| Biostatistics | 0308 |
| Botany | 0309 |
| Cell | 0379 |
| Ecology | 0329 |
| Entomology | 0353 |
| Genetics | 0369 |
| Limnology | 0793 |
| Microbiology | 0410 |
| Molecular | 0307 |
| Neuroscience | 0317 |
| Oceanography | 0416 |
| Physiology | 0433 |
| Radiation | 0821 |
| Veterinary Science | 0778 |
| Zoology | 0472 |
| Biophysics | |
| General | 0786 |
| Medical | 0760 |

### EARTH SCIENCES

| | |
|---|---|
| Biogeochemistry | 0425 |
| Geochemistry | 0996 |
| Geodesy | 0370 |
| Geology | 0372 |
| Geophysics | 0373 |
| Hydrology | 0388 |
| Mineralogy | 0411 |
| Paleobotany | 0345 |
| Paleoecology | 0426 |
| Paleontology | 0418 |
| Paleozoology | 0985 |
| Palynology | 0427 |
| Physical Geography | 0368 |
| Physical Oceanography | 0415 |

### HEALTH AND ENVIRONMENTAL SCIENCES

| | |
|---|---|
| Environmental Sciences | 0768 |
| Health Sciences | |
| General | 0566 |
| Audiology | 0300 |
| Chemotherapy | 0992 |
| Dentistry | 0567 |
| Education | 0350 |
| Hospital Management | 0769 |
| Human Development | 0758 |
| Immunology | 0982 |
| Medicine and Surgery | 0564 |
| Mental Health | 0347 |
| Nursing | 0569 |
| Nutrition | 0570 |
| Obstetrics and Gynecology | 0380 |
| Occupational Health and Therapy | 0354 |
| Ophthalmology | 0381 |
| Pathology | 0571 |
| Pharmacology | 0419 |
| Pharmacy | 0572 |
| Physical Therapy | 0382 |
| Public Health | 0573 |
| Radiology | 0574 |
| Recreation | 0575 |
| Speech Pathology | 0460 |
| Toxicology | 0383 |
| Home Economics | 0386 |

### PHYSICAL SCIENCES

#### Pure Sciences

| | |
|---|---|
| Chemistry | |
| General | 0485 |
| Agricultural | 0749 |
| Analytical | 0486 |
| Biochemistry | 0487 |
| Inorganic | 0488 |
| Nuclear | 0738 |
| Organic | 0490 |
| Pharmaceutical | 0491 |
| Physical | 0494 |
| Polymer | 0495 |
| Radiation | 0754 |
| Mathematics | 0405 |
| Physics | |
| General | 0605 |
| Acoustics | 0986 |
| Astronomy and Astrophysics | 0606 |
| Atmospheric Science | 0608 |
| Atomic | 0748 |
| Electronics and Electricity | 0607 |
| Elementary Particles and High Energy | 0798 |
| Fluid and Plasma | 0759 |
| Molecular | 0609 |
| Nuclear | 0610 |
| Optics | 0752 |
| Radiation | 0756 |
| Solid State | 0611 |
| Statistics | 0463 |

#### Applied Sciences

| | |
|---|---|
| Applied Mechanics | 0346 |
| Computer Science | 0984 |
| Engineering | |
| General | 0537 |
| Aerospace | 0538 |
| Agricultural | 0539 |
| Automotive | 0540 |
| Biomedical | 0541 |
| Chemical | 0542 |
| Civil | 0543 |
| Electronics and Electrical | 0544 |
| Heat and Thermodynamics | 0348 |
| Hydraulic | 0545 |
| Industrial | 0546 |
| Marine | 0547 |
| Materials Science | 0794 |
| Mechanical | 0548 |
| Metallurgy | 0743 |
| Mining | 0551 |
| Nuclear | 0552 |
| Packaging | 0549 |
| Petroleum | 0765 |
| Sanitary and Municipal | 0554 |
| System Science | 0790 |
| Geotechnology | 0428 |
| Operations Research | 0796 |
| Plastics Technology | 0795 |
| Textile Technology | 0994 |

### PSYCHOLOGY

| | |
|---|---|
| General | 0621 |
| Behavioral | 0384 |
| Clinical | 0622 |
| Developmental | 0620 |
| Experimental | 0623 |
| Industrial | 0624 |
| Personality | 0625 |
| Physiological | 0989 |
| Psychobiology | 0349 |
| Psychometrics | 0632 |
| Social | 0451 |

An efficient encoding of test sets based on the reseeding of LFSRs.

# Abstract

An approach for input data compaction in the testing of circuits using scan and partial scan has recently been developed based on the reseeding of multi-polynomial Linear Feedback Shift Registers (LFSRs). This thesis explores further compression of test data through the grouping of closely related vectors into clusters and extracting both common information, in a form compatible with the reseeding method, and individual information for the component vectors.

Two greedy algorithms for finding these clusters are described, the first trying to form the largest groupings while the second trying to make the most advantageous mergers at each step. In addition, several schemes of storing and decoding the information needed for individual vectors are presented, some of which use memories in the form of stacks or caches to take advantage of the distribution of this data. Aside from the LFSR, some counters and the memory, little additional hardware is required. The solutions offer a range of trade-offs between test length, hardware complexity and test data storage.

Results are given for experiments carried out on ISCAS-89 benchmark circuits and on a set of industrial circuits contrasting the performances of the algorithms and the memory requirements of the different methods of storage. Considerable improvements over reseeding are demonstrated, more so for the industrial circuits which are inherently hard to test.

# Résumé

Récemment, une méthode axée sur la ré-initialisation des compteurs pseudo-aléatoires à multiple polynômes a été introduite pour la compression des données à l'entrée des chaînes de balayage, ainsi que les chaînes de balayage partielles. Cette thèse explore une amélioration à la méthode de ré-initialisation en prenant avantage du regroupement de vecteurs tests qui sont reliés par leur similitude. Un groupe de vecteurs est formé en partageant l'information qui relie ses vecteurs et celle qui les distingue. Le processus de regroupement se fait de façon à ce qu'il demeure compatible à la méthode de ré-initialisation.

Deux algorithmes voraces sont présentés pour la formation des groupes. Le premier tente de former des groupes à forte population tandis que le second tente de distribuer les vecteurs à travers les groupes de façon optimale. En outre, divers arrangements pour le storage et le décodage de l'information qui distingue les vecteurs d'un groupe sont présentés. Certains de ces arrangements utilisent une mémoire organisée en pile ou en antémémoire dans le but de profiter de la distribution de cette information. Sans considérer la chaîne à balayage, quelques compteurs et la mémoire nécessaire, notre méthode requiert peu de circuits de soutien. Les différentes solutions que nous proposons varient selon la longueur du test, la complexité du circuit de soutien et la quantité de mémoire requise.

Des résultats expérimentaux ont été compilés pour les circuits étalons ISCAS-89 et une série de circuits industriels. Les résultats démontrent la différence de performance entre les deux algorithmes et les variations en quantité de mémoire requise pour les diverses configurations de mémoire. Des gains importants sont obtenus par rapport à la méthode de ré-initialisation sans regroupement, particulièrement pour les circuits industriels qui sont difficiles à vérifier.

# Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Janusz Rajski, whose guidance, support and encouragement throughout the course of my graduate studies has been invaluable. His infectious drive and enthusiasm cannot be understated. I would also like to thank Dr. Leendert Huisman and Dr. Sandip Kundu of IBM Corp. for supplying additional benchmark circuits.

I am extremely grateful to all staff and students at the MACS Lab for their friendship, unhesitant assistance and sense of camaraderie, specifically Charles Arsenault, Mark Kassab, Eric Masson and Michael Toner. Special thanks to Eric for his help with the French version of the abstract, and to Mark for his helpful hints throughout. I am extremely indebted to all my friends, especially to Mary Ludovico, Tania Boutilier and Lisa Ramsaran who were always there, and to the members of the McGill badminton club who were always hungering for a game. But most importantly, I wish to thank my family for their support, and unconditional love and understanding.

Lastly, I am most appreciative to Mr. Watterson for his unique sense of humour.

# Glossary of Terms

**ATPG** Automatic Test Pattern Generation.

**BIST** Built-in Self-Test.

**CA** Cellular Automata.

**CI** Cube Implicit.

**CUT** Curcuit Under Test.

**DFT** Design for Testability.

**GURT** Generator of Unequiprobable Random Patterns.

**ISCAS circuits** A set of benchmark circuits from the International Symposium on Circuits and Systems.

**LFSM** Linear Finite State Machine.

**LFSR** Linear Feedback Shift Register.

**MICA** Multiple Input Cellular Automata.

**MISR** Multiple Input Shift Register.

**TCK** Test Clock.

**TDI** Test Data In.

**TDO** Test Data Out.

**TMS** Test Mode Select.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As the design of digital intergrated circuits make great inroads into everyday life, emphasis on placing larger and more complex designs in smaller areas using technologies with more minute feature sizes is increasing. As a consequence of this trend, the need for testing is becoming critical while the penalties for not considering testing features at design time are becoming exorbitant. In addition to determining the integerity of a newly fabricated die, a well designed testing strategy may be an invaluable tool in later stages of product life, useful at board and system levels, and for field test and diagnosis.

In spite of these benefits, testing does not come without an associated cost which accrues from

- The guidelines employed by designers to help ensure the testability of a design which in many cases limit their flexibility.
- Silicon overhead due to modifications and additional structures to aid testability. This also contributes to a reduction in yield and a possible increase in packaging.
- Possibility of increase in the delay of the circuit resulting in a degradation in performance. However, through careful redesign, this can often be eliminated.

- Significant computational overhead to generate, where necessary, vectors, weights and coverage.
- The amount of data, whether it be weights or actual test vectors, that needs to be stored or applied to the circuit.
- The time to apply the test data to the circuit.
- The use of expensive test equipment.

Chapter 2 consists of some motivation to the need and uses of circuit testing followed by an introduction to various basic testing concepts and an overview of the areas of activity. Chapter 3 introduces a novel method called *reseeding* which is instrumental to the further work described in the chapters following.

# Chapter 2

# The Testing of Digital Circuits

A digital circuit accepts a set of values at its inputs (the *input vector*) and, as a result, produces a set of values on its outputs (the *output vector*) which depends on the input vector and on the state of the circuit if it is a sequential circuit. The *state* of a sequential circuit is determined by the values of memory elements contained within the circuit.

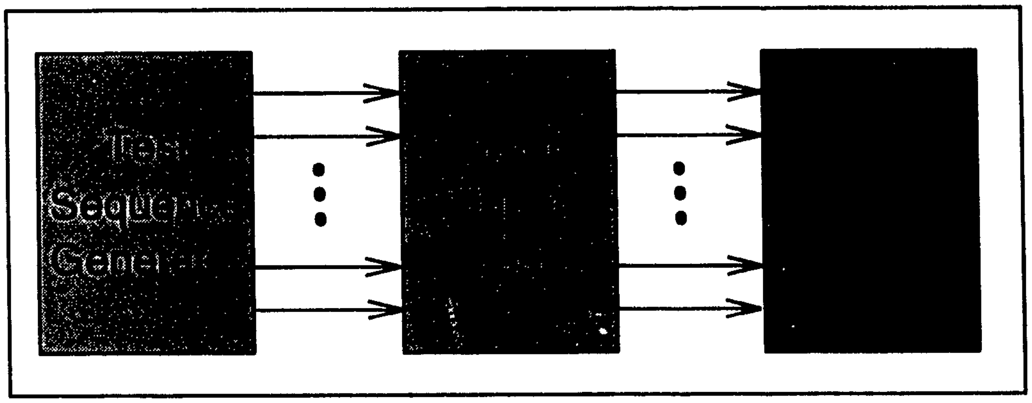


Figure 2.1: The Testing of a Circuit

The objective of testing is twofold: insuring that the circuit does what it is supposed to and that it is built correctly [Info90]. The first objective is assumed to be taken care of in the design and synthesis stages thus emphasis is placed on verifying that the fabricated circuit matches the netlist. Generally this is done by analyzing the response of a circuit based on an input vector. An incorrect response implies that the circuit is faulty while a correct one, though not decisive on the correct working of the whole circuit, indicates that certain defects cannot be present. Hence many tests may be needed to obtain a required confidence to the proper operation of the circuit.

When testing a circuit, it must be driven by a known set of inputs, and the values observed on the outputs analyzed. However, many circuits contain memory elements or have portions which are hard to test as is and may be modified to aid in the testing process. Figure 2.1 is a representation of a circuit under test (CUT) showing two major aspects of testing, the generation of test patterns and the analysis of the circuit's response.

## 2.1 Motivation

In view of the random processes which dictate present fabrication methods, defects and process variations which adversely affects the behavior of circuits is an expected result. Further to this, the constant drive to denser packing with smaller feature sizes continuously pushes these technologies to their limits and have precipitated the implementation of new but still immature ones resulting in yet more defects. Also contributing to failures are 'handling errors' caused by the presence of dust particles (even microscopic ones) or a shifting of the masks. Hence faulty chips are a natural consequence of the process.

Table 2.1 [Pen92] demonstrates that even a low probability of chip failure (1%) translates to a high probability of failure at the board level. Assuming that the board contains 40 (or 200) chips, the probability that the board would fail could reach as high as 33.1% (or 86.6%). Furthermore, a well accepted metric in the test community is *the rule of tens* which states that each level of testing (wafer, chip, board, system) a defective element escapes increases the cost correction by a factor of ten [Dav82, Bar87]. It is therefore essential that a very high quality of test is obtained as early in the process as possible.

| Number of Chips | Defects per Million | Probability of failure |
|---|---|---|
| 40 | 10,000 | 33.1% |
| | 1,000 | 4.0% |
| | 100 | 0.4% |
| 200 | 10,000 | 86.6% |
| | 1000 | 18.0% |
| | 100 | 2.0% |

Table 2.1: Estimated Board Failure Rate given a Defect Level

Demand has led to an explosive growth in the level of integration provided. However, if nothing is done to ease the problem, the cost of test increases faster than linearly for a linear increase in circuit complexity [Tur90]. So while the total product cost has decreased, test cost has risen to more than 55% of this cost in some cases [Dea91]. To combat this trend, new and innovative techniques are continuously in demand even to keep costs at their present level.

Defects are a natural, if undiserable outcome of the fabrication process, and through packaging into chips/boards/systems and through general use, more errors are expected to develop. In general, it is felt that the current level of testing is barely, if at all adequate, yet the greater complexity demanded of chips further complicates the problem. Testing is an essential step to guarantee the quality of chips produced and can be used as a valuable tool at later stages.

## 2.2 Defects and Fault Modeling

Dealing with these physical defects is generally intractable, except for the smallest of circuits, and is highly technology dependent. However, defects of interest can be mapped to definable logic behavior with many fewer possibilities. These maps are called *fault models*.

The *stuck at* fault model [Poa62, Sch72] is, by far, the most prevalent in the testing industry and is in fact the *defacto* standard. This model assumes three possible modes of behavior for each line in the circuit, fault free behavior, the line with value always logic '0' (*stuck-at-0*) or always logic '1' (*stuck-at-1*). As each line has 3 possible modes, in a circuit with $n$ lines there are $3^n - 1$ possible stuck-at fault combinations. This becomes very large even for moderately sized circuits. In view of this, multiple faults are not usually explicitly considered. Though this assumption does not reflect reality, and the presence of multiple faults may mask each other, it has been shown that tests found for single fault sets do well for multiple faults as well [Hug86, Jac87, Wai88]. An alternative to this is to find fault-free lines [Raj87] which implicitly considers all possible modeled faults.

In addition to stuck-at faults, other fault models have been introduced and studied, many of which are technology dependent. Some of these *non-classical* faults result in incorrect non-stuck-at behavior and even in memory elements being introduced into the circuit. *Stuck-on* and *stuck-open* faults [Wad78, Cha85, Jha86] are modeled from MOS technology and correspond to transistors being permanently conducting or permanently broken. These often result in memory being introduced into the circuit and thus require two vectors, an initialization input followed by a test input.

*Bridging faults* [She85] involves two or more lines being shorted together resulting in unpredictable logic values when the lines have conflicting assignments. *Delay faults* [Smi85, Lin87] models failures which may cause unacceptable delays along paths from inputs to outputs.

A large portion of these non-classical faults are covered by test sets determined for single stuck-at behavior. This model is assumed throughout this thesis unless otherwise indicated.

## 2.3 Fault Sensitization and Propagation

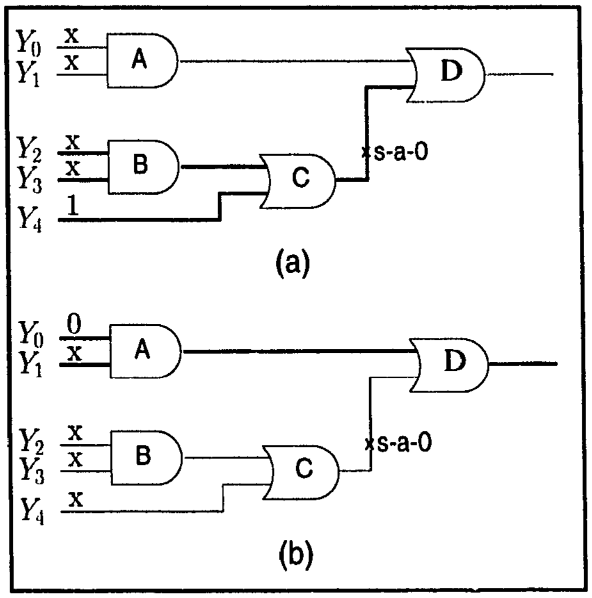


Figure 2.2: Sensitizing and Propagating Faults

To test for a fault on a line, the converse value must be asserted and then propagated to at least one output. Figures 2.2(a) and (b) illustrate path sensitization and propagation respectively. The input cube (x,x,x,x,1) excites the stuck-at-0 (s-a-0) fault site by asserting a '1' on this line for fault-free behavior. The input cube (0,x,x,x,x) produces non-controlling values along a path from the fault site to an output allowing the presence of this fault to be observed on the output. Any non-conflicting combination of cubes which sensitize and propagate a fault is a valid test. (0,x,x,x,1) is such a candidate.

## 2.4 Design for Testability and Built-In Self-Test

DFT techniques are guidelines and methods, implemented at design time, which improve the testability of a circuit or system. They accomplish this in two basic ways, by augmenting some structures to make them more testable, and by completely avoiding others which are known to be hard to test. Generally structures are argumented to increase the ability to control or observe the value of a line while structures which introduce timing problems, such as asynchronous circuits, are avoided.

Some of these guidelines include [Bar87, Abr90, Tur90]:

- Isolating clocks form logic.
- Avoiding asynchronous logic.
- Making sequential circuits initializable.
- Avoiding redundancy.
- Using test points. Includes scan and boundary scan.
- Partitioning long counters and shift registers.
- Partitioning large circuits.

The built-in self-test (BIST) approach is an extension to DFT, to have a chip test itself. Though this is a laudable goal, it is not expected to totally replace external testers [Info90] as it cannot measure input and output characteristics to the required accuracy. It does, however, reduce the cost of test and lends itself well, when coupled with boundary scan, to the hierarchical solution of the testing problem [Agr93]. The

BIST strategy is to have all circuitry for the generation of patterns and the analysis of the circuit's response on-chip so, upon its initiation, the circuit self-tests and returns a single value indicating whether it passes or fails.

## 2.5 Test Points and Scan Techniques

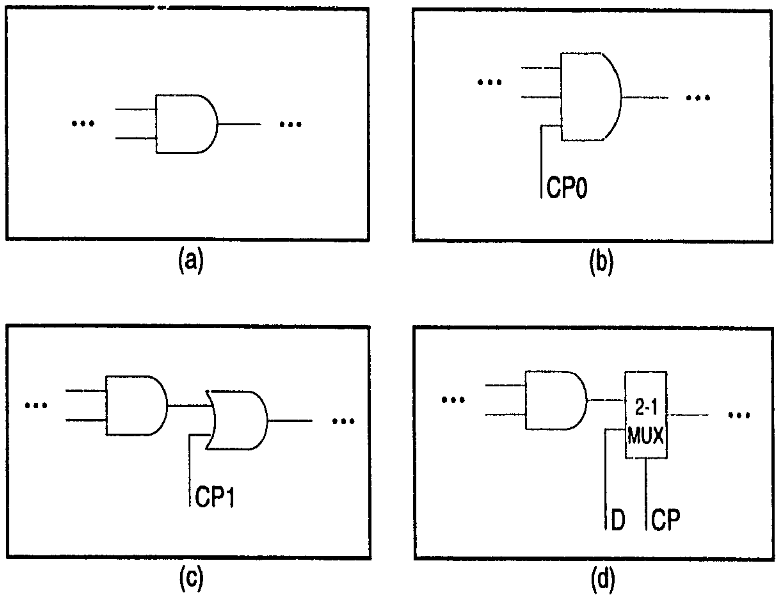


Figure 2.3: Adding control points: (a) the original circuit, (b) forcing 0 on the line, (c) forcing 1 on the line, (d) forcing both

Difficulties arise when a fault is hard to sensitize (*controllability*) or hard to propagate (*observability*). The introduction of *test points* can be used to ease this problem. Figure 2.3 shows an example where control points are added to a circuit in (a) to force a '0' on the line in (b), to force a '1' on the line in (c) and to force both in (d).

One feature which makes a circuit hard to test is the presence of storage elements such as flip-flops and registers. For a valid test, the values in these elements must be determined, controlled and observed. Often, before a useful vector can be applied to

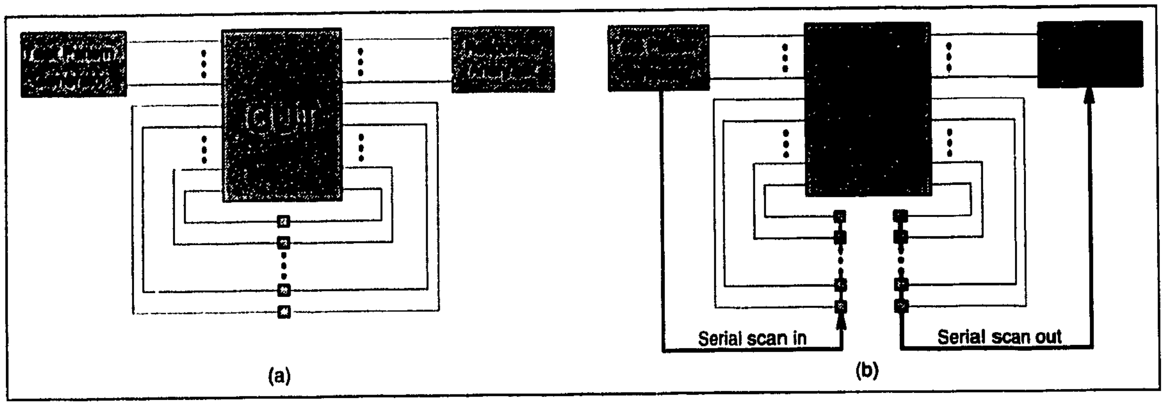


Figure 2.4: (a) Model of a Sequential Circuit Under Test (b) Model of a Sequential Circuit with Scan Under Test

the circuit, a setup stage must be executed to set certain values in the circuit, and after the vector is submitted, several cycles may pass before the effects of a fault may propagate to a primary output. Though several attempts have been made to tackle this problem [Ma88, Agr89, Pom91], an effective solution for large circuits has yet to be developed. Thus it is common practice to transform sequential circuits during test through the use of *scan design*. Both full scan [Wil73, Mcc85], in which all the memory elements are modified and chained to form a shift-register, and partial scan [Tri80, Agr87], where not all of the flip-flops form part of the scan chain, are examples of this technique. The effect of this is to break the feedback during testing as depicted in figure 2.4 and transform these flip-flops into test points which are fully controllable and observable.

Full scan essentially reduces the test problem from a sequential circuit test to the test of a combinational circuit plus a shift register. Though this is an NP-complete problem [Iba75], many successful automatic test pattern generators exist based on tractable heuristics [Raj87, Lis87, Sch88] giving the possibility for near-100% stuck-at fault coverage.

However, often-cited penalties of using scan [Dea91] include

- Additional design effort.
- Additional circuitry (4-20% overhead).
- Additional device pins, sometimes requiring the use of the next package size which takes up more space and costs more.
- Possible increase in test application time. A significant amount of the test time is spent shifting patterns in and out of scan chains which may be thousands of bits long.
- Multiplexors used in scan cells to select between the regular and the scan chain inputs may introduce delays in the circuit which may force the use of a slower clock.
- Degradation in reliability and yield.

In spite of these, [Dea91] shows that the cost of test favours the use of scan even when the benefits to other levels of testing are not taken into account.

## 2.6 Boundary Scan

Coupled with the increasing density and complexity of circuits on-chip is a similar desire to pack as many chips in as close a proximity as possible. Associated with such techniques as surface mount is a reduction in the ability to access pins and interconnect to verify the connections of the chips to the pins and the routing connecting the pins.

To help alleviate this problem, another type of scan has been proposed which concentrates on the boundaries of the chips (inputs and outputs) converting them to scan chains during test. This is called *boundary scan* and is based on the IEEE/ANSI std. 1149.1-1990 [Mcc85, Glo89, Has92, Zor92]. Figure 2.5 shows a block diagram of a simple boundary scan cell and the configuration of a board under test. Two additional

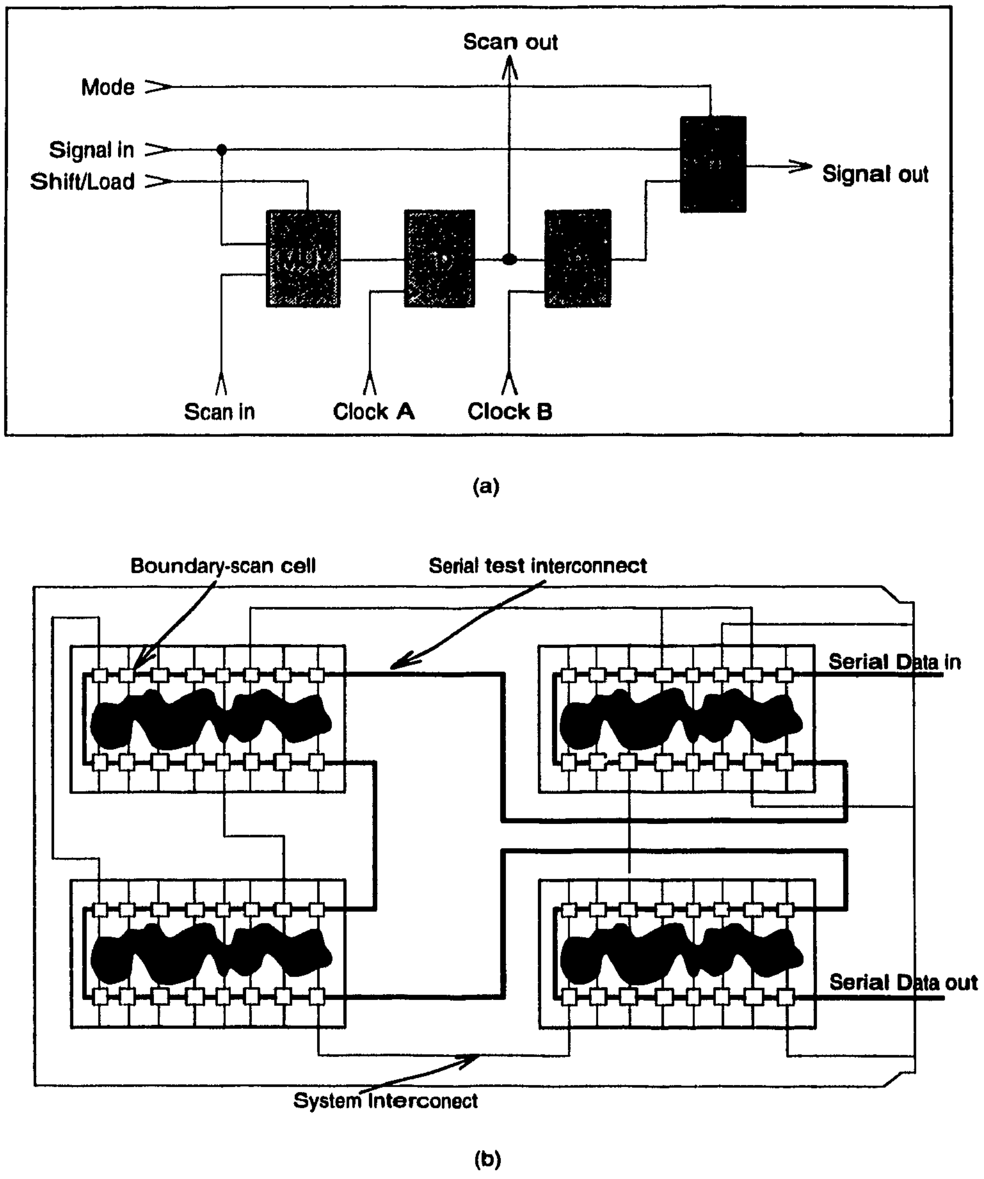


Figure 2.5: (a) A Simple Boundary Scan Cell. (b) A Printed Circuit Board under Test.

control pins, TMS (test mode select) and TCK (test clock), are needed along with two scan pins, TDI (test data in) and TDO (test data out). Boundary scan should support the following modes:

- *External test:* This mode tests the interconnects of the printed circuit board. For each test, data is shifted in providing values at output pins. The values at input pins are captured, shifted out and analyzed.

- *Internal test:* This mode tests the internal logic of the design and the connections to the pins. Data is applied from the input registers to the internal circuit. The response is captured and shifted out.
- *Sample test:* This allows the test engineer to take a snapshot of the circuit in time. Primary input, primary output and interior register values may be captured and shifted out.
- *Bypass:* During in-circuit testing, the engineer may want to test only a few chips. To reduce the scan length in this case, chips are supplied with multiplexors to optionally bypass the chip's boundary-scan path.
- *Built-in self-test:* This mode instructs the chip to carry out self test.

## 2.7 Random Pattern Generators and Response Compactors

A pseudo-random pattern generator is at the core of many test pattern generators and response compactors. They are usually based on one of two designs, *linear feedback shift registers* (LFSR) or *cellular automata* (CA).

### 2.7.1 Linear Feedback Shift Register

An LFSR is a finite state machine commonly used in BIST because it is simple and has a fairly regular structure, its shift property integrates well with scan, and it can generate exhaustive and/or pseudorandom patterns with many random properties. It is made up of a chain of flip-flops and XOR (XNOR) gates and implements polynomial division on the input sequence. There are two canonical structures, both equivalent, which are shown on figure 2.6. Associated with each LFSR is a polynomial equation which characterizes it and can be used to predict LFSR behavior. A brief summary of LFSR theory can be found in [Agr93] while a more complete mathematical treatment can be found in [Gol82].

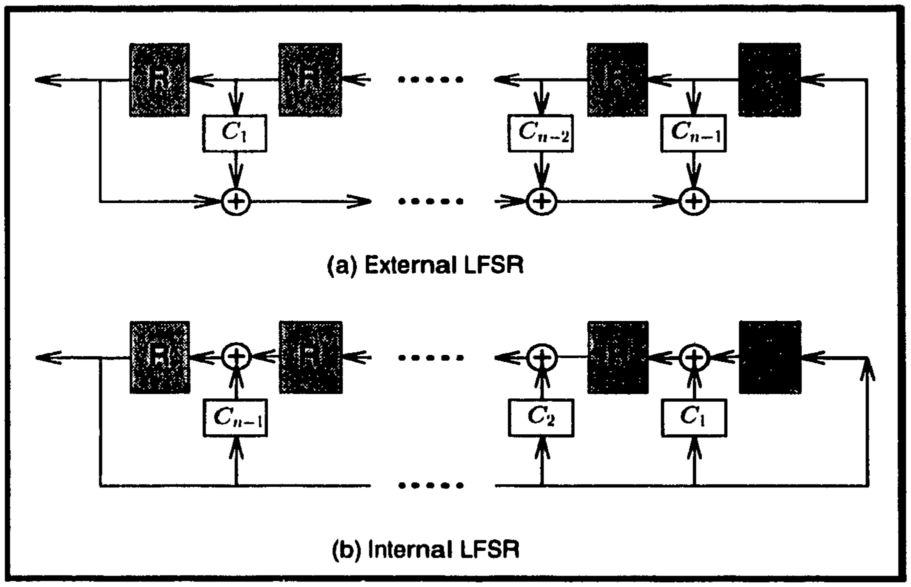


Figure 2.6: Two LFSR implementations of $C_nX^n + C_{n-1}X^{n-1} + ... + C_1X + 1$

The value contained in a register can be expressed as a polynomial. For instance a binary vector $V = v_mv_{m-1} \dots v_0$ can be written as $v_mX^m + v_{m-1}X^{m-1} + ... + v_0$. An interesting observation is that for the internal LFSR, the value contained in the register, when the input is null, is the remainder of the polynomial division of the previous value times $X$ while the output is the quotient. So, if the present value in a LFSR whose characteristic polynomial is $\mathcal{C}(x)$ is $\mathcal{G}_0(x)$, then the output after $k$ cycles would be $X^k\mathcal{G}_0(x)$ *div* $\mathcal{C}(x)$ while the value in the register, $\mathcal{G}_k(x)$ would be $X^k\mathcal{G}_0(x)$ *mod* $\mathcal{C}(x)$.

A discrable property possessed by some LFSRs is its ability to generate *maximal* *length sequences* before repeating. Maximal length sequences are of length $2^n - 1$ and represent all possible non-zero vectors. Such LFSRs are said to have a *primitive* characteristic polynomial. They are better as generators when doing pseudorandom testing for one does not need to worry about the length of sequence given a starting value and no pattern can be excluded, and they have better aliasing properties when used for signature analysis [Wil87]. A table of primitive polynomials up to length 300 can be found in [Bar87].

For signature analysis, an LFSR can be converted to accept inputs by adding exclusive-or gates to the inputs of some or all of the inputs to the registers and connecting the inputs there. LFSRs with multiple inputs are known as a *multiple input shift registers* (MISRs).

## 2.7.2 Cellular Automata

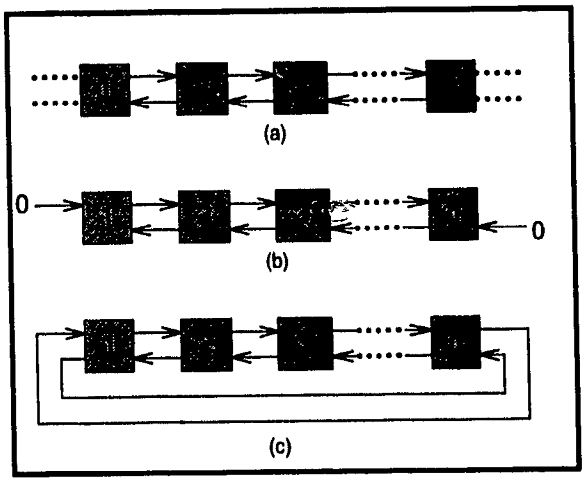


Figure 2.7: (a)A simple 1-dimensional CA. (b) Null boundary conditions. (c) Cyclic boundary conditions

The cellular automata (CA) is another sequential structure which can be used for pseudo-random pattern generation. The value of each cell is calculated based on the previous values of the cells in its neighbourhood. The extent of the neighbourhood can vary, depending, among other factors, upon the dimensionally of the CA under consideration. Only simple 1-dimensional CAs are considered here where the next value depends on a cell's present value and on those of its left and right neighbours. The first and last cells may have fixed values or they can be cyclically connected. For CAs of this type, the *rule* of each cell is based on how the 3 neighbourhood triplet (a word made up of {left value, own value, right value}) determines its next state. This triplet can have 8 possible values each resulting in the cell taking one of two states making 256 possible rules. For instance, table 2.2 gives the mapping for rule 90 (formed by adding the bits in table 2.2 base 2).

| 111 | 110 | 101 | 100 | 011 | 010 | 001 | 000 | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | = 90 |

Table 2.2: Rule 90

In the simplest cases, CAs can be formed with cells of all the same rule. In general, the properties of these CAs are not optimal, and unlike LFSRs, paths connecting subsequent states are not all cyclic but may have paths leading to cycles as shown in figure 2.8. This results in a reduction in the effectiveness of test generation and in increased aliasing [Hor89]. Luckily, not all CAs possess this trait. A family of CAs based on rules 90 and 150 have been found with null boundary conditions and with a cycle of length $2^n - 1$ where $n$ is the length of the CA. These have 'equivalent' or 'better' random properties when compared to LFSRs especially with respect to correlations over space and time between different outputs [Hor89, Hor90, Zha92].

The size of a CA cell in the 90/150 family is larger than that of an LFSR and its structure is not regular, however all connections are to neighbours thus avoiding the routing and driving problems presented by the long feedback loops of LFSRs.

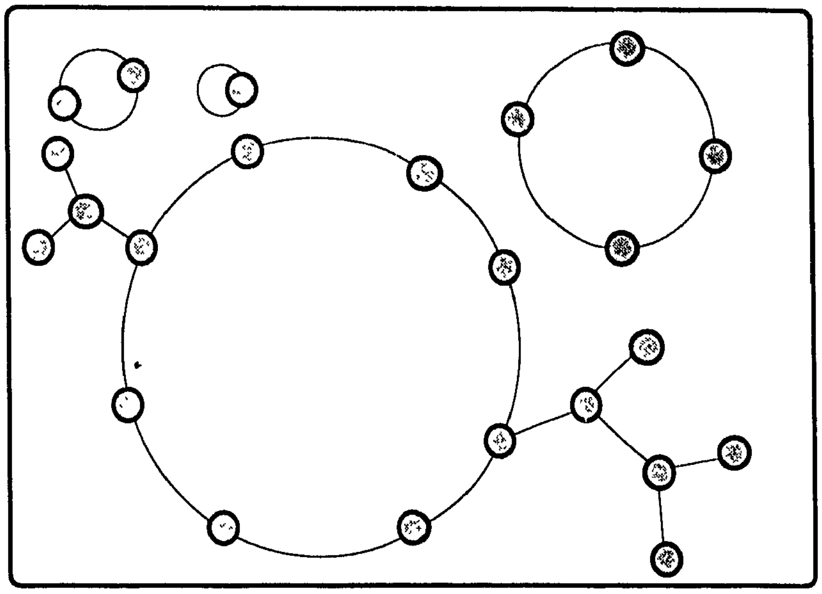


Figure 2.8: Typical cycles and paths to the cycles for cyclic rule 30 CAs

Multiple input CAs (MICA) can be made in much the same way as MISRs based on the exclusive-or of the input with the output of the cell.

## 2.8 Test Response Analysis

The analysis of the response of the CUT is vital in determining whether a circuit works, or further it can indicate where the fault exists. Finding the fault may be important in failure analysis and for the test of multi-chip modules, but it is of little use for the routine testing of chips. Hence several compaction methods have been developed which give a simple pass/fail result and which are very economical in storage and hardware demands but which have a finite probability of overlooking an erroneous response (*aliasing*).

One branch is called *signature analysis* which uses linear finite state machines (LFSM) such as single-input LFSRs and CAs, and multiple-input MISRs and MI-

CAs. Judicial choice of feedback polynomials or CAs are essential. In the case of LFSRs and MISRs, primitive polynomials have better aliasing characteristics in that the aliasing probability reaches the asymptotic limit of $2^{-n}$ more quickly [Wil87]. For CAs and MICAs, rules which result in non-cyclic paths connecting subsequent states give aliasing probabilities above $2^{-n}$ [Hor89, Hor90].

When an LFSMs is used for compaction, the register is started in a known state and it is fed input from the circuit's outputs. At the end of the test, the register ends up in a final state called the signature. If it is as expected, the circuit passes the test.

Another branch uses a counter to keep track of some characteristic of a circuit's output. These methods are as follows:

- *Ones Counting:* The number of ones in the output stream is counted, usually using a $\lceil log_2(n) \rceil$ bit ripple counter where $n$ in the number of test patterns to be applied. The aliasing probability depends on the number of ones in the output stream peaking as this number approaches the half the length of the output stream [Bar87, Abr90].

- *Transition Counting:* The number of 1-to-0 and 0-to-1 transitions in the output stream is kept track of in much the same way as in the case of ones counting except that a transition detection circuit is needed. The aliasing probability is dependent on the number of transitions in the output stream.

## 2.9 Test Pattern Generation Techniques

### 2.9.1 Exhaustive Testing

Exhaustive testing is the process of testing the circuit under test (CUT) using every possible input combination. This can be modified by splitting the CUT into different sub-sections and then applying every possible input combination to the inputs involving the subsection.

This method has very high coverage and tests every non-redundant fault a number of times increasing the chance of detecting unmodeled faults. It uses minimal extra circuitry, for example a counter, thus lends itself well to BIST. However, it has one major drawback which prohibits its use in all but the most simple of circuits. Even for outputs depending on a modest number of inputs, the test length can be such that it takes too long to test the circuit. For example, with thirty two inputs, the number of vectors required is $2^{32}$ ie. $4.3 * 10^9$ vectors. Assuming a system working at 20MHz, and a self test pattern applied every 3 cycles [Wun87], the test would take over 600 sec. As present circuits may have many hundreds of inputs (along scan paths), this method is not adequate.

### 2.9.2 Deterministic Testing

Deterministic testing requires a preliminary step of test generation. This is done to get a test set of vectors which give an acceptable coverage of the modeled fault types. This test set is then stored and used during the testing phase. For larger more complex circuits, on-board storage may be prohibitive thus external testing may have to be done.

This form of testing gives a very high, pre-determined coverage of known faults in a minimal of time. However, if the vectors have to be stored off chip, it cannot be used in BIST and may need expensive testers, and if they are to be stored on chip, the increase in cost due to the extra ROM needed would have to be taken into account. Also, the test vector generation can be quite costly, and the test set may have a lower coverage of non-modeled faults [Wai88] when compared to random pattern type testing.

### 2.9.3 (Pseudo)Random Pattern Testing

This test method requires the application of vectors which randomly cover the input space. Vectors are actually chosen pseudorandomly so that the test set is repeatable. Generally, linear feedback shift registers (LFSRs) or cellular automata (CA) are used for this purpose. Vectors produced using these constructs are not exactly random, but if the number of vectors used are small compared to the total number of states of the LFSR, they require essentially the same number of patterns as if they were random [McC87, Wun88].

In general, random pattern test generation require a minimal of extra circuitry to calculate the patterns and hence, are ideal for BIST. Using a long enough test length, coverage close to 100% can be attained. Unfortunately, an expected test length of $L$ given by $L = \lceil \frac{ln(e_t)-ln(k)}{ln 1-p} \rceil$ is needed for coverage with escape probability $e_t$ for a circuit which has $k$ faults whose detection probabilities are comparable to the minimum detection probability $p$ [Sav84]. For example, to detect up to 50 hard faults, each having a detection probability $p$, with a confidence of 99.9%, a test length of about $11/P_{min}$ would be required. Considering the complexity of circuits, $P_{min}$ can be excessively small requiring the application of a prohibitively long sequence of vectors.

### 2.9.4 Hybrid Combinations

An obvious alternative is to apply a reasonable length of random patterns followed by a deterministic test to test the remaining faults. However, in many cases, it has been found to require stored test sets of up to 70% of the original deterministic test set. Hence, this method alone does not quite address the major disadvantage of deterministic testing, the storage required for the vectors.

### 2.9.5 Weighted Random Pattern Testing

These schemes use prior knowledge of the circuit in order make more intelligent choices when choosing patterns to apply to the CUT. It sacrifices some resources in order to keep this circuit information, as *weights*, and in order to generate these inputs. This is generally done by biasing the probabilities of a '1' in the inputs away from the 0.5 value in random pattern testing.

This method can obtain coverage comparable to deterministic testing with most the advantages of random pattern testing. The gain is that this coverage is obtained in test lengths orders of magnitude less than random, but much more hardware has to be devoted to this.

#### 2.9.5.1 Obtaining Weighted Inputs

In order to get weighted outputs, a simple method relys on the properties of AND and OR gates, where the AND gate acts as the logical disjoint of the probabilities of its inputs, and the OR as the logical union. A sample circuit to obtain weights is given in figure 2.9.

In this circuit, weights of 0.5, 0.25, 0.125 and 0.0625 are generated and fed into a multiplexer. Two of the three bits which represent the weights are used to choose which of the inputs to the multiplexer is chosen. This input is then fed into an XOR gate and may be inverted depending on the third input from the weights. Hence weights of 0.5, 0.25, 0.125, 0.0625, 0.75, 0.875, 0.9375 can be obtained from this simple circuit. These numbers assume that all the inputs from the random generator are independent. Care has to be taken to ensure this.

Two possible ways of generating the weights are used, via local generators, and via global generators. Using local generators, a circuit as in figure 2.9 is present at each of the inputs to the CUT. Four bits from a random pattern generator must be routed

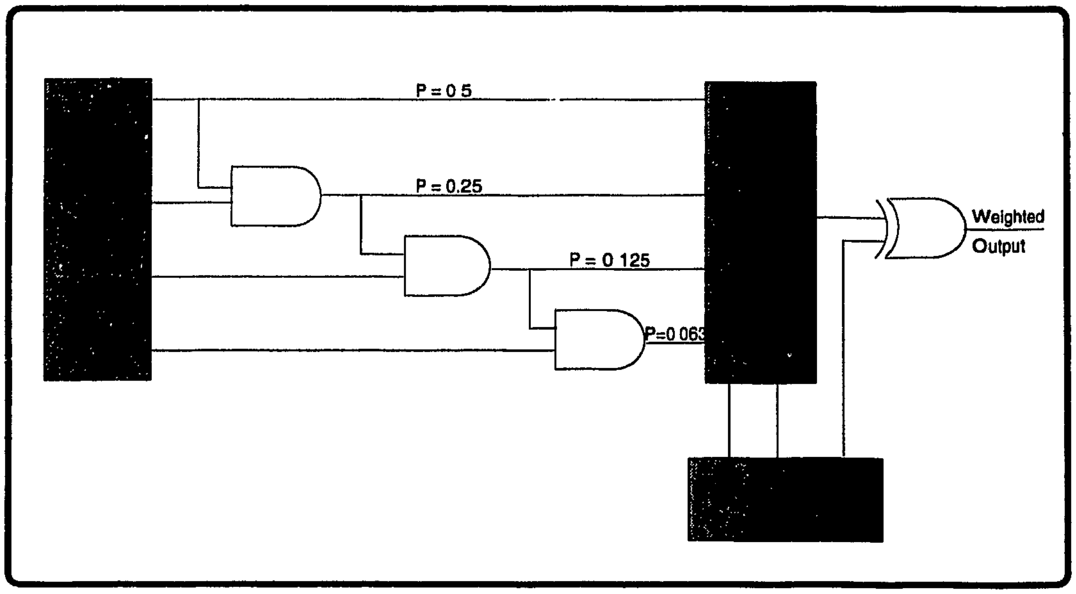


Figure 2.9: A Weight Generator

to each of the inputs. Using bits from neighbouring inputs may seem attractive, but a lot of care has to be taken to avoid correlations between inputs. Also, just one shift of the LFSR may not be sufficient to generate the next set of inputs, also because of correlations [Wun87]. Generally many shifts are performed, even to the extent of the length of the LFSR. Another method using GURT's (Generators of Unequiprobable Random Patterns) is presented in [Wun87].

Global generators generate all the weighted inputs in one spot and then shifted to the required inputs. Schemes for this are given papers such as [Brg89]. Care about correlations between inputs also have to be taken.

#### 2.9.5.2 Weights Based on Path Tracing

[Bar87] proposes a method whereby the signal probabilities of a given input (its weight) is calculated using a path tracing algorithm. This algorithm requires an initial assignment of probabilities of each gate based on theorems 1 and 2 below, and for each assignment, the probabilities are propagated back to the inputs according to the formulae given in table 2.3. After this is done for each gate in the circuit, the average of the calculated input probabilities are used as the weights.

The theorems are as follows:

**Theorem 1:** *The optimal signal probability assignments to the inputs of an AND or NAND gate with n inputs is* $\frac{n}{n+1}$ *for* $t = n + 1$ *and* $\frac{n-1}{n}$ *for t large.*

**Theorem 2:** *The optimal signal probability assignments to the inputs of an OR or NOR gate with n inputs is* $\frac{1}{n+1}$ *for* $t = n + 1$ *and* $\frac{1}{n}$ *for t large.*

| BLOCK | $P^i$ |
|---|---|
| AND | $P_o^{\frac{1}{k}}$ |
| OR | $1 - (1 - p_o)^{\frac{1}{k}}$ |
| INV | $1 - P_o$ |
| NAND | $(1 - P_o)^{\frac{1}{k}}$ |
| NOR | $P_o^{\frac{1}{k}}$ |
| FANOUT | Average of stem probabilities |

Table 2.3: Backtrace Signal Probabilities Update Formulae

**The Algorithm:**

**Step 1:** Assign to the inputs of the gate in question the associated probabilities for large t using Theorems 1 and 2.

**Step 2:** Moving backwards from the lines that have been assigned, calculate the input probabilities by recursively applying the formulae in table 2.3.

**Step 3:** Record the signal probability assignments computed for the primary inputs.

**Step 4:** Repeat steps 1 - 3 for all gates.

**Step 5:** At each primary input, assign weights equal to the average of all the recorded weights.

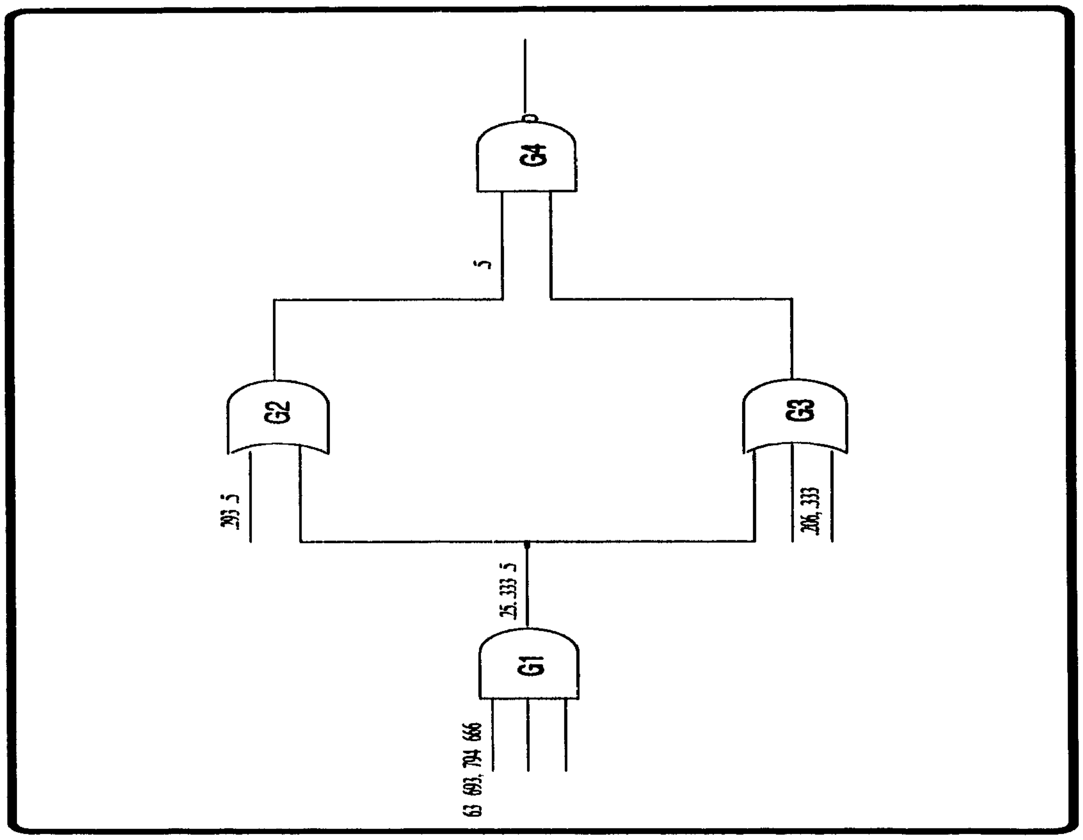


Figure 2.10: A Sample Circuit

As an example, the input weights of the circuit in figure 2.10 will be calculated.

Applying Step 1 to gate G1, the input probabilities to gate G1 become 0.5. This value is propagated back to gates G2 and G3. Appling Step 2 to gate G2, its input probabilities become 0.293. The same is done to Gate G3 giving its input probabilities as 0.206. The input to gate G4 is taken as the average of 0.293 and 0.206 as there is a fanout. This value is then propagated back to the inputs of gate G4 giving an input probability of 0.63.

This is repeated for each of the other gates giving the values indicated in figure 2.10. Now Step 5 is applied, and the averages at each input is calculated and this is used as the final weights.

This method is quite simple, but may not give the optimal probabilities as the relative 'importance' of each branch, when the averages are found, are not taken into account.

#### 2.9.5.3 Weights Based on Switching Characteristics

In contrast to Method 1 which gives weights referring to the probabilities of a '1' at the input, this method gives weights biased on the switching that occurs in the CUT due to a transition to a given input [Sch75].

In the generator in figure 2.11, eight outputs from a LFSR are fed into a 1 of 256 line decoder. This causes one of the outputs of the decoder to go high in a random manner. For Bit Changer 1, there is a $\frac{150}{256}$ chance that one of the outputs affecting it would cause a transition to Chip Input 1. Note that at most one input to the CUT changes at a time.

The weights (probabilities of a transition in an input to the CUT) are obtained using the following algorithm:

**Step 1:** Assign weights according to the relative importance on the inputs, ie. ad hoc.

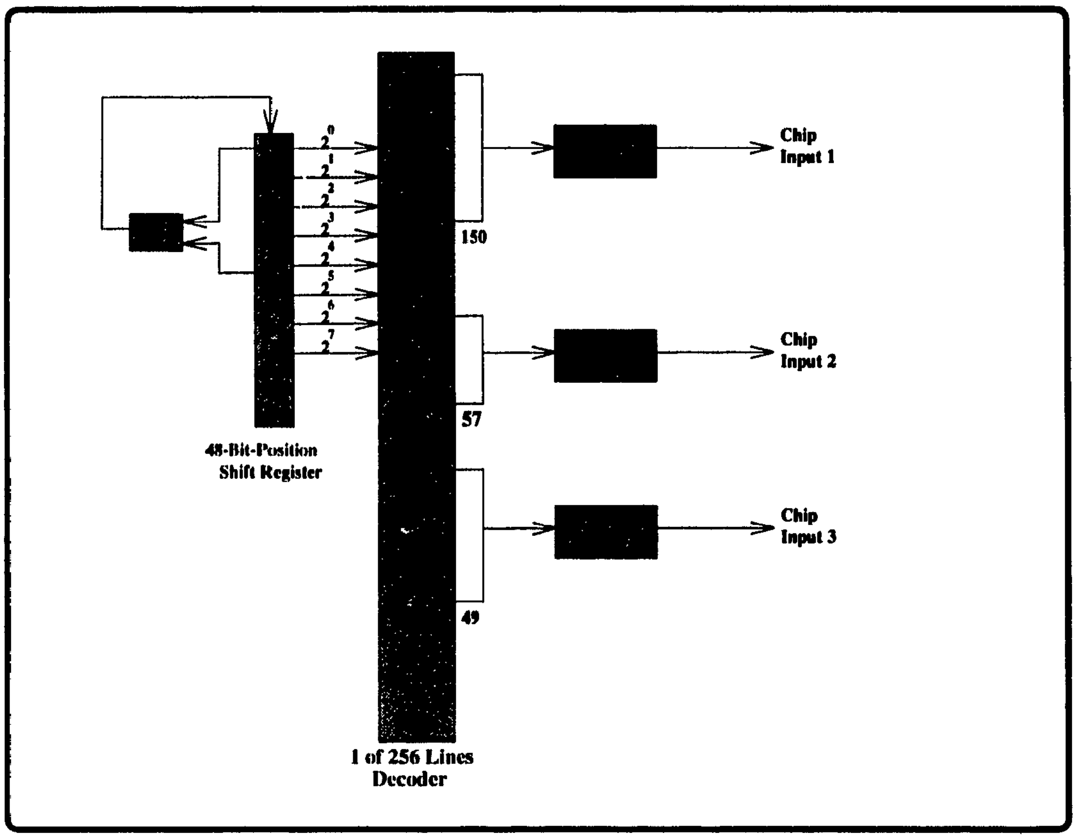


Figure 2.11: A Hardware Pattern Generator

**Step 2:** Simulate the circuit as driven by the hardware pattern generator remembering that only one input would be changing at a time.

**Step 3:** For each input, count the number of changes in previously unchanged nodes.

**Step 4:** Repeat steps 1, 2, and 3 until no more significant activity is observed.

**Step 5:** Weights are assigned based on the accumulated count in Step 3.

This system has one major failing. Such lines as the reset line has a high initial activity as it causes a lot of blocks to go into an initial state. This would cause a high initial count to be assigned this line, and hence a rather large weight. In a circuit such as a counter, this would be counter-productive as each toggle on the reset line would cause the counter to be reset thus resulting in an extremely low chance that the counter ever reaches to a high number, and that the overflow be used. In order to overcome this, a solution called *Dynamic Adaptation* [Sch73] is used.

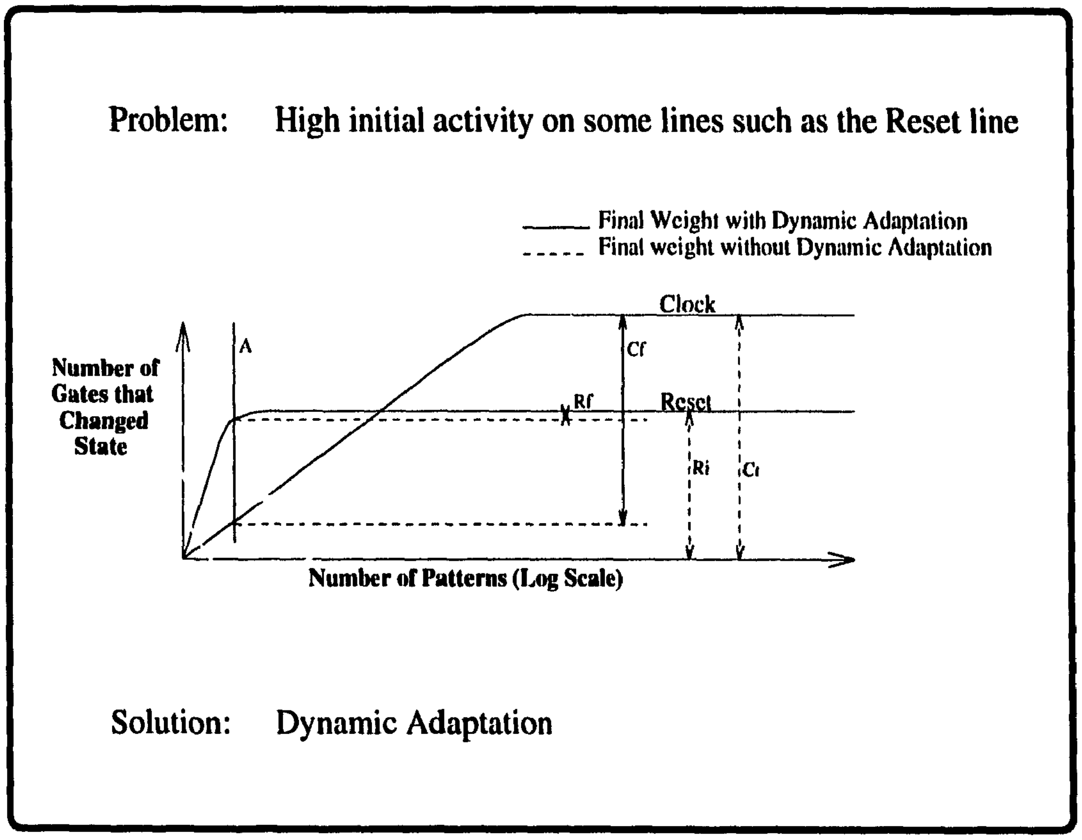

Problem: High initial activity on some lines such as the Reset line
Final Weight with Dynamic Adaptation
Final weight without Dynamic Adaptation
Clock
A
Cf
Rf
Reset
Ri
Ci
Number of Gates that Changed State
Number of Patterns (Log Scale)
Solution: Dynamic Adaptation

Figure 2.12: Clock/Reset Activity

Dynamic Adaptation is illustrated in figure 2.12 for the counter described above. In the original algorithm, all counts are taken from the first input vector. This gives the counts as $R_i$ for the reset line and $C_i$ for the clock. Note that the reset line initially shows the steeper increment in activity, and that the final count is significant when compared to the final count of the clock.

The approach in Dynamic Adaptation is to start the count at a later stage, after the 'transient' activity of the reset line has fallen below a predefined rate of increase, as shown by line A in the figure. At this time, all counts are restarted. After this time, the increase in the reset count would be small, given by $R_f$ while the count for the clock would still be quite large, $C_f$. This would therefore reduce the switching activity of the reset line. This handling does not necessarily presuppose prior identification of the functional characteristics of the inputs.

This technique was proposed in the mid-70's. It, however, is not very applicable to modern circuits due to the large number of circuit inputs resulting in the generation scheme being infeasible.

Some modern weight generation schemes do however keep some of the characteristics of this one, such as the measuring of the switching in the CUT due to a transition in an input. From this, weights can be generated. One simple way may be that an input which causes a large amount of switching in the circuit be given a weight close to 0.5, while those with low switching values be given weights close to 0 or 1 depending on whether it is preferable to have the input a '1' or a '0'.

#### 2.9.5.4 The ESPRIT Algorithm

This section will briefly cover the ESPRIT (Enhanced Statistical PRoduction of Test vectors) algorithm proposed by [Lis87]. In this approach, a function representing the CPU time required to get a desired fault coverage and the simulated test length is derived. This function **U**, shown in figure 2.13, is given in terms of the probabilities of detecting a fault $Pd_j$, and the number of undetected modeled faults **M**. The probability testability algorithm used was COP (Controllability/Observability Procedure) to obtain a set of input probabilities (weights).

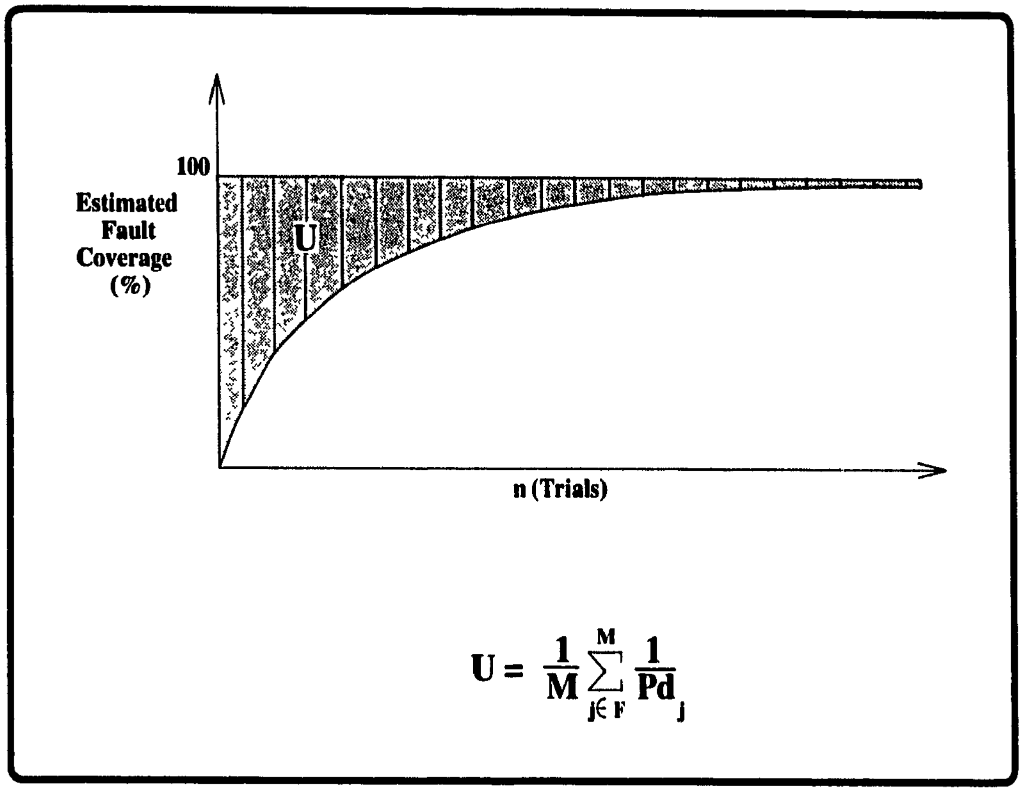


Figure 2.13: Area Cost Function

The main aim of this procedure is to use a steepest descent algorithm to minimize the cost function, **U**, in attaining a predefined threshold fault coverage. Supplemental weight distributions can be obtained by repeating this process on the remaining undetected faults.

### 2.9.5.5 A Single Weight Set Based on a Complete Test Set

In many of the above methods, several weights need to be used to get a significant reduction in test length. To accommodate these weights, a sizable amount of hardware is needed. This procedure [Mur90] attempts to calculate a single set of weights, which will be employed after a reasonable length of equiprobable random patterns have been used, to detect the random pattern resistant faults.

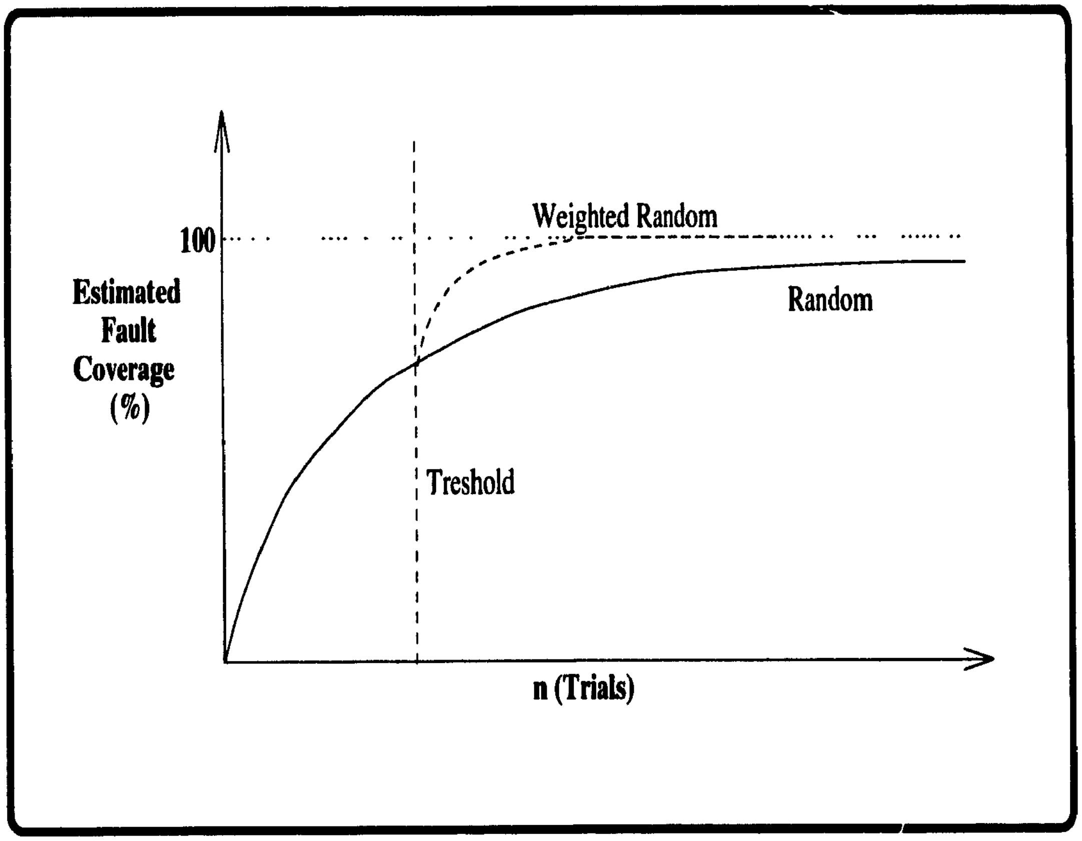


Figure 2.14: Expected Effect of Two Weight Sets

This effect is shown in figure 2.14 where the equiprobable random patterns have been used until the threshold indicated on the graph. If it were continued, note the long tail before it reaches close to 100%. At the threshold, when the weighted patterns are used, the curve rises quickly to 100%.

The innovative parts of this method involves the calculation of the weights In all the previous methods, the circuit itself was used to aid in the determination of weights. It is conjectured here that a pre-determined test set contains sufficient circuit information to calculate an efficient set of weights. The test set can be calculated using any available method, and hence obtaining weights can be a lot less CPU intensive than existing methods.

Before presenting the algorithm employed to calculate the weights, a few necessary concepts should be defined, *bit flipping* and *weight relaxation.*

*Bit flipping* is used to get rid of unnecessary assignments in the vectors of the test set. The number of faults required to be covered by each vector is reduced as much as possible constrained to covering all the required faults. After this reduction, a number of the bits in the vectors do not affect the coverage of that vector thus can be deemed unneeded and ignored. Unnecessary bits are determined heuristically by flipping each bit one at a time. If all the required faults are still covered, that bit is inessential.

| Vector | Before Bit Flipping | | | | | | After Bit Flipping | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $v_1$ | 1 | 1 | 1 | 0 | 0 | 1 | x | 1 | 1 | x | x | x |
| $v_2$ | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | x | 1 | x | x |
| $v_3$ | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | 0 |
| $v_4$ | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | x | x | x |
| Weight | 0.5 | 1 | 0.5 | 0.25 | 0 | 0.75 | 0.33 | 1 | 0.5 | 1 | 0.5 | 0 |

Table 2.4: Bit Flipping

The effect of removing these unneeded assignments is demonstrated in table 2.4. Weights are calculated by finding the fraction of '1' assignments to a given position. Before bit flipping is performed, the unnecessary bit positions act as 'noise' corrupting the weight estimates. This can be seen in table 2.4, in the sixth position for example, before bit flipping, the weight was found to be .75. Through this procedure, it is found that a 0 weight is better as all the '1' assignments are unnecessary.

After a suitable weight is found, some of the remaining faults may still be hard to test with these weights. In such cases, a new set of weights are calculated using the remaining vectors. These weights are compared as in table 2.5, and if bit positions differ by a large amount, the weight is relaxed to, say, 0.5. This is a potentially dangerous move, but if it is not done too frequently so that the number of positions is small, the effect can be minimized. This is referred to as *weight relaxation.*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Original Weights | 1 | 0 | 0.2 | 0.7 | 0.8 | 0.2 | 0.1 |
| New Weights | 1 | 0 | 0.4 | 0.8 | 0.1 | 0.9 | 0.3 |
| Modified Weights | 1 | 0 | 0.2 | 0.7 | 0.5 | 0.5 | 0.1 |

Table 2.5: Weight Relaxation

An algorithm for calculating the weights is given in figure 2.15.

### 2.9.6 Other Random Pattern Based Generators

Due to the large hardware overhead and the cost of generation of multiple weights, classical weighted random pattern generation has been slow to catch on. This is already evident in the system given in section 2.9.5.5 where attempts were made to limit the number of weights to one. However, two new and interesting methods proposed in [Pat91a, Pat91b] which attain similar coverage as weighted random patterns but with less hardware will be described in this section. Both methods will be described with respect to the random pattern resistant circuit given in figure 2.16. As with method 4 above, all generation is done on a test set, and not on the actual CUT for the same reasons given.

#### 2.9.6.1 Correlated Random Pattern Testing

A complete test set for the circuit in figure 2.16 is given in figure 2.17(a). An imaginary input value is added (figure 2.17(b) in bold) and the correlation of each bit value in the vector and the corresponding value in the imaginary input is calculated. Here, each bit is the same value as the extra input $\frac{8}{9}$ of the time. Thus it seems reasonable to

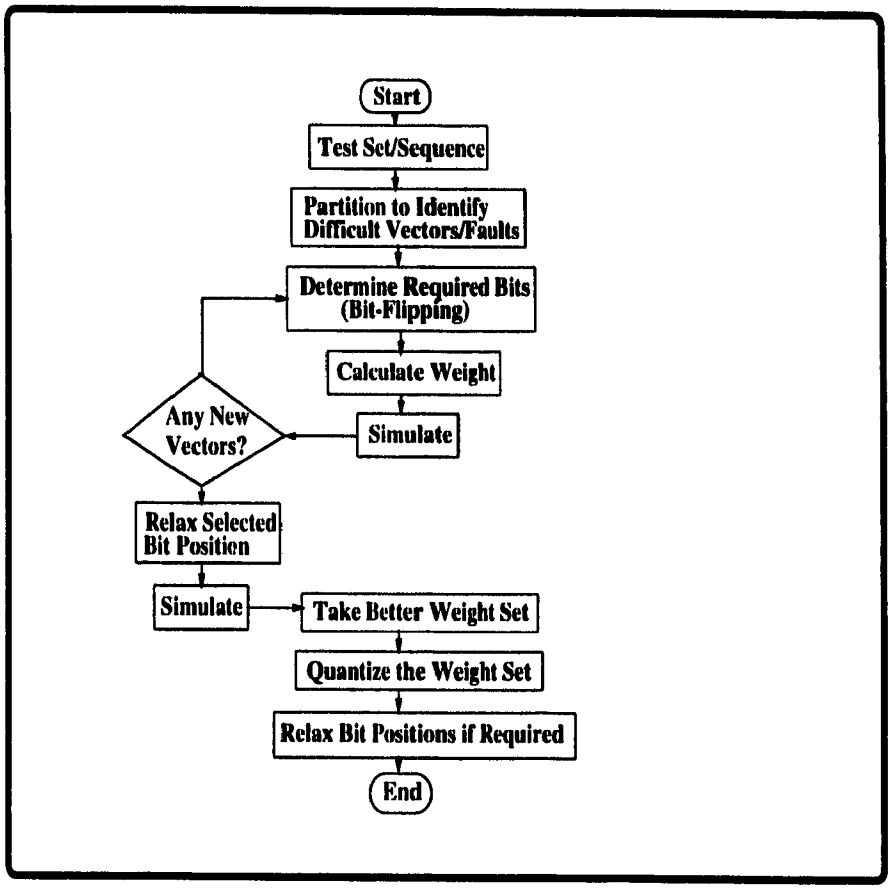

Start
Test Set/Sequence
Partition to Identify Difficult Vectors/Faults
Determine Required Bits (Bit-Flipping)
Calculate Weight
Simulate
Any New Vectors?
Relax Selected Bit Position
Simulate
Take Better Weight Set
Quantize the Weight Set
Relax Bit Positions if Required
End

Figure 2.15: Flow Chart of the Weight Estimation Method

generate all inputs using this one common random source. We thus get the generating circuit in figure 2.18. Here, $P_1^1 \ldots P_1^8$ are used to invert the value of $P_1$ via the XOR gate. Hence these probabilities should be $1 - \frac{8}{9}$.

This method is not limited to one independent input, nor just to imaginary inputs to which the others are correlated to. A real input can be used for this purpose.

Experimentally, the average random pattern test length needed for the circuit is

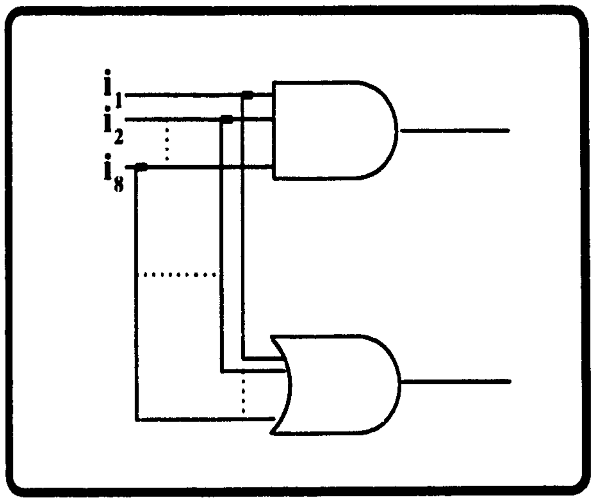


Figure 2.16: A Random Pattern Resistant Circuit

860. Using Correlations, this average length is reduced to only 120, less than the average number of 150 needed with two weights.

| | | |
|---|---|---|
| 00000000 | 0 | 00000000 |
| 10000000 | 0 | 10000000 |
| 01000000 | 0 | 01000000 |
| 00100000 | 0 | 00100000 |
| 00010000 | 0 | 00010000 |
| 00001000 | 0 | 00001000 |
| 00000100 | 0 | 00000100 |
| 00000010 | 0 | 00000010 |
| 00000001 | 0 | 00000001 |
| 11111111 | 1 | 11111111 |
| 01111111 | 1 | 01111111 |
| 10111111 | 1 | 10111111 |
| 11011111 | 1 | 11011111 |
| 11101111 | 1 | 11101111 |
| 11110111 | 1 | 11110111 |
| 11111011 | 1 | 11111011 |
| 11111101 | 1 | 11111101 |
| 11111110 | 1 | 11111110 |
| (a) | | (b) |

Figure 2.17: (a) A Complete Test Set for the Random Pattern Resistant Circuit above, (b) Test Set with an Imaginary Input Value Added to each Input

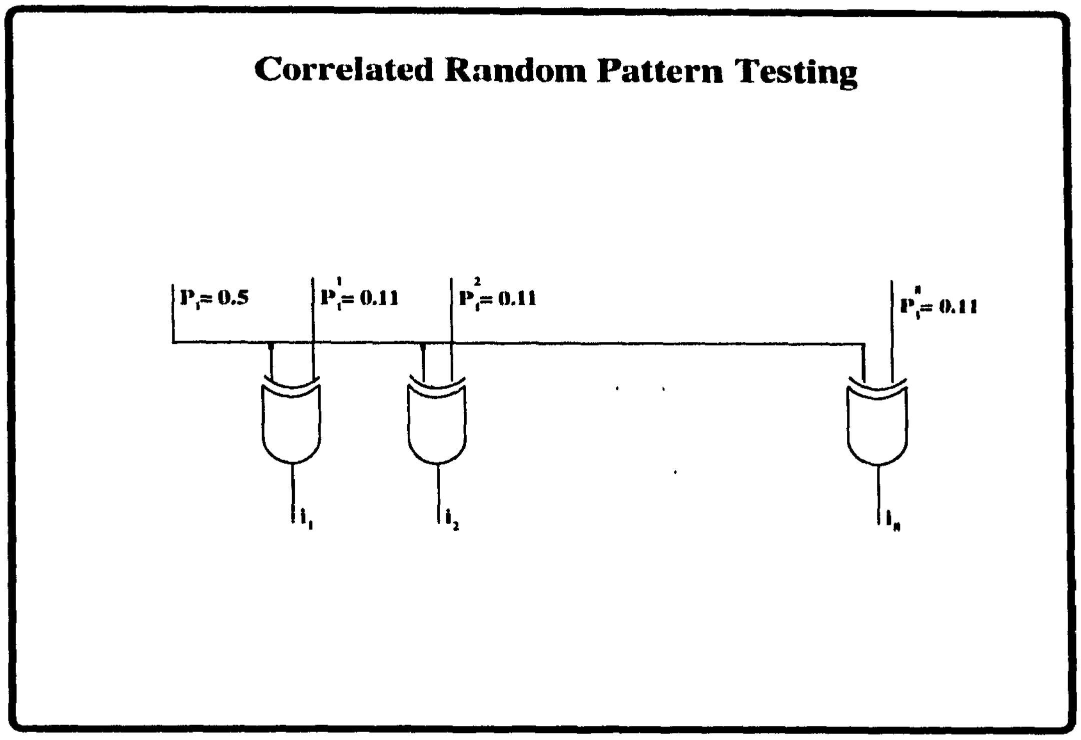


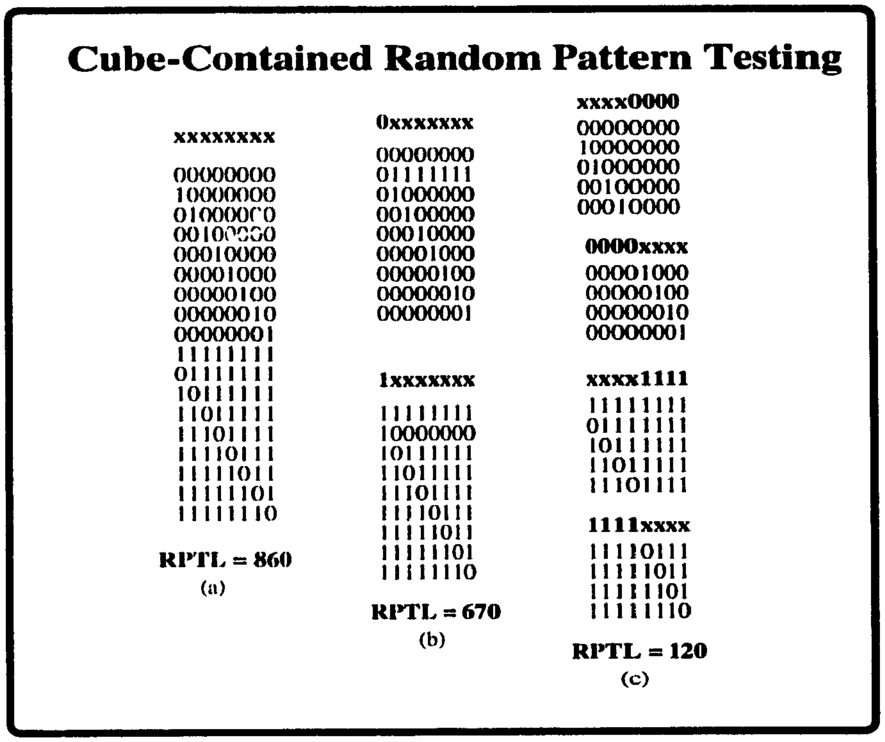
Figure 2.18: Generation of Correlated Random Patterns

#### 2.9.6.2 Cube Contained Random Pattern Testing

Correlated random pattern testing described in the above section does not really address the problem of the amount of extra hardware required as a correlation probability for each input must be generated, and thus it requires the same amount of overhead as a weighted random pattern test with a comparable number of weights.

Cube contained testing [Pat91b] tends to reduce the hardware overhead as each bit assignment is either a fixed value or is chosen with a 0.5 probability, totally eliminating the circuitry for weighting the probabilities.

The aim of this method is to partition the test set into sets where several bit positions are identical for all vectors. In generating the test vectors, these positions become fixed while those with differences are randomly chosen.



**Cube-Contained Random Pattern Testing**

| (a) | (b) | (c) |
|---|---|---|
| **xxxxxxxx** | **0xxxxxxx** | **xxxx0000** |
| 00000000 | 00000000 | 00000000 |
| 10000000 | 01111111 | 10000000 |
| 01000000 | 01000000 | 01000000 |
| 00100000 | 00100000 | 00100000 |
| 00010000 | 00010000 | 00010000 |
| 00001000 | 00001000 | **0000xxxx** |
| 00000100 | 00000100 | 00001000 |
| 00000010 | 00000010 | 00000100 |
| 00000001 | 00000001 | 00000010 |
| 11111111 | **1xxxxxxx** | 00000001 |
| 01111111 | 11111111 | **xxxx1111** |
| 10111111 | 10000000 | 11111111 |
| 11011111 | 10111111 | 01111111 |
| 11101111 | 11011111 | 10111111 |
| 11110111 | 11101111 | 11011111 |
| 11111011 | 11110111 | 11101111 |
| 11111101 | 11111011 | **1111xxxx** |
| 11111110 | 11111101 | 11110111 |
| | 11111110 | 11111011 |
| | | 11111101 |
| | | 11111110 |
| **RPTL = 860** | **RPTL = 670** | **RPTL = 120** |
| (a) | (b) | (c) |

Figure 2.19: Partitioning a Test Set to Reduce the Random Pattern Test Length

This is shown again for the circuit in figure 2.16. Once again the complete test set is given, in figure 2.19(a). It is then split into two partitions in figure 2.19(b), based on only the leading bit being identical. This reduces the test length from 860 to 670. On partitioning the set into four with four inputs specified, this average length falls to 120, the same as in correlated testing.

# Chapter 3

# Intelligent Reseeding of LFSRs

The test generation methods presented in section 2.9 are all stretched or fall short in one or more of the many, sometimes conflicting parameters which gauge the effectiveness and cost of test. Coverage of stuck-at faults is guaranteed through exhaustive testing as all possible relevant input combinations for each output are used [Bar87], but this method quickly becomes infeasible for circuits with outputs dependent on many inputs. Deterministic testing involves the application of a stored set of vectors resulting in a high fault coverage in a short test application time. However, storage and bandwidth considerations can become costly. The advantages of random based methods lie in its ability to attain fairly high coverage of a fault set with minimal stored information. However, with the raised requirements demanded by new circuits, 95%, or even 99% fault coverage obtained by these methods in reasonable test times is no longer adequate. In fact, close to 100% coverage of non-redundant faults is now required.

Weighted random pattern testing has been quite successful in this respect, but with at least four stored bits per bit in the scan chain, this can lead to excessive memory requirements for complete tests. In addition to this, the hardware overhead needed to produce the probabilities for the weights can be substantial.

None of these methods take advantage of the fact that a large number of input bits can be inverted without a loss of coverage and hence their values are unnecessary and need not be explicitly stored. In 1991, [Kon91] introduced an innovative scheme for deterministic testing with storage requirements comparable to that of weighted random pattern testing for a 100% coverage but using only a fraction of the on-chip pattern generation hardware overhead. This method, termed ***intelligent LFSR reseeding***, involves a mixed mode paradigm for testing with the initial application of a number of pseudo random patterns, through an LSFR, to achieve an initial coverage of 90% - 95% of the faults (generally the easy to detect ones). Following this, a set of stored seeds would be loaded into the LFSR, each resulting in a pre-determined pattern being applied to the circuit. It was found that to encode a test cube with $s$ specified bits, at least $s + 19$ bits must be stored (as the seed) to keep the probability of not finding an encoding below $10^{-6}$.

[Hel92] followed by [Ven93] further refine the process resulting in just $s$ stored bits to encode test cubes with at most $s$ specified bits while keeping the probability of not finding an encoding below $10^{-6}$. They accomplish this through using multiple-polynomial LFSRs and through implicit assignment of test cubes to polynomials.

## 3.1 Reseeding of Multiple-Polynomial LFSRs

Figure 3.1 illustrates the architecture proposed in [Ven93] for decoding the stored information into test vectors which will be applied to the circuit under test (CUT). An $m$-bit test cube is encoded into an $s$-bit word which is used as a seed to an LFSR. The position of the seed in the memory denotes which feedback polynomial is chosen. Generation of a test vector involves the submission of the next available seed from the memory and the reconfiguration of the feedback links of the LFSR according to the modulo $p$ counter and the decoding logic. Clocking the LFSR for $m$ cycles produces the required vector.

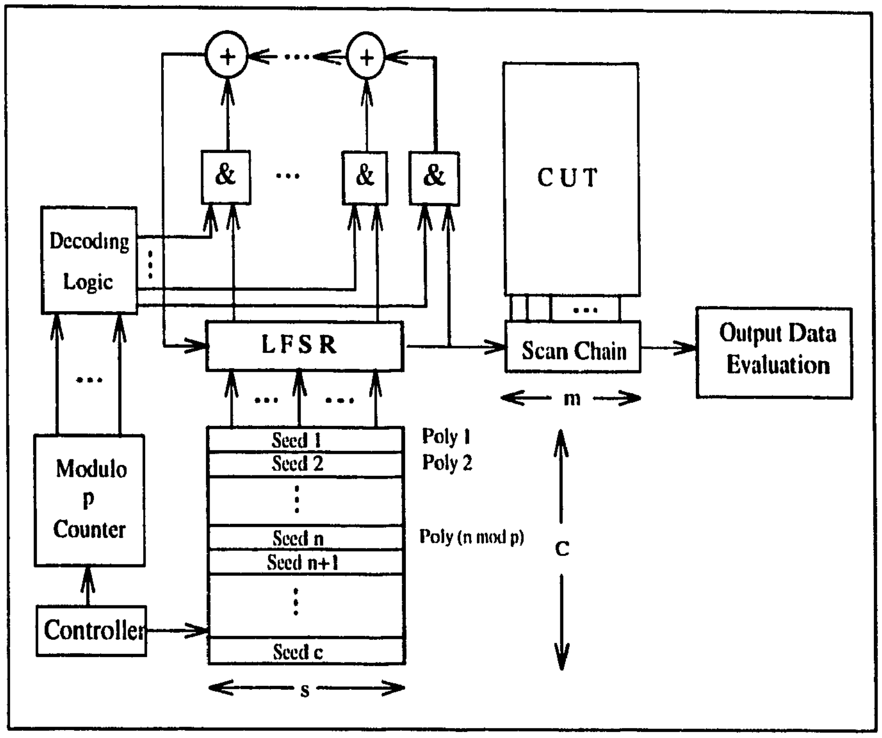


Figure 3.1: Scheme Based on Multiple-Polynomial LFSRs

A test cube $C = (c_0, ..., c_{m-1}) \in \{0, 1, x\}^m$ is consistent with an LFSR output sequence $(a_i)_{i \geq 0}$ provided $c_i = a_i$ for all specified bit positions $\{i = 1, ..., m | c_i \neq 'x'\}$. Let $S(C)$ be the number of specified bits in $C$. For each polynomial, given a test cube, C, a seed can be found through solving a set of $S(C)$ linear equations provided a solution can be found. These equations are found by recursively equating the appropriate feedback equation to $c_i$ when $c_i \neq 'x'$. Though this set of equations appear to be always solvable because there are less than or the same number of equations as there are variables, this is not always the case through the dependencies imposed by the feedback equation of the LFSR. The following example illustrates such a case:

**Example 1:** A test cube $C := (x, x, 1, 0, x, 0, x) \in 0, 1, x^7$ is to be generated by a 3-stage LFSR. The output sequence depends on the seed $a(0) = (a_0, a_1, a_2)$ and the feedback polynomial, $\mathcal{H}(X) = X^3 + X + 1$. The following set of equations are obtained:

i $1 = a_2$

ii $0 = a_0 + a_1$

iii $0 = a_0 + a_1 + a_2$

In general, linear independence of the equations is a sufficient but not a necessary condition of a solution. For example, the above system of linear equations has a solution for $a_2 = 1$, $a_3 = 1$ and $a_5 = 0$.

[Hel92] evaluates the probabilities of an encoding not being found of the two extremes; fully programmable polynomials with fixed seeds, and a single polynomial with arbitrary seeds.

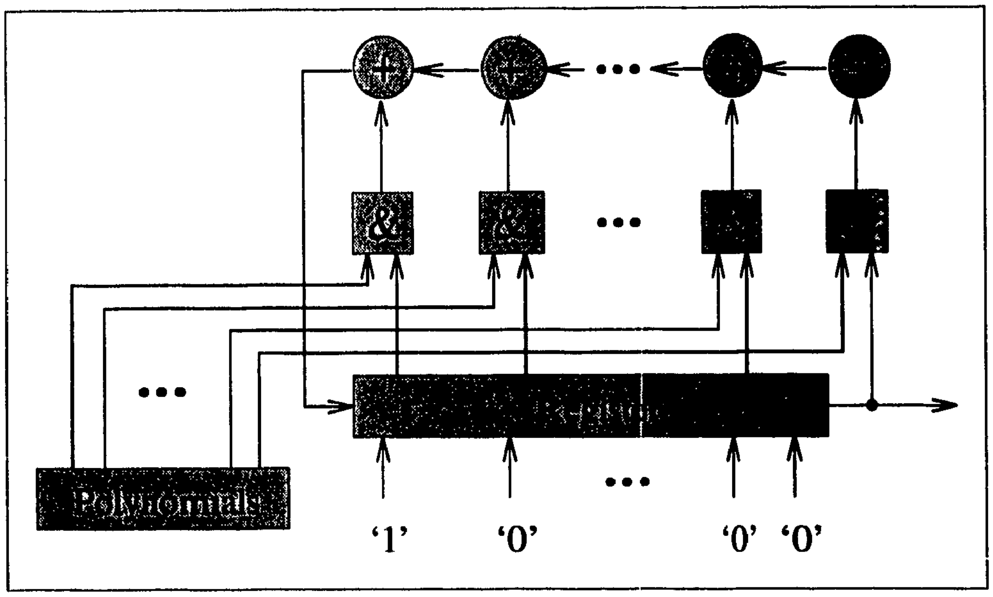


Figure 3.2: Scheme Based on Fully Programmable Polynomials

Figure 3.2 shows a scheme based on fully programmable polynomials with a fixed seed. In this example, a constant seed, $[1, 0, 0, ..., 0]$, is used to generate the vector. The feedback of the LSFR is fully reconfigurable and is designated through the polynomials. The probability of not finding an encoding, $P_{nopol}(k, s)$, using a polynomial of degree

$k$ for a sequence of length $m$ with $s$ specified bits is given by

$$P_{noplo}(k,s) = \left(\frac{2^m - 2^k}{2^m}\right)^{2^{m-s}} \tag{3.1}$$

$$\approx (e^{-1})^{2^{k-s}} \text{ for sufficiently large } m \tag{3.2}$$

Thus, the probability of not finding an encoding is, for practical applications, independent of the length of the sequence, $m$, and depends only on $k - s$. For $P_{nopol} \leq 10^{-6}$, $k \geq s+4$. This is the best possible encoding examined, but it is computationally costly as it requires the solution of a system of non-linear equations.

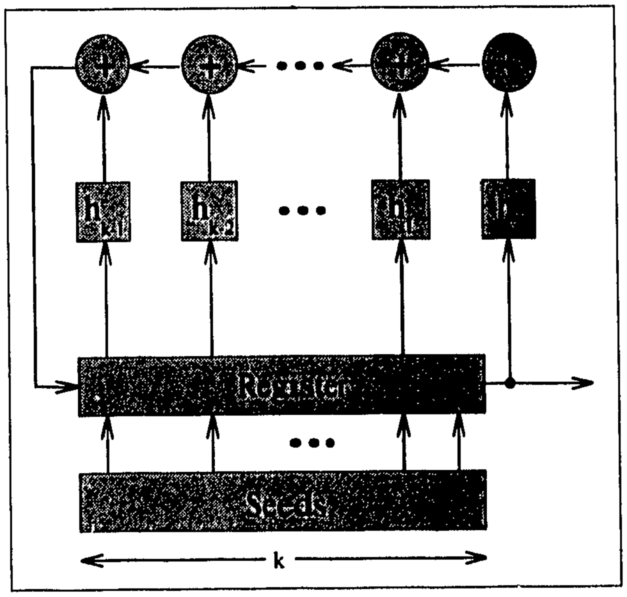


Figure 3.3: Scheme Based on the Reseeding of Single Polynomial LFSRs

An implementation predicated on the other extreme, the reseeding of a single polynomial LFSR, is given in figure 3.3. A fully programmable vector is used to seed an LFSR with a fixed feedback, denoted by $\mathcal{H} = X^k + h_{k-1}X^{k-1} + \ldots + h_1 X + h_0$. Here too, an encoding is not assured. The probability of no seed being found, $P_{noseed}(k,s)$ is approximated by

$$P_{noseed}(k,s) \approx 2^{s-k+1} \text{ for } s \ll k \tag{3.3}$$

For $P_{noseed}(k,s) \leq 10^{-6}$, this equation gives $k = s + 21$. In fact, calculating the exact values for $P_{noseed}(k,s)$ gives $k = s + 19$. Though the result is not as compact as in the case of full polynomial flexibility, the seed is found as a solution of a set of linear equations thus is computationally tractable.

The authors then examined trade-offs to attempt to find a scheme which possesses the benefits of both, compaction comparable to the former method with computation needs comparable to the latter. It was found that with 16 polynomials, $k = s + 4$, the probability that no encoding can be found is less than $10^{-6}$ achieving the encoding efficiency of full programmability while maintaining the computational simplicity of single reseeding.

[Ven93] proposes an implementation for test sets where the encoding of the feedback polynomial is done implicitly by ordering the test cubes. Figure 3.1 shows the general structure of such a scheme. With this implementation, maximum encoding efficiency in attained only if the cubes are evenly distributed among the polynomials. In cases where this is not possible, dummy cubes must be introduced to balance it resulting in lost memory and increased test application time.

# Chapter 4

# Clustering Test Cubes

The approach adopted here to test a circuit is based on a mixed mode form of testing where a number of random patterns would be generated followed by a set of deterministic vectors generated through clustering. The first stage effectively removes most of the easy faults leaving the harder to test faults for the second.

*Clustering* or grouping of related test cubes is the process whereby cubes that have much in common are collected into sets each of which can be represented by a single cube plus a small amount of individual information which is used to recreate the component vectors. Though similar to the method in [Ven93], this technique is fundamentally different in that it allows a small number of inconsistencies between the cubes in a cluster. It is based on an observation that test cubes for hard to detect faults do not occur randomly but tend to form in clusters. This tendency may be as a result of the following:

- In a test cube, all the specified bits are necessary assignments, thus the requirement of a similar condition for another fault would be satisfied by these same specified bits.

- If a hard to detect fault has a very difficult setup or propagation condition, any fault which requires a similar condition either as part of its setup or propagation conditions would have many common bits. A common setup condition may occur in case of a fanout at the line while a common propagation condition due to a fan-in to that line.

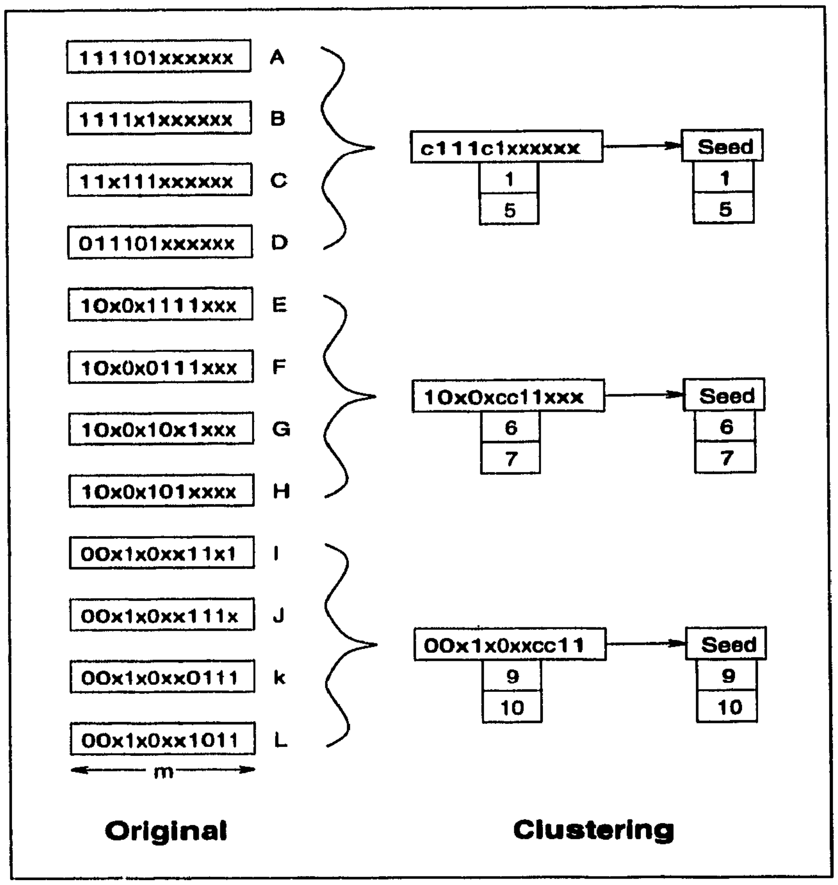


Figure 4.1: The Process of Clustering

Figure 4.1 shows the aims of clustering while figure 4.2 gives a conceptual representation of the process consistent with these test cubes. The original test cubes (shaded circles) fall in three main locations with {A, B, C, D} in the first, {E, F, G, H} in the second and {I, J, K, L} in the last. One thrust of this work is to identify these

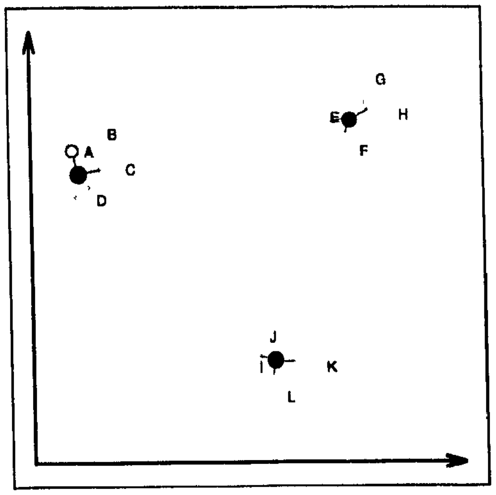


Figure 4.2: Conceptual Representation of Clustering

groups and all the constituent cubes, and to find suitable compromise cubes (solid circles) which embodies the common information within the clusters. In addition to this, adjustments (solid lines) must also be established which maps these constituent cubes into each of the component cubes. A second area of activity is in the coding of adjustments that are needed, and in the determination of the type of structure and hardware necessary to exploit the multiplicity, correlation and locality of the information.

For example, consider a case where the scan chain is 1000 bits long, and 3 hard to detect faults, represented by test cubes with at most 80 specified bits, must be tested. Furthermore, assume that these cubes are very similar but have 1 conflict when any two are paired. Conventional methods of deterministic testing would require 3000 bits of storage while the reseeding method needs 240 bits. Clustering, however, only requires about 100 bits, 80 for 1 seed, 10 for each of 2 conflicts and a few bits for control.

Before continuing, the mathematics employed in this process will be summarized. The input is made up of test cubes which are ordered sets of the elements {0, 1, x} representing the two specified values and the don't care. The output is another ordered set of the elements {0, 1, x, c} where the first three are the same as before while the 'c' indicates a *conflict* or a position for which at least one of the component cubes specifies the bit set while at least one other specifies the converse. When cubes are combined, $\cap_c$, into clusters, the result is a cube which represents the intersection of the spaces once there are no conflicts. In cases where there are conflicts, a 'c' is introduced in that position. This is illustrated in figure 4.3.

| $\cap_c$ | 0 | 1 | x | c |
|---|---|---|---|---|
| 0 | 0 | c | 0 | c |
| 1 | c | 1 | 1 | c |
| x | 0 | 1 | x | c |
| c | c | c | c | c |

Figure 4.3: The Function, $\cap_c$

The effectiveness of clustering depends on the amount of memory required to create a complete test compared to that of reseeding. In reseeding, each test cube is compressed into a seed of length approximately that of the number of specified bits in the cube, and on occasion, multiple cubes can be intersected resulting in fewer seeds and a shorter test length. For clustering, cubes with many similarities are grouped together with the common information represented by a single test cube and the differences by conflict information. A seed is constructed, as in reseeding, for this cube. To contrast the two cases, the hardware structure described in section 4.2.1 (Direct Implementation) is assumed. Thus, for compaction, the conflicts in the resultant test cubes (figure 4.1) can be viewed as 'x's.

In this example, there are 3 intersections which can be done for both methods reducing the number of seeds and the test length from 12 to 9. To compare the necessary memory, let $m$ be the length of the scan chain, $s$ the length of a seed, $Clst$ the total number of clusters with $Clst_i$ being the number of clusters with $i$ conflicts, and let $TL$ be the test length. For clustering, let the maximum number of conflicts allowed (the *conflict count*) be $Confl_M$. For reseeding, the memory required is given by:

$$Memory = 9 * s \; bits \tag{4.1}$$

Rewriting equation (4.6) obtained using the direct implementation scheme as equation (4.2), and then substituting for values:

$$\begin{aligned} mem &= [Clst * s] + [(TL - Clst_0 - 2 * Clst_1) * (Confl_M + 1)] + [Confl * \lceil log_2(m) \rceil] & (4.2) \\ &= 3 * s + 6 * \lceil log(m) \rceil + 9 * (Confl_M + 1) \; bits & (4.3) \end{aligned}$$

$$2 * s \geq 2 * \lceil log_2(m) \rceil + 3 * Confl_M + 3 \tag{4.4}$$

Equating (4.1) and (4.3) to get (4.4), clustering is preferred when the size of the seed is large compared to the memory needed to store the conflicts.

Though this method does not concentrate on reducing the number of vectors which must be applied to a circuit, often there is an overlap (like {A, B} and {G, H} in figure 4.2) and a cube in this common area is selected effectively reducing the number of vectors in the test. In the above example, both methods resulted in a test of the same length. Typically though, reseeding results in shorter test lengths.

## 4.1 The Clustering Algorithms

These algorithms are based on greedy methods where, based on a weight function, the most favorable decisions are made at each step. Before going into a detailed description of these processes, certain key elements should be presented. Those which give a measure on the effect of a given merger are illustrated in figure 4.4. One is the *conflict count* which indicates the number of conflicts that occur between the constituent cubes making up the cluster. Another is the specified bit count which indicates the number of specified bits (0,1) in the cluster. Note that this excludes the conflicts. A parameter which aids in the choice of the next clusters to combine is the ***extra specified bit count*** which denotes the increase in the specified bit count after the combination. It gives an indication of the similarity between the cubes.

| | Conflict Count | Specified Bit Count | Extra Specified Bit Count |
|---|---|---|---|
| 1001xx1xxc | 1 | 5 | - |
| $\cap_c$ | | | |
| 100cx100xx | 1 | 6 | - |
| 100cx1c0xc | 3 | 5 | -1 |

Figure 4.4: Merging of Two Test cubes

Another element is the concept of *partitions* which has an effect on the choice of the next combination by limiting the scope of possible combinations. By constraining the effect of non-related cubes on the weighing functions, results are often better when the partitions are well chosen. Also, it speeds up the process for the searching is confined to smaller groups where the best choice is most likely to be. The input set of test cubes is initially partitioned based on conflict and extra specified bit counts. Two cubes are in different partitions provided that there are no pairs of cubes, one from each, whose result when combined has conflict and extra specified bit counts both not greater than those specified for the partition. Partitioning the test set does not preclude clustering between cubes in different partitions. This is due to the extra specified bit count limit which may prevent two cubes from being in the same partition, but it may be advantageous to cluster them. Hence, extra processing is required at the end to search for these mergings.

## 4.1.1 The Single Cluster Algorithm

This method is based on the greedy principle that the largest cluster in the set would likely be a good choice in the result. Thus only a single cluster is being formed at any given time, and once finished it is not affected until the end. The pseudocode for the algorithm is given in Fig. 4.5. The first step in this process is to divide the test set into partitions.

```
SINGLE_CLUSTER()
    /* Given $P = \{P_i\}$; $P_i = \{C_{i,j}\}$ of testcubes. */
begin
    $T := \emptyset$                                /* Test cubes to be returned */
    for each ($P_i \in P$) begin
        $R := \emptyset$                            /* Used vectors */
        while ($R \neq P_i$) begin
            $Q := P_i \backslash R$                 /* Remaining vectors to be considered */
            $v := \{x, ..., x\}$                    /* Test cube representing this cluster */
            while ($Q \neq \emptyset$) begin
                $\{Q, v, R\} := restrict(Q, v, R)$  /* Get next vector to add to the cluster */
            end while
            $T := T \cup \{v\}$                     /* Save test cube */
        end while
    end for
    $T := cleanup(T)$                               /* Check for clustering between partitions */
    return $T$
end
```

Figure 4.5: The Single Clustering Algorithm

The algorithm then sets the working set ($Q$) to be the current partition. The **restrict()** function (figure 4.6) then finds, in $Q$, a cube whose result when merged ($\cap_c$) with $v$ can be merged with most of the other cubes without exceeding the conflict

```
Restrict($Q$, $v$, $R$)

begin
    for each ($e_i \in Q$) begin                         /* Take each vector in Q one at a time */
        $v_i := e_i \cap_i v$                            /* Find its result with $v$ */
        for each ($C_j \in Q$) begin
            $Q_i := \emptyset$                           /* And use it to find how many other vectors */
            $u := c_j \cap_i v_i$                        /* can be merged with it without exceeding*/
            if({no_confl($u$) $\leq$ conflict_count} &&
              no_spec_bits($u$) $\leq$ max_spec_bit)     /* the constraints. */
                $Q_i := Q_i \cup \{c_j\}$
        end for
    end for
    find $i$ : $|R_i|$ is maximal                        /* Find the largest group */
    return $Q_i$, $v_i$, $R \cup \{e_i\}$)               /* and return it. */
end
```

Figure 4.6: The Restrict() Procedure

and the specified bit counts. This result becomes $v$ while $Q$ is restricted to the other cubes which can be successfully merged with it. The newly chosen cube is then added to $R$, the set of merged cubes. This iterates until $Q$ is empty, and the new cluster formed in $v$ added to the set of test cubes. $Q$ is then set to the current partition less the used cubes ($Q := P \backslash R$ in the algorithm where this means "let $Q$ be the set of vectors in $P$ but not in $R$"), $v$ is cleared, and the process repeats until all the cubes have been processed. This is repeated for all partitions.

When there are no more partitions left, a **clean-up()** function is initiated whereby any clusters which can be merged without introducing conflicts are grouped. This is done by applying the process described above to the set of test cubes in one partition.

## 4.1.2 The Multiple Cluster Algorithm

In this process, the pair of cubes deemed most favourable to merge in the set in question is identified and merged. Thus, at any time, multiple clusters are in the process of being

```
MULTIPLE_CLUSTER()
    /* Given $P = \{P_i\}$; $P_i = \{C_{i,j}\}$ of testcubes. */

begin
    $T := \emptyset$                                            /* Test cubes to be returned */
    for each $(P_i \in P)$ begin
        while[$(\{C_{i,1}, C_{i,2}\} :=$ find_best$(P_i)) \neq$ NULL] begin   /* Get best pair of cubes to cluster */
            $C := C_{i,1} \cap_c C_{i,2}$
            $P_i := P_i \backslash \{C_{i,1}, C_{i,2}\} \cup C$              /* Remove used cubes and add new cluster */
        end while
        $T := T \cup P_i$                                       /* Save test cubes */
    end for
    while[$(\{C_{T,1}, C_{T,2}\} :=$ find_best$(T)) \neq$ NULL] begin      /* Check for clustering between partitions */
        $C := C_{T,1} \cap_c C_{T,2}$
        $T := T \backslash \{C_{T,1}, C_{T,2}\} \cup C$
    end while
    return $T$
end
```

Figure 4.7: The Multiple Clustering Algorithm

```
find_best($P$)

begin
    weight := 0                                            /* Initialize weight */
    for each $(c_i \in P)$ begin                            /* For every pair of vectors */
        for each $(C_j \in P; j > i)$ begin
            if (weight $<$ weight_of$(c_i \cap_c c_j)$) begin
                weight = weight_of$(c_i \cap_c c_j)$            /* If it is a better choice */
                $I = i, J = j$                                  /* Save it */
            end if
        end for
    end for
    return $(c_I, c_J)$                                     /* Return best choice. */
end
```

Figure 4.8: The find_best() Procedure

formed. As previously, the first step is to partition the cubes.

This algorithm is outlined in figure 4.7. The **find_best()** function (figure 4.8) goes through all the relevant clusters and chooses the pair with the most favorable weight and returns them (or NULL if there are none). This weight is determined primarily by the conflict count, followed by the extra specified bit count and then by the specified bit count. These cubes are then removed from the partition and the result of their clustering is added. This continues until there are no more beneficial merges remaining at which point the next partition is considered.

The first **while** loop clusters all the cubes in each partition while the second is needed in case combinations are possible on clusters between partitions but were not possible before, analogous to the clean-up phase in the previous algorithm.

## 4.2 Encoding of Test vectors and Conflict Information

This aspect of the work involves compressing and encoding conflict information in forms which can be easily decoded and used when required. A number of alternatives will be presented which reduce storage demands with a small penalty in hardware overhead and the amount of vectors supplied. However, these extra vectors may be regarded as additional random patterns.

In many of the following subsections, equations indicating the amount of memory are derived, all starting from the point:

$$memory(in\ bits) = [A] + [B] + [C] \tag{4.5}$$

where $[A]$ represents the component due to the storage of the seeds, $[B]$ represents the component due to indicating which conflicts are involved at this particular time and

[C] the component necessary for the conflicts themselves. Also, the amount of memory to explicitly store a conflict would be $\lceil log_2(scan\ length)\rceil$ bits as it can occur anywhere on the scan chain. Below definitions for the variables used are given.

- $s \equiv$ Maximum number of specified bits in any test cube which is also the seed length.
- $Clst_i \equiv$ Number of clusters with $i$ conflicts.
- $Clst \equiv$ Total number of clusters $= \sum Clst_i$.
- $TL \equiv$ Test length.
- $Confl_M \equiv$ Maximum number of conflicts allowed in a single test cube.
- $Confl \equiv$ Total number of conflicts.
- $m \equiv$ Scan length.
- $Mem\_len \equiv$ Cache or Stack depth.
- $Loads \equiv$ Total number of cache loads or stack pushes.
- $offset \equiv$ Difference in position between a conflict and the one immediately lower to it.

### 4.2.1 Direct Implementation

This method involves explicitly representing each conflict as it occurs. The cubes are grouped into sets each with the same number of conflicts, the first with none, the second with one, and so on. The number in each set is stored and decremented as vectors are created so that the number of conflicts for each cube is implicitly known.

Associated with each stored cube is the required number of conflicts, as indicated by the set in which it is in. For clusters with 0 or 1 conflicts, all vectors suggested by the conflicts must be applied, thus no more information needs to be kept. Others require another set of vectors, *conflict vectors*, to point out which of the conflict bits need to be inverted to create the current vector. In addition to this, this set of vectors should tell when the use of the current cube has come to an end. Thus, it must be one bit longer than the maximum number of conflicts allowable.

To create the vectors which would be applied to test the circuit, first the seed is loaded into the LFSR, the appropriate feedback configuration chosen, and a base vector starts to be produced. At this time also, the conflicts associated with this cluster is loaded as is the conflict vector (if applicable). When a bit is created which corresponds to a position referred to by the conflict vector and the conflicts, it is inverted. At the end of the test vector, the conflict vector is checked to see if the same or the next seed should be loaded. If the next seed is to be loaded, its associated conflicts are also loaded. Then the next conflict vector is loaded and the process continued.

$$mem = [Clst * s] + [(TL - Clst_0 - 2 * Clst_1) * (Confl_M + 1)] + [Confl * \lceil log(m)_2 \rceil] \quad (4.6)$$

### 4.2.2 Stack and Cache Based Implementations

These implementations attempt to reduce memory requirements by making use of the fact that many conflicts are used on several occasions with different test cubes. Extra memory organized as a stack or a cache is used for this purpose. They must be at least as deep as the maximum number of conflicts allowed, but may be deeper for more flexibility. However, this would have a negative impact on the conflict vectors as their length must correspond to this depth.

In these methods, the memories are loaded and as many cubes as are consistent with these conflicts are used. Conflicts are then changed either by pushing and popping in the case of the stack, or by a least recently used cache replacement strategy. Other cache reload strategies could be used but most have the undesired effect of requiring more memory. Also, by judiciously choosing the order and value of the conflict vectors, great control on the reload is provided. In fact, through the addition of extra conflict vectors, full control of the reloading is possible. For instance, it may be necessary to keep one of the conflicts which has not been used for a long time and due to be replaced. By including this conflict in the generation of a vector where it is an unspecified bit, it can be kept without any penalty.

In addition to the extra cache/stack memory, a little additional overhead is necessary for control. Conflict vectors as described in the previous section are used, but here they indicate which cache/stack positions contain the relevant conflict. An extra bit for each seed and conflict is necessary, in the seed to tell whether new conflicts need be loaded, and in the conflict to show whether further conflicts are needed. Creating the test vectors are analogous to the direct implementation with the exception of how conflicts are loaded, if at all.

$$mem = [Clst * (s+1)] + [TL * (Mem\_len + 1)] + [Loads * (\lceil log(m) \rceil + 1)] \tag{4.7}$$

### 4.2.3 A Scheme to realize these Implementations

An implementation of these methods is given in figure 4.9 where the hardware unique to clustering is represented by the shaded portion. An XOR gate is used to invert the bits indicated by the conflict positions stored in the memory/stack/cache labeled 'relevant conflicts' and the conflict vectors. Control is provided based on extra bits stored with the conflict vectors, and conflicts and the seeds as described in the above implementations.

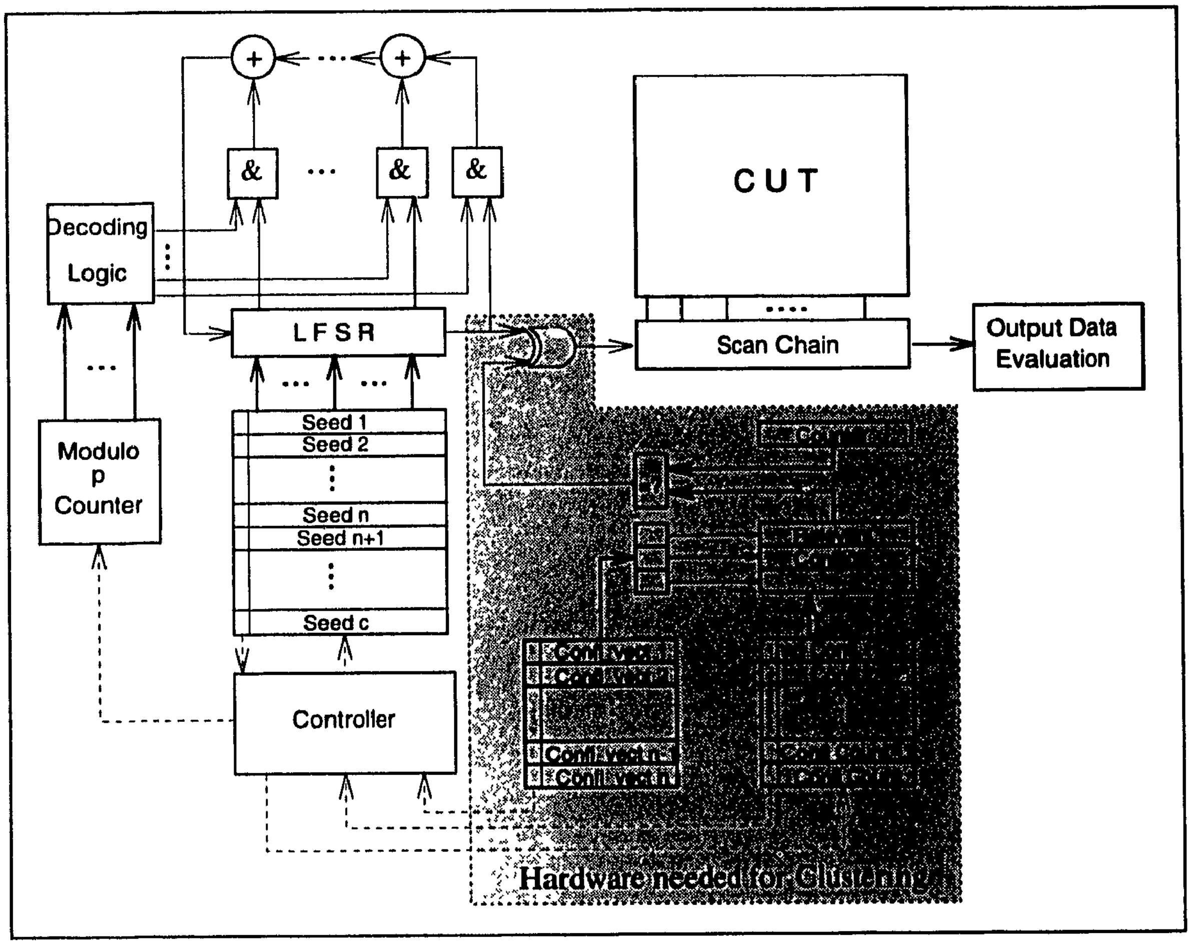


Figure 4.9: BIST Scheme Based on Clustering

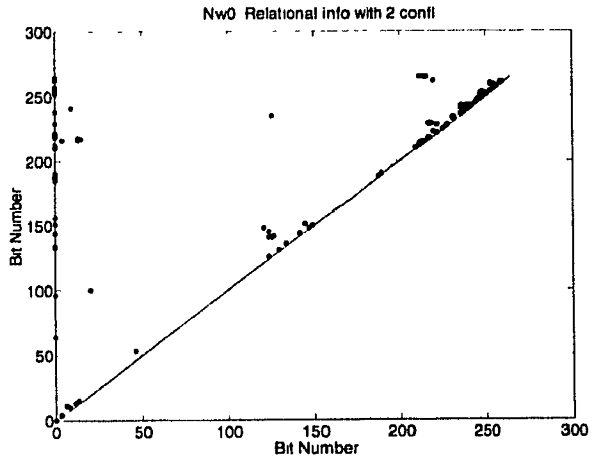
The hardware overhead of this implementation in excess of that of random pattern testing will now be determined to help evaluate its impact in BIST applications. Random pattern testing was used as a base for it has been shown to be cost effective in many BIST applications and uses some of the same structures such as output evaluation, the scan chain and most of the LFSR. The decoding logic, modulo p counter (which can be efficiently implemented as an LFSR), controller and seed memory are very similar to the hardware needed in pure reseeding. Depending on the area required by the memory, this can be implemented on chip or on an adjacent chip. The movement to clustering is to reduce the total memory needed to generate the vectors by using a slightly more elaborate compression scheme. This requires the addition of a minimal amount of additional hardware which comprises of an XOR gate, a counter (which can also be implemented as an LFSR), a few storage elements and some extra control logic.

### 4.2.4 Additional Variations

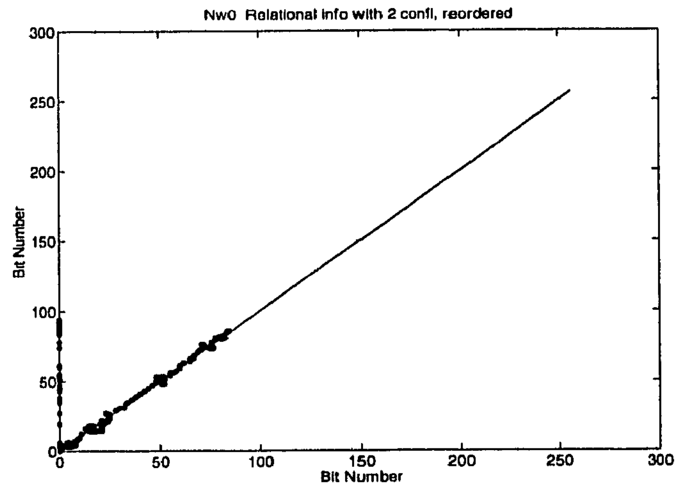
Various techniques can be applied in association with these methods in order to further reduce the memory requirements for the storage of conflicts. One such technique is ***cube implication*** (CI) which involves making the assumption that many of the cubes reachable by inversions made on the bits implied by the conflicts would have to be created. Thus rather than having to indicate them individually using conflict vectors, all would be cycled through, whether needed or not, reducing the memory needed for the [B] component in equation (4.6) to 0 and in equation (4.7) to $[Clst * (Mem\_len)]$. This results in more vectors than are actually needed being applied, but if the maximum number of conflicts is kept small, the memory savings may be worth the overhead. In the case when the conflict count is $k$, a worse case scenario would see $2^{k-1}$ times the required number of vectors being applied, $k$ is usually small.

Another interesting direction with great potential for savings can be realized provided that the inputs may be ordered at will. This technique, *reordering*, orders the inputs such that inputs representing conflicts in the same cluster occur as close together as possible and as close to the beginning of the vector as possible. This has the effect of improving on the locality of conflicts.

Figure 4.10 shows the effect of reordering, based on a maximum of two conflicts, on the circuit Nw0 which has 265 inputs. Each point on the graph indicates that there is at least one cluster for which its conflicts occur on the bits denoted by its co-ordinates. The line is the reference, $y = x$. In figure 4.10(A), even though many of the points are close to the reference line, there are many which are significantly distant, such as point



(A)



(B)

Figure 4.10: Conflict Profile before and after Reordering

(9, 241). Also they may exist anywhere above the line (as the conflicts are ordered). Figure 4.10(B) shows that all the conflicts occur on the first ninety or so inputs and that the difference between conflicts on the same cluster is small, less than five in this case in spite of the fact that the program was written to try and minimize the average distance between conflicts, not the maximum distance. Using this information, a very compact realization can be formulated. For instance, a counter representing the base conflict can be used, and the offset of the next conflict (three bits) can be submitted if different from the previous one. As the counter is incremented (in this case, say, from 0 to 90), all test cubes with this as the lower conflict can be tested. When both cube implication and reordering are assumed, memory use is given in equation (4.8). Furthermore, a small stack/cache type memory may be used so as to reuse offset information.

$$mem \;=\; \lceil Clst * s\rceil + \lceil \;\; \rceil + \lceil Clst + (\sum_{i\geq 2}((i-1) * Clst_i)) * \lceil log_2\lceil offset\rceil\rceil\rceil \tag{4.8}$$

# Chapter 5

# Experimental Results

Experiments based on the clustering algorithm were carried out on both a subset of the ISCAS 89 benchmark circuits and a set of industrial circuits provided by IBM. Using these results, memory requirements for many of the various implementations were calculated and compared to that needed for reseeding.

Test sets were found differently depending on which set the circuit was from. In the case of the ISCAS circuits, a predetermined number of random patterns were applied to the circuit in an attempt to remove the relatively easy to detect faults. After this, the remaining faults were targeted and patterns found to detect each. *Bit stripping* [Mur90] was then used to reduce the number of specified bits. For the industrial circuits, test sets were found through ATPG and all vectors with fewer than 20 specified bits were deemed as 'easy to detect' and removed.

Tables [5.1, 5.2, 5.3, 5.4, 5.5] apply to the IBM set while tables [5.6, 5.7, 5.8] refer to the ISCAS circuits

Tables [5.1, 5.2] compare the effectiveness of the two clustering methods with different partition sizes, conflict counts and ranges. For both of these, the conflict count was kept at 3. In order to help in the grouping of clusters for use with the cache method, where the cache depth is greater than the conflict count, the concept of a *range* was introduced. The range represents the cache, so a group of clusters can be in the same range provided that the union of their conflicts has a cardinality less than the cache depth. From these results, it is evident that the multiple cluster algorithm consistently outperforms the single cluster one in all cases unless the range is much larger than the conflict count. However, in the cases where the number of test cubes is lower in the single cluster algorithm, the total number of conflicts present is significantly larger. Thus it was deemed that the multiple cluster algorithm was distinctly superior so the rest of the experiments were carried out using only this algorithm.

Also evident in these tables is the fact that choosing a good partitioning for the cubes can have an effect in the resulting numbers of clusters and conflicts. It is generally noted that for the multiple cluster algorithm, a single partition is not necessarily the best choice.

Tables 5.3 and 5.6 give a brief impression of the initial test sets. To work out an estimate for the memory required using conventional methods, once again the multiple cluster algorithm was used but here the conflict count was put to 0 while the maximum number of specified bits allowed was set to the length of the vector. This has the effect of packing as many of the cubes together as possible minimizing the consequent vectors, and hence the memory needed to store them implicitly.

Tables 5.4 and 5.7 show the effect of clustering on these sets. When no conflicts were allowed and the maximum number of specified bits allowed was set to the number of specified bits in the most specified cube in the set, the algorithm reduces to that of reseeding and these values were used in estimating the memory requirements for this method. In general, clustering does not result in a test length much longer than reseeding but the number of clusters resulting is significantly reduced.

Tables 5.5 and 5.8 give the memory requirements for some of the competing configurations. The [memory bits, storage] for reseeding is compared to the conventional results and presented in brackets as a percentage. This is also done with the [memory bits, storage] in the [stack], [cache] and [reorder and CI] results where the memory required is compared to that of both reseeding and conventional results.

For better results with the direct, cache and stack methods, the cube implicit (CI) assumption can be made, but is not given here. Also, in the cache and stack estimates, it is assumed that the cache/stack depth is equal to the maximum conflict count allowed, and that the reloading of the cache can be fully controlled using the conflict vectors. However, this assumption would not hold if the cube implicit technique is adopted.

To work out the 'Max. Offset' when the number of conflicts was greater than 2, the conflicts were split up into all possible combinations taken two at a time and submitted to the reordering program. So, if the conflicts for a cluster were (2 56 109), this would be split into the triplet (2 56), (2 109) and (56 109) for reordering. Thus reordering gives the maximum distance from the lowest position conflict to the highest. For instance, if the number of conflicts in a cluster was 3, the minimum the maximum distance could be is 2 when the three conflicts are in adjacent positions. In an attempt to compensate for the fact that the average rather than the maximum distance is minimized in this algorithm, 200 iterations were done while increasing the weight of the link with the maximum distance, and the minimum of these taken.

The memory demanded by the direct method was less than or at worse equal to that demanded using pure reseeding, equality occurring when the results were identical. In some cases with the ISCAS circuits, the penalty incurred using the stack/cache methods was greater than its benefits but generally they proved superior, especially in the case of the industrial circuits with up to about a 20% savings in the case of nw4. The use of a greater conflict count had mixed savings, sometimes 2 conflicts and sometimes 3 were better. For the ISCAS circuits, the savings in memory, without allowing for reordering and cube implication (CI), was typically less than that of the IBM circuits, ranging from 0% (in the case where no conflicts were found) to 42% compared to a low of 48% up to 66% in the case of the industrial circuits. When reordering and cube implication is considered, even less memory is required to a maximum of 76% savings.

| IBM Circuit | Range | Extra Spec. | Single Cluster Algo. | | Multiple Cluster Algo. | |
|---|---|---|---|---|---|---|
| | | | No. of Clusters | No. of Conflicts | No. of Clusters | No. of Conflicts |
| nw1 | 3 | 0 | 163 | 432 | 157 | 349 |
| | 3 | 1 | 169 | 436 | 157 | 350 |
| | 3 | 2 | 171 | 434 | 158 | 347 |
| | 3 | 3 | 171 | 447 | 158 | 346 |
| | 3 | 4 | 173 | 454 | 158 | 347 |
| | 3 | 5 | 172 | 451 | 157 | 347 |
| | 3 | 6 | 173 | 449 | 157 | 347 |
| | 3 | 7 | 174 | 443 | 157 | 347 |
| | 3 | 8 | 174 | 443 | 159 | 346 |
| | 3 | $\infty$ | 174 | 442 | 159 | 346 |
| nw3 | 3 | 0 | 68 | 175 | 52 | 132 |
| | 3 | 1 | 73 | 177 | 52 | 125 |
| | 3 | 2 | 69 | 169 | 52 | 123 |
| | 3 | 3 | 71 | 180 | 51 | 124 |
| | 3 | 4 | 71 | 174 | 50 | 124 |
| | 3 | 5 | 70 | 172 | 49 | 122 |
| | 3 | 6 | 73 | 179 | 49 | 122 |
| | 3 | 7 | 69 | 170 | 49 | 122 |
| | 3 | 8 | 69 | 170 | 49 | 122 |
| | 3 | $\infty$ | 73 | 178 | 49 | 122 |
| nw8 | 3 | 0 | 118 | 327 | 89 | 231 |
| | 3 | 8 | 118 | 327 | 89 | 231 |
| | 3 | $\infty$ | 103 | 283 | 113 | 195 |

Table 5.1: Statistics comparing the performance of the two algorithms and showing the effect of the Ex-spec parameter

| IBM Circuit | Range | Single Cluster Algo. | | Multiple Cluster Algo. | |
|---|---|---|---|---|---|
| | | No. of Testcubes | No. of Conflicts | No. of Testcubes | No. of Conflicts |
| nw0 | 8 | 192 | 909 | 196 | 522 |
| nw1 | 8 | 113 | 498 | 111 | 326 |
| nw2 | 8 | 37 | 182 | 48 | 133 |
| nw3 | 8 | 39 | 201 | 42 | 119 |
| nw4 | 8 | 147 | 800 | 154 | 528 |
| nw6 | 8 | 96 | 430 | 89 | 439 |
| nw7 | 8 | 88 | 397 | 83 | 396 |
| nw8 | 8 | 58 | 323 | 54 | 228 |

Table 5.2: Statistics comparing the performance of the two algorithms for ranges $\gg$ conflict count

| Name | Scan Length | No. of Vectors | Av. No of Spec. Bits | Conventional Results | |
|---|---|---|---|---|---|
| | | | | Test Length | Mem. Bits for Storage |
| nw0 | 265 | 1465 | 45.26 | 620 | 164k |
| nw1 | 401 | 513 | 89.08 | 391 | 157k |
| nw2 | 407 | 210 | 67.62 | 146 | 59.4k |
| nw3 | 80 | 197 | 27.92 | 174 | 13.9k |
| nw4 | 163 | 1295 | 32.97 | 924 | 151k |
| nw6 | 443 | 1121 | 32.83 | 202 | 89.5k |
| nw7 | 486 | 971 | 27.60 | 140 | 68.0k |
| nw8 | 317 | 372 | 40.32 | 111 | 38.5k |

Table 5.3: Statistics on the Original Test Sets of the Industrial Circuits

| Name | Seed Length | Max. Confl. | No. of Clusters | No. of Clusters with | | | | Total No. of Conflicts | Test Length | Av. No. of Spec. Bits |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 0 Confl. | 1 Confl. | 2 Confl. | 3 Confl. | | | |
| nw0 | 87 | 0 | 627 | 627 | 0 | 0 | 0 | 0 | 627 | 48.61 |
| | | 2 | 314 | 44 | 70 | 200 | 0 | 470 | 632 | 50.96 |
| | | 3 | 245 | 23 | 18 | 100 | 104 | 530 | 631 | 51.84 |
| nw1 | 148 | 0 | 505 | 505 | 0 | 0 | 0 | 0 | 505 | 89.50 |
| | | 2 | 199 | 8 | 71 | 120 | 0 | 311 | 506 | 89.95 |
| | | 3 | 157 | 1 | 27 | 64 | 65 | 350 | 506 | 87.26 |
| nw2 | 84 | 0 | 210 | 210 | 0 | 0 | 0 | 0 | 210 | 67.62 |
| | | 2 | 78 | 0 | 24 | 54 | 0 | 132 | 210 | 67.44 |
| | | 3 | 58 | 0 | 4 | 32 | 22 | 131 | 210 | 64.83 |
| nw3 | 45 | 0 | 171 | 174 | 0 | 0 | 0 | 0 | 174 | 28.10 |
| | | 2 | 71 | 6 | 20 | 45 | 0 | 110 | 174 | 27.61 |
| | | 3 | 49 | 0 | 4 | 17 | 28 | 122 | 174 | 27.35 |
| nw4 | 45 | 0 | 973 | 973 | 0 | 0 | 0 | 0 | 973 | 34.12 |
| | | 2 | 326 | 0 | 47 | 279 | 0 | 605 | 1085 | 32.82 |
| | | 3 | 204 | 0 | 17 | 34 | 153 | 544 | 1075 | 32.55 |
| nw6 | 79 | 0 | 319 | 319 | 0 | 0 | 0 | 0 | 319 | 64.89 |
| | | 2 | 188 | 86 | 30 | 72 | 0 | 174 | 325 | 64.36 |
| | | 3 | 155 | 69 | 3 | 25 | 58 | 227 | 321 | 63.74 |
| nw7 | 50 | 0 | 318 | 318 | 0 | 0 | 0 | 0 | 318 | 42.14 |
| | | 2 | 183 | 80 | 30 | 73 | 0 | 176 | 326 | 44.48 |
| | | 3 | 158 | 68 | 16 | 28 | 46 | 210 | 312 | 45.06 |
| nw8 | 62 | 0 | 275 | 275 | 0 | 0 | 0 | 0 | 275 | 53.82 |
| | | 2 | 123 | 5 | 16 | 102 | 0 | 220 | 343 | 43.66 |
| | | 3 | 89 | 0 | 1 | 34 | 54 | 231 | 350 | 41.91 |

Table 5.4: Clustering Statistics on the Industrial Circuits

| | Reseeding | | Direct | Stack | | Cache | | Reorder + CI | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Mem. Bit, Storage | Max. Confl. | Mem. Bit, Storage | No. of Pushes | Mem. Bit, Storage | No. of Loads | Mem. Bit, Storage | Max Offset | Mem. Bit, Storage |
| nw0 | 54.5k (33) | 2 | 32.9k (60) (20) | 148 | 31.0k (57) (19) | 130 | 30.8k (57) (19) | 5 | 28.2k (52) (17) |
| | | 3 | 28.4k (52) (17) | 191 | 26 0k (48) (16) | 146 | 25.6k (47) (16) | 40 | 23.4k(43) (14) |
| nw1 | 74.7k (48) | 2 | 33.3k (45) (21) | 175 | 32.9k (44) (21) | 154 | 32.7k (44) (21) | 16 | 30.1k (40) (19) |
| | | 3 | 28.1k (38) (18) | 191 | 27.3k (37) (17) | 164 | 27.1k (36) (17) | 25 | 24.4k (33) (16) |
| nw2 | 17.6k (30) | 2 | 8.23k (47) (14) | 102 | 8.28k (47) (14) | 96 | 8.22k (47) (14) | 1 | 6.68k (38) (11) |
| | | 3 | 6.89k (39) (12) | 112 | 6.89k (39) (12) | 95 | 6.72k (38) (11) | 4 | 5.08k (29) (9) |
| nw3 | 7.83k (56) | 2 | 4.35k (56) (31) | 55 | 4.23k (54) (30) | 42 | 4.12k (53) (30) | 7 | 3.40k (43) (24) |
| | | 3 | 3.72k (48) (27) | 75 | 3.55k (45) (26) | 55 | 3.39k (43) (24) | 22 | 2.62k (33) (19) |
| nw4 | 43.8k (29) | 2 | 22.5k (51) (15) | 127 | 19.4k (44) (13) | 102 | 19.2k (44) (13) | 14 | 16.1k (37) (11) |
| | | 3 | 17.7k (40) (12) | 153 | 15.1k (34) (10) | 108 | 14.7k (34) (10) | 15 | 10.7k (24) (7) |
| nw6 | 25.2k (28) | 2 | 17.0k (68) (19) | 124 | 17.3k (69) (19) | 113 | 17.1k (68) (19) | 3 | 15.2k (60) (17) |
| | | 3 | 15.3k (61) (17) | 178 | 15.5k (62) (17) | 153 | 15.2k (60) (17) | 19 | 13.1k (52) (15) |
| nw7 | 15.9k (23) | 2 | 11.3k (71) (17) | 139 | 11.7k (74) (17) | 130 | 11.6k (73) (17) | 5 | 9.55k (60) (14) |
| | | 3 | 10.6k (67) (16) | 169 | 11.0k (69) (16) | 150 | 10.8k (68) (16) | 6 | 8.42k (53 ) (12) |
| nw8 | 17.1k (44) | 2 | 10.5k (61) (27) | 186 | 10.6k (62) (28) | 173 | 10.5k (61) (27) | 1 | 7.85k (46) (20) |
| | | 3 | 8 99k (53) (23) | 199 | 9.00k (53) (23) | 181 | 8.82k (52) (23) | 3 | 5.89k (34) (15) |

Table 5.5: Memory Requirements for the Different Configurations

| Name | Scan Length | Ran. Pat. | No. of Vectors | Av. No. of Spec. Bits | Conventional Results Test Length | Conventional Results Mem. Bits for Storage |
|---|---|---|---|---|---|---|
| s298 | 17 | 32 | 25 | 9.44 | 19 | 323 |
| s1423 | 91 | 1k | 36 | 13.78 | 8 | 728 |
| s1488 | 14 | 1k | 51 | 10.06 | 20 | 280 |
| s1494 | 14 | 1k | 53 | 9.92 | 22 | 308 |
| s5378 | 214 | 1k | 135 | 39.78 | 85 | 18.2k |
| s9234 | 247 | 1k | 313 | 34.82 | 132 | 32.6k |
| s13207 | 700 | 1k | 378 | 45.77 | 195 | 137k |
| s35932 | 1763 | 32 | 49 | 121.22 | 49 | 86.4k |
| s38417 | 1664 | 80k | 359 | 49.86 | 62 | 103k |

Table 5.6: Statistics on the Original Test Sets of the ISCAS 89 Circuits

| Name | Ran. Pat | Seed Len. | Max. Confl. | No. of Clusters | No. of Clusters with 0 Confl. | 1 Confl. | 2 Confl. | 3 Confl. | Total No. of Confl. | Test Len. | Av. No. of Spec. Bits |
|---|---|---|---|---|---|---|---|---|---|---|---|
| s298 | 32 | 17 | 0 | 19 | 19 | 0 | 0 | 0 | 0 | 19 | 11.11 |
| | | | 2 | 10 | 2 | 5 | 3 | 0 | 11 | 19 | 13.90 |
| | | | 3 | 9 | 4 | 1 | 1 | 3 | 12 | 19 | 14.56 |
| s1423 | 1k | 21 | 0 | 12 | 12 | 0 | 0 | 0 | 0 | 12 | 15.75 |
| | | | 2 | 8 | 4 | 1 | 3 | 0 | 7 | 12 | 16.88 |
| | | | 3 | 6 | 1 | 1 | 0 | 4 | 13 | 13 | 15.67 |
| s1488 | 1k | 11 | 0 | 22 | 22 | 0 | 0 | 0 | 0 | 22 | 10.18 |
| | | | 2 | 10 | 3 | 1 | 6 | 0 | 13 | 21 | 9.80 |
| | | | 3 | 7 | 0 | 1 | 1 | 5 | 18 | 22 | 9.86 |
| s1494 | 1k | 11 | 0 | 22 | 22 | 0 | 0 | 0 | 0 | 22 | 10.05 |
| | | | 2 | 10 | 1 | 4 | 5 | 0 | 14 | 22 | 9.40 |
| | | | 3 | 7 | 0 | 2 | 1 | 4 | 16 | 22 | 9.57 |
| s5378 | 1k | 128 | 0 | 85 | 85 | 0 | 0 | 0 | 0 | 85 | 61.88 |
| | | | 2 | 62 | 25 | 5 | 32 | 0 | 69 | 100 | 77.10 |
| | | | 3 | 66 | 44 | 0 | 4 | 18 | 62 | 88 | 74.24 |
| s9234 | 1k | 126 | 0 | 141 | 141 | 0 | 0 | 0 | 0 | 141 | 66.88 |
| | | | 2 | 97 | 47 | 10 | 40 | 0 | 90 | 149 | 87.84 |
| | | | 3 | 88 | 33 | 9 | 10 | 36 | 137 | 145 | 93.80 |
| s13207 | 1k | 263 | 0 | 195 | 195 | 0 | 0 | 0 | 0 | 195 | 85.41 |
| | | | 2 | 117 | 50 | 16 | 51 | 0 | 118 | 198 | 131.02 |
| | | | 3 | 106 | 36 | 16 | 11 | 43 | 167 | 197 | 140.58 |
| s35932 | 32 | 253 | 0 | 49 | 49 | 0 | 0 | 0 | 0 | 49 | 121.18 |
| | | | 2 | 49 | 49 | 0 | 0 | 0 | 0 | 49 | 121.18 |
| | | | 3 | 49 | 49 | 0 | 0 | 0 | 0 | 49 | 121.18 |
| s38417 | 80k | 224 | 0 | 98 | 98 | 0 | 0 | 0 | 0 | 98 | 171.34 |
| | | | 2 | 85 | 59 | 8 | 18 | 0 | 44 | 111 | 191.96 |
| | | | 3 | 82 | 49 | 4 | 17 | 12 | 74 | 115 | 197.00 |

Table 5.7: Clustering Statistics on the ISCAS 89 Circuits

| Name | Ran. Pat. | Reseeding Mem. Bit, Storage | Max. Confl. | Direct Mem. Bit, Storage | Stack No. of Pushes | Stack Mem. Bit, Storage | Cache No. of Loads | Cache Mem. Bit, Storage | Reorder + CI Max. Offset | Reorder + CI Mem. Bit, Storage |
|---|---|---|---|---|---|---|---|---|---|---|
| s298 | 32 | 323 (100) | 2 | 246 (76) (76) | 7 | 279 (86) (86) | 7 | 279 (86) (86) | 1 | 183 (57) (57) |
| | | | 3 | 265 (82) (82) | 8 | 286 (89) (89) | 7 | 280 (87) (87) | 3 | 176 (54) (54) |
| s1423 | 1k | 252 (35) | 2 | 235 (93) (32) | 7 | 268 (106) (37) | 7 | 268 (106) (37) | 1 | 179 (71) (25) |
| | | | 3 | 257 (102) (35) | 13 | 288 (114) (40) | 13 | 288 (114) (40) | 2 | 140 (56) (19) |
| s1488 | 1k | 242 (86) | 2 | 210 (87) (75) | 9 | 228 (94) (81) | 9 | 228 (94) (81) | 2 | 126 (52) (45) |
| | | | [illegible] | 22[illegible] (95) (82) | 12 | 232 (96) (83) | 10 | 222 (92) (79) | 5 | 117 (48) (42) |
| s1494 | 1k | 242 (79) | [illegible] | 20[illegible] (85) (67) | 9 | 231 (95) (75) | 9 | 231 (95) (75) | 2 | 125 (52) (41) |
| | | | [illegible] | 21[illegible] (88) (69) | 11 | 227 (94) (74) | 10 | 222 (92) (72) | 3 | 102 (42) (33) |
| s5378 | 1k | 10.9k (60) | 2 | 8.68k (80) (48) | 45 | 8.70k (80) (48) | 37 | 8.63k (79) (47) | 6 | 8.09k (74) (44) |
| | | | 3 | 9.1[illegible] (84) (50) | 42 | 9.24k (85) (51) | 37 | 9.20k (84) (51) | 10 | 8.67k (80) (48) |
| s9234 | 1k | 17.8k (55) | 2 | 13.2k (74) (40) | 60 | 13.3k (75) (41) | 55 | 13.3k (75) (41) | 7 | 12.4k (70) (38) |
| | | | 3 | 12.6k (71) (39) | 94 | 12.6k (71) (39) | 80 | 12.5k (70) (38) | 22 | 11.6k (65) (36) |
| s13207 | 1k | 51.3k (37) | 2 | 32.3k (63) (24) | 37 | 31.9k (62) (23) | 32 | 31.8k (62) (23) | 7 | 31.0k (60) (23) |
| | | | 3 | 30.1k (61) (22) | 67 | 29.5k (58) (22) | 52 | 29.3k (57) (21) | 11 | 28.4k (55) (21) |
| s35932 | 32 | 12.4k (11) | 2 | 12.1k (100) (11) | 0 | 12.5k (101) (14) | 0 | 12.5k (101) (14) | 0 | 12.4k (100) (14) |
| | | | 3 | 12.4k (100) (14) | 0 | 12.5k (101) (14) | 0 | 12.5k (101) (14) | 0 | 12.4k (100) (14) |
| s38417 | 80k | 22.0k (21) | 2 | 19.7k (90) (19) | 40 | 20.0k (91) (19) | 40 | 20.0k (91) (19) | 2 | 19.1k (87) (19) |
| | | | 3 | 19.4k (88) (19) | 67 | 19.7k (90) (19) | 65 | 19.7k (90) (19) | 3 | 18.5k (84) (18) |

Table 5.8: Memory Requirements for the Different Configurations

# Chapter 6

# Conclusion

A new method of using reseeding of LFSRs to generate a deterministic test set capable of 100% non-redundant fault detection of circuits has been presented. By allowing a small number of conflicts or inconsistencies when merging cubes, the test set can be reduced to a much greater degree than that of pure reseeding. However, to successfully recreate the required vectors, further information reguarding the positions and the necessary values of the conflicts also need to be stored.

Algorithms were developed to merge the cubes in a test set and several ways, using little additional hardware, were proposed to minimize the memory penalty resulting from the conflicts. It is possible to trade-off test time against test data through, both by varying the length of the random pattern test stage or by using cube implication. Furthermore, hardware complexity can also be traded-off through the use of the different methods suggested. Results based on a subset of the ISCAS benchmark circuits and on a set of industrial circuits show that a 35% reduction on average over pure reseeding can be achieved without significantly increasing the test length. Further reductions are possible, up to an average of 48%, with an increase in test length provided reordering on the inputs is po ible. When compared to conventional deterministic requirements, these reductions are 69% and 77% respectively. The industrial circuits were chosen based on their inherent resistance to random testing yet have performed notably better than the ISCAS counterparts. This is encouraging for it appears that some of the factors which make circuits hard to test are the very ones which benefit this approach.

Though envisioned for BIST applications and though it uses memory comparable to that of weighted random pattern testing without much of its hardware overhead, it is conceivable that the memory requirement may be excessive. In such cases this method, like reseeding, can be applied to external testing effectively reducing data bandwidth by storing and downloading only the seeds and the conflict information. Only the on chip decoding logic would be required to generate the vectors.

In addition to chip level testing, this scheme can also be used at board level and with multiple chip modules. Extensions to this method can be useful with delay testing for this type of testing requires the application of two consecutive vectors which are identical on all except for a few bits. By having two classes of 'conflicts', one for the conflicts as defined in this thesis while the other for the differences between the two consecutive vectors, delay faults can be tested with significantly reduced memory overhead.

# Bibliography

[Abr90] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[Agr89] V. D. Agrawal, K. Chen, D. Johnson, and T. Lin. "A Directed Search Method for Test Generation using a Concurrent Simulator". *IEEE Design and Test of Computers*, vol. 5:8–15, April 1988.

[Agr87] V. D. Agrawal, K. Cheng, D. Johnson, and T. Lin. "A Complete Solution to the Partial Scan Problem". *International Test Conference*, pages 44 50, 1987.

[Agr93] V. D. Agrawal, C. Kime, and K. Saluja. "A Tutorial on Built-In Self-Test, Part I: Principles". *IEEE Design and Test of Computers*, pages 73 82, Mar. 1993.

[Bar87] P. Bardell, W. McAnney, and J. Savir. *Built-In Test for VLSI.* Wiley-Interscience, New York, 1987.

[Info90] B. Bennetts, B. Courtois, C. Maunder, J. Mucha, F. Pool, G. Robbinson, B. Schneider, and T. Williams. "The Challenges of Self-Test". *IEEE Design and Test of Computers*. pages 46 -54, 1990.

[Brg89] F. Brglez, C. Gloster, and G. Kedem. "Hardware-Based Weighted Random Pattern Generation for Boundary Scan". *IEEE International Test Conference*, pages 264–273, 1989.

[Cha85] R. Chandramouli and H. Sucar. "Defect Analysis and Fault Modelling in MOS Technology". *International Test Conference*, pages 313–321, 1985.

[Dav82] B. Davis. *The Economics of Automatic Testing.* McGraw-Hill, New York, 1982.

[Dea91] I. Dear, C. Dislis, A. Ambler, and J. Dick. "Economic Effects on Design and Test". *IEEE Design and Test of Computers*, pages 64–77, 1991.

[Glo89] C. Gloster and F. Brglez. "Boundary Scan with Built-In Self-Test". *IEEE Design and Test of Computers*, pages 36–44, 1989.

[Gol82] S. Golomb. *Shift Register Sequences.* Aegean Park Press, Larguna Hills, Calif, 1982.

[Has92] A. Hassan, V. K. Agarwal, B. Nadeau-Doste, and J. Rajski. "BIST of PCB Interconects Using Boundary-Scan Architecture". *IEEE Transactions on CAD*, Vol. 11(No. 10):1278–1287, Oct. 1992.

[Hel92] S. Hellebrand, S. Tarnick, J. Rajski, and B. Courtois. "Generation of Vector Patterns Through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers". *IEEE International Test Conference*, pages 120–129, 1992.

[Hor90] P. Hortensius, R. McLeod, and B. Podaima. "Cellular Automata Circuits for Built-In Self-Test". *IBM Journal on Research and Development*, Vol. 34(No. 2/3):389–405, Mar/May 1990.

[Hor89] P. Hortensius, R. McLeod, E. Pries, D. Miller, and H. Card. "Cellular Automata-Based Pseudorandom Number Generators for Built-In Self-Test". *IEEE Transactions on CAD*, Vol. 8(No. 8):842–858, Aug. 1989.

[Hug86] J. Hughes and E. McCluskey. "Multiple Stuck-at Fault Coverage of Single Stuck-at Fault Test Sets". *International Test Conference*, pages 368–374, 1986.

[Iba75] H. Ibarra and S. Sahni. "Polynomially Complete Fault Detection Problems". *IEEE Trans. Computing*, Vol. C-24:242–249, Mar. 1975.

[Jac87] J. Jacob and N. Biswas. "GTBD Faults and Lower Bounds on Multiple Fault Coverage of Single Fault Test Sets". *International Test Conference*, pages 849–855, 1987.

[Jha86] N. Jha. "Detecting Multiple Faults in CMOS Circuits". *International Test Conference*, pages 514–519, 1986.

[Kon91] B. Könemann. "LFSR-Coded Test Patterns for Scan Designs". *Proc. European Test Conference*, pages 237–242, 1991. Munich.

[Lin87] C. Lin and S. Reddy. "On Delay Fault Testing in Logic Circuits". *IEEE Tranactions on CAD*, pages 649–703, Sep 1987.

[Lis87] R. Lisanke, F. Brglez, A. Degeus, and D. Gregory. "Testability-Driven Random Test-Pattern Generation". *IEEE Transactions on CAD*, Vol. CAD-6:1082–1087, Nov. 1987.

[Ma88] H. Ma, S. Devadas, A. Newton, and A. Sangiovanni-Vincentelli. "Test Generation for Sequential Circuits". *IEEE Tranctions on CAD*, pages 212–220, Feb. 1990.

[Mcc85] E. McCluskey. "Built-in-Self-Test Structures". *IEEE Design and Test*, pages 29–36, Apr. 1985.

[McC87] E. McCluskey, S. Makar, S. Mourand, and K. Wagner. "Probablity Models for Pseudorandom Test Sequences". *IEEE International Test Conference*, pages 471–479, 1987.

[Mur90] F. Muradali, V. Agarwal, and B. Nadeau-Dostie. "A New Procedure for Weighted Random Built-In Self-Test". *IEEE International Test Conference*, pages 660–669, 1990.

[Pat91b] S. Pateras and J. Rajski. "Cube-Contained Random Patterns and their Application to the Complete Testing of Synthesized Multi-level Circuits". *IEEE International Test Conference*, 1991.

[Pat91a] S. Pateras and J. Rajski. "Generation of Correlated Random Patterns for the Complete Testing of Synthesized Multi-level Circuits". *28th ACM/IEEE Design Automation Conrerence*, pages 347–352, 1991.

[Pen92] S. Peng. "Test Methodology for a large ASIC Design". *Wescon Conference Record*, V36:64–77, 1992.

[Poa62] J. Poage. "Derivation of Optimum Tests to Detect in Combinational Circuits". *Proc Symp on Mathmetical Theory of Automata*, pages 483–528, 1962.

[Pom91] I. Pomeranz, L. Reddy, and S. Reddy. "On Achieving a Complete Fault Coverage for Sequential Machines using the Transition Fault Model". *International Test Conference*, Oct. 1991.

[Raj87] J. Rajski and H. Cox. "A Method of Test Generation and Fault Diagnosis in Very Large Combinational Circuits". *International Test Conference*, pages 932–943, 1987.

[Sav84] J. Savir and P. Bardel. "On Random Pattern Test Length". *IEEE Transactions on Computers*, Vol. C-33(No. 6):467–474, June 1984.

[Sch72] D. Schertz and G. Metze. "A New Representation for Faults in Combinational Digital Circuits". *IEEE Transactions on computing*, C-21:858–866, Aug 1972.

[Sch73] H. Schnurmann. "A Computer Program for Weighted Test Pattern Generation in Monte Carlo Testing of Integrated Circuits". *IBM Tech. Disclosure Bull*, Vol. 16:417–423, July 1973.

[Sch75] H. Schnurmann, E. Lindbloom, and R. Carpenter. "The Weighted Random Test-Pattern Generator". *IEEE Transactions on Computers*, Vol. c-24(No. 7):695-700, July 1975.

[Sch88] M. Schulz, E. Trischler, and T. Sarfert. "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System". *IEEE Transactions on CAD*, Vol. 7(No. 1):126–136, Jan. 1988.

[She85] J. Shen, W. Maly, and F. Ferguson. "Inductive Fault Analysis of MOS Integrated Circuits". *IEEE Design and Test of Computers*, pages 13–26, Dec 1985.

[Smi85] G. Smith. "Model for Delay Faults based upon Paths". *International Test Conference*, pages 342–349, 1985.

[Tri80] E. Trischler. "Incomplete Scan Path with an Automatic Test Generation Methodology". *International Test Conference*, pages 153–162, Nov. 1980.

[Tur90] J. Turino. *Design to test.* Van Nostrand Reinhold, New York, 2 edition, 1990.

[Ven93] S. Venkataraman, J. Rajski, S. Tarnick, and S. Hellebrand. "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers". *International Conference on Computer-Aided Design*, pages 572–577, 1993.

[Wad78] R. Wadsack. "Fault Modelling and Logic Simulators of CMOS and MOS integrated circuits". *The Bell System Technical Journal*, pages 1449–1473, 1978.

[Wai88] J. Waicukauski and E. Lindbloom. "Fault Detection Effectiveness of Weighted Random Patterns". *International Test Conference*, pages 245–255, 1988.

[Wil73] M. Williams and J. Angel. "Enhancing Testability of Large Scale Integrated Circuits via Test Points and Additional Logic". *IEEE Transactions on Computers*, C-22(No. 1):46–60, Jan. 1973.

[Wil87] T. Williams and W. Daehn. "Aliasing Errors in Signature Analysis Registers". *IEEE Design and Test*, pages 39–45, Apr. 1987.

[Wun87] H. Wunderlich. "Self Test Using Unequiprobable Random Patterns". *International Symposium on Fault-Tolerant Computing, FTCS-17*, pages 258–263, 1987.

[Wun88] H. Wunderlich. "Multiple Distributions for Biased Random Test Patterns". *IEEE International Test Conference*, pages 236–244, 1988.

[Zha92] S. Zhang, R. Byrne, and D. Miller. "BIST Generators for Sequential Faults". *IEEE International Conference on Computer Design*, 1992.

[Zor92] Y. Zorian. "A Universal Testability Strategy for Multi-Chip Modules Based on BIST and Boundary Scan". *International Conference on Computer Design: VLSI in Computers and Processors*, pages 59–66, 1992.