Scalable Deterministic Logic Built-In Self-Test

Von der Fakultät Informatik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) vorgelegte Abhandlung

Vorgelegt von
Valentin Gherman

Hauptberichter: **Prof. Dr. rer. nat. H.-J. Wunderlich**
Mitberichter: **Prof. Dr. rer. nat. W. Anheier**

Tag der mündlichen Prüfung: 19. Mai 2006

Institut für Technische Informatik der Universität Stuttgart

2005

Abstract

The core-based design style of integrated circuits (ICs) helps to manage the development challenges brought by the ever increasing complexity of integrated systems and the ever tighter time-to-market. Nevertheless, test-related problems are still far away from having a unitary and satisfactory solution, especially in the system on a chip (SOC) context.

For the test of ICs two reference approaches are available: external testing and built-in self-test (BIST), out of which a variety of hybrid test strategies are obtained by test resource partitioning (TRP). The final goal is to provide advantageous tradeoffs of the test evaluation indicators like: test development and application cost, hardware overhead, fault coverage, etc.

BIST offers support for in-field, on-line, burn-in and at-speed test that is indispensable for delay fault testing. Moreover, tradeoffs between fault coverage, hardware overhead and test length are possible. External testing is characterized by flexibility, reduced hardware overhead and high fault coverage for a given test length.

Deterministic logic BIST (DLBIST) is an attractive test strategy, since it combines the advantages of deterministic external testing and pseudo-random logic BIST (LBIST). Unfortunately, previously proposed DLBIST methods are unsuited for large ICs, since computation time and memory consumption of the DLBIST synthesis algorithms increase exponentially, or at least cubically, with the circuit size.

In this work, a novel procedure for the development of the so-called *bit-flipping* DLBIST scheme is proposed, which has nearly linear complexity in terms of both computation time and memory consumption. This new method is based on the use of Binary Decision Diagrams (BDDs). The efficiency of the employed algorithms is demonstrated for industrial designs containing up to 2M gates.

The embedded test sequences obtained by mapping deterministic cubes to pseudo-random sequences are also evaluated with respect to the coverage of non-target defects, which are modeled with the help of resistive bridging faults. The experimental results prove that both deterministic cubes and pseudo-random sequences are useful for detecting non-target defects. Moreover, possible tradeoffs between test length, hardware overhead, fault coverage and non-target defect coverage are analyzed.

This work additionally presents the results of extending the bit-flipping DLBIST scheme such that it also supports the transition fault testing besides the stuck-at fault testing. Transition faults model defects which are responsible for the incorrect operation of the core under test (CUT) at the desired speed. The importance of these defects is continuously enhanced by the ever increasing clock rates and integration density of today's circuits. Experimental results obtained for large industrial benchmark designs are reported. No *pure* DLBIST approach for the test of delay faults in circuits with standard scan design has been published so far.

In order to decrease the logic overhead of DLBIST, an innovative way of constructing efficient implementations for the involved Boolean functions (e.g. bit-flipping functions) is presented. A key feature of these functions is their incomplete specification which is based on large *don't care sets* (sets of input assignments for which it does not matter whether they are mapped to '0' or '1'). Reduced ordered Binary Decision Diagrams (ROBDD) are used for representing and manipulating the involved functions and multi-level implementations are obtained based on the use of free BDDs (FBDD). Experimental results show that for all the considered functions, implementations are found with a significant reduction of the gate count as compared to a state-of-the-art multi-level synthesys tool (SIS [Sen92]) or to methods offered by a state-of-the-art BDD package. This performance is due to a reduction of the node count in the corresponding FBDDs and a decrease in the average number of gates needed to implement the FBDD nodes.

The experimental results obtained for large industrial benchmark designs show that DLBIST may be well suited for use in special segments of IC development, like the ones dealing with security chips or hard cores.

Acknowledgments

First of all, I would like to thank Professor Hans-Joachim Wunderlich who gave me the chance to express myself working towards the accomplishment of this work and who supervised my activity.

I am very grateful to Professor Walter Anheier[1] for his survey of this work.

I wish to express my gratitude to Harald Vranken[2] and Friedich Hapke[3] for their assistance during the completion of my work.

I would also like to mention the precious technical support of Michael Garbers[3], Michael Wittke[3], Andreas Glowatz[3], Rüdiger Solbach[3] and Ralf Reche[3].

I am also very grateful to all my current and former-colleagues for our discussions and exchange of knowledge, in particular to Arnould Virazel, Abdul-Wahid Hakmi, Gundolf Kiefer, Rainer Dorsch and Yuyi Tang. Thanks go to Nicoleta Pricopi and Karin Angele for their support and comradeship.

I would like to thank Carmen Constantinescu and David Rio Mascarenas for their help and assistance in writing down the manuscript. I am also very thankful to my colleagues Wolfgang Moser, Tobias Bergmann, Alexandra Wiedmann, Hairuo Qiu and Andreas Heinchen for correcting and refining the style of the section written in German.

Finally, I would like to dedicate this work to my family, especially to my mother. I am very grateful to my uncle, Niculae Balan, who proved so much talent and patience in revealing me, since I was a child, the existence and the dynamic of a fascinating Universe.

[1] University of Bremen.
[2] Philips Research, in Eindhoven.
[3] Philips Semiconductors GmbH, in Hamburg.

Zusammenfassung

System on a Chip (SOC) sind komplexe Systeme mit Millionen von Transistoren auf einer einzelnen integrierten Schaltung (engl. *integrated circuit* (IC)). Solche ICs enthalten in der Regel verschiedene Komponenten und Technologien, wie beispielsweise Speicher, Prozessoren, anwendungsspezifische Logik, Hochfrequenz- und Analogmodule. Um die Kosten der Entwicklung zu reduzieren, werden immer häufiger vorentworfene Funktionsblöcke verwendet (engl. *core-based design*). Dieser Entwurfsstil stellt eine große Herausforderung an die Testverfahren dar.

Das Hauptziel eines jeden Testverfahrens ist es, einen Kompromiss zwischen den Testkosten und der Testqualität (Produktqualität) zu finden. Wichtige Kriterien für den Test von ICs sind: Testentwicklungskosten, Testapplikationskosten, zusätzlicher Bedarf an Schaltungsfläche, Fehlererfassung, etc.

Zu den Anforderungen, die an die heutigen IC Testverfahren gestellt werden, gehören spezifische Anforderungen, die auf die verschiedenen Typen von Modulen abgestimmt sind. Zudem besteht zunehmender Bedarf an Verzögerungstests, *in-field* und *on-line* Tests.

In den traditionellen Testverfahren werden die ICs normalerweise extern mit speziellen Testautomaten (engl. *automated test equipment* (ATE)) getestet. Die externen Testverfahren zeichnen sich durch hohe Flexibilität, relativ geringen zusätzlichen Bedarf an Schaltungsfläche und hohe Fehlererfassung für eine bestimmte Testlänge aus.

Die zunehmende Komplexität und der wachsende Umfang der Testdaten machen die Durchführung externer Tests immer schwierig. Die Qualität der Fehlererfassung wird durch die Unzugänglichkeit der internen Blöcke verringert. Außerdem ist es sehr teuer, mit ATEs die maximal mögliche Taktfrequenz moderner ICs zu messen. Diese Probleme werden durch die hohe Komponentendichte sowie die Anwendung verschiedener Technologien in der SOC-Fertigung vergrößert, so dass Messmethoden mit externen Testautomaten ungenau, und das ATE selbst sehr teuer wird.

Selbsttestverfahren (engl. *built-in self-test* (BIST)) sind dem oben genannten Verfahren in Bezug auf die erwähnten Problemen überlegen. Mit BIST können *on-line, in-field, burn-in* und *at-speed* Tests realisiert werden. *In-field* Tests werden für periodische Wartungen verwendet, *burn-in* Tests sind wichtig für die Steigerung der Produktqualität und Zuverlässigkeit und *at-speed* Tests sind notwendig um die Verzögerungsfehler zu finden. Außerdem sind Kompromisse zwischen der erreichten Fehlererfassung, dem zusätzlichen Bedarf an Schaltungsfläche und der Testlänge möglich.

Neben den externen und den Selbsttestverfahren gibt es noch verschiedene Arten von Testverfahren, die auf *test resource partitioning* (TRP) basieren.

Deterministic logic BIST (DLBIST) ist eine sehr attraktive Teststrategie, weil sie die Vorteile von deterministischen externen Testverfahren und pseudozufälligen logic BIST kombiniert. Die bisher vorgeschlagenen DLBIST Verfahren können nur bei kleineren Entwürfen angewendet werden, da die Laufzeit und der Speicherbedarf der Hardwaresynthese exponentiell oder wenigstens kubisch mit der Schaltungsgröße steigen.

Es gibt zwei grundlegende DLBIST Verfahren: *store and generate* Verfahren und *test set embedding* Verfahren. Bei dem *store and generate* Verfahren werden die Testmuster in einer komprimierten Form auf dem Chip gespeichert und darauf ein Dekompressionsalgorithmus angewendet. Bekanntere Beispiele dieser DLBIST Methode basieren auf Kodierung durch rückgekoppelte Schieberegister (engl. *LFSR*) [Koe91], Multipolynom Reinitialisierung [Hel92] [Hel95] und Faltender Zähler (engl. *folding counter*) [Lia02].

Bei *Test set embedding* Verfahren werden pseudozufallsgenerierte Muster durch deterministische Muster ergänzt. Bekannte *test set embedding* Verfahren sind die *bit-flipping* [Kie00][Wun96][Kie97][Kie98] und *bit-fixing* [Tou96] Verfahren.

Bei diesen beiden DLBIST Verfahren werden Testmustergeneratoren eingesetzt, die eine gute Fehlererfassung ermöglichen. Die Besonderheit dieser Testmustergeneratoren ist ein Logikmodul, das eine *bit-flipping* Funktion (BFF), beziehungsweise *bit-fixing* Funktion (BFX), durchführt. Die Implementierung dieser Testmustergeneratoren umfasst zwei Schritte: (1) die Abbildung einer Reihe deterministischer Testmuster zu einer Folge von Pseudozufalltestmustern und (2) die Synthese des Logikmoduls, welches die Abbildung durchführt.

In dieser Arbeit wird ein neues Verfahren für den Aufbau der *bit-flipping* DLBIST Hardware vorgeschlagen. Die BFF beschreibt das Einbetten von deterministischen Testmustern zu einer pseudozufälligen Testfolge, die durch einen LFSR und eventuell einen Phasenschieber (engl. *phase shifter*) erzeugt wird. Die Suche nach einem effizienten deterministischen Testmuster-Einbettungsverfahren mit geringem zusätzlichem Bedarf an Schaltungsfläche ist eine schwierige Aufgabe.

Ein Beitrag dieser Arbeit ist eine skalierbare Lösung, sowohl für die Abbildung von deterministischen Testmustern (d.h. die Generation von BFF), als auch für die Logiksynthese der resultierenden BFF [Ghe04]. Ein ATPG Werkzeug wird verwendet, um deterministische Testmuster für alle Fehler zu erzeugen, die nicht durch die pseudozufällige Testfolge entdeckt werden. Diese deterministischen Testmuster enthalten eine große Zahl nicht spezifizierter Bits (engl. *don't care (DC) bits*)). Ein Pseudozufallstestmuster wird jedem dieser deterministischen Testmuster zugeteilt, so dass die Größe der resultierenden BFF minimiert wird. In Anbetracht eines deterministischen Testmusters werden nur diejenigen Pseudozufallstestmuster untersucht, die eine minimale Zahl von unpassenden (engl. *conflicting*) Bits enthalten. Um weiter die Abbildung von Testmustern zu optimieren, wird eine Kombination von folgenden Maßnahmen verwendet:

- Minimierung der Taktzyklen, die sowohl zusammenpassende (engl. *matching*) als auch unpassende Bits enthalten. Dadurch wird versucht den Logikanteil, der bei der BFF Implementierungen für unterschiedliche Prüfpfade gemeinsam benutzt wird, zu maximieren.
- Minimierung der Zahl von Prüfpfaden, die sowohl zusammenpassende als auch unpassende Bits pro eingebettetem Testmuster enthalten. Dies erhöht die Optimierungsmöglichkeiten für die BFF Implementierung jedes Prüfpfades.

Das neue Verfahren stützt sich auf die Effizienz und die Kompaktheit der BDD-basierten Funktionsdarstellung und hat eine beinahe lineare Komplexität in Bezug auf Laufzeit und Speicherbedarf.

Die Effizienz des neuen Verfahrens wird für industrielle Schaltungen bis zu einer Größe von 2 Millionen Gattern nachgewiesen. Mit der neuen Einbettungsmethode sind Verbesserungen mehrerer Größenordnungen, verglichen mit den vorherigen Verfahren [Wun96] sowohl in Bezug auf den Laufzeitbedarf, als auch in Bezug auf den Speicherbedarf, erreichbar. Die neue DLBIST Hardware-Synthese hat jetzt denselben Laufzeit- und Speicherverbrauch wie die anderen benötigten Verfahrensschritte, ATPG und Fehlersimulation. Die Laufzeitverbesserungen können auch dazu verwendet werden, um noch bessere Lösungen in Bezug auf den zusätzlichen Bedarf an Schaltungsfläche und Fehlererfassung zu erhalten.

Ein anderer Beitrag dieser Arbeit ist eine Studie zur Wirksamkeit des *bit-flipping* DLBIST im Test von nicht modellierten Defekten [Eng05]. Die widerstandsbehafteten Brückenfehler (engl. *resistive bridging faults*) wurden verwendet, um nicht modellierte Defekte zu simulieren. Experimentelle Ergebnisse zeigen, dass sowohl deterministische als auch pseudozufällige Testmuster nützlich sind, um nicht modellierte Defekte zu testen. Außerdem werden mögliche Kompromisse zwischen der Testlänge, zusätzlichem Bedarf an Schaltungsfläche, Fehlererfassung und Erfassung der nicht modellierten Defekte analysiert. Es zeigt sich, dass durch die Erhöhung der Anzahl von Testmustern die Defekterfassung erhöht und der zusätzliche Bedarf an Schaltungsfläche bedeutend reduziert wird. Das vergrößert die Attraktivität der vorgeschlagenen DLBIST Architektur und reduziert den Bedarf an teuren ATEs.

Diese Arbeit enthält auch eine Erweiterung der entwickelten *bit-flipping* DLBIST Architektur, so dass neben Haftfehlern auch Übergangsfehler testbar werden [Ghe05]. Die Übergangsfehler (engl. *transition faults*) sind eine Art von Verzögerungsfehler und modellieren Defekte, die für eine nicht funktioniernde Schaltung bei der verwendeten Taktfrequenz verantwortlich sind. Die Bedeutung dieser Defekte wird durch die jeweils zunehmende Taktrate und Integrationsdichte heutiger Schaltungen ständig erhöht.

Es ist bis jetzt kein DLBIST Verfahren für die Prüfung der Übergangsfehler veröffentlicht worden. Die Besonderheit der Tests von Übergangs- und allgemeinen Verzögerungsfehlern besteht in der Notwendigkeit, Paare von Testmustern und nicht einzelne Testmuster, wie im Falle von Haftfehlern, anzuwenden. Es wird in diesem Verfahren das erste Testmuster jedes Paares genau wie im Fall von Haftfehler-Test erzeugt, und die Schaltungsantwort auf das erste Testmuster wird als zweites Testmuster verwendet (engl. *functional justification*).

Da bei diesem Test Testmusterpaare verlangt werden, ist die zufällige Fehlererfassung bedeutend kleiner als für Haftfehler. Um das Einbetten von Testmustern effizienter zu machen, wird ein spezielles Modul, *correction logic* (CRL) genannt, eingeführt. Das CRL-Modul wird genau wie das BFF-Modul synthetisiert. Die Ausgangssignale des CRL-Moduls müssen während der *scan*-Taktzyklen, entsprechend jedem Testmuster, unverändert bleiben. Da sich einige der Eingangssignale während der *scan*-Taktzyklen ändern, werden die Ausgangssignale des CRL-Moduls in einem Flipflop gespeichert. Dieses Flipflop kann nur dann beschrieben werden, wenn ein neues Testmuster in den Prüfpfad gescannt wird. Das Flipflop wird durch das *scan enable*-Signal gesteuert, das benutzt wird, um die *scan*-Flipflops zwischen *scan mode* und *functional mode* zu schalten.

Um die Größe des CRL und BFF zu beschränken, werden die deterministischen Testmuster am Ende der pseudozufälligen Testfolge eingebettet. Die Länge der pseudozufälligen Testfolge, die modifiziert werden kann, ist ein Bruchteil der ganzen Testlänge. Um die Bits in der pseudozufälligen Testfolge davor zu bewahren, gekippt (engl. *flipped*) zu werden, werden die Ausgangssignale des BFF mit der Hilfe eines UND-Gatters pro Prüfpfad auf Null gesetzt.

Die geringere pseudozufällige Testbarkeit von Übergangsfehlern relativ zu den Haftfehlern verlangt bedeutend längere Testmusterfolgen. Diese eignen sich sowohl zum Begrenzen des zusätzlichen Bedarfs an Schaltungsfläche als auch für eine verbesserte Erfassung von modellierten und nicht modellierten Defekten. Experimentelle Ergebnisse für große Industrieschaltungen zeigen mögliche Kompromisse zwischen der Testlänge, zusätzlichem Bedarf an Schaltungsfläche und Fehlererfassung auf.

Ein weiterer Beitrag dieser Arbeit ist ein Logikoptimierungswerkzeug, das verwendet wird, um die Implementierung des BFF zu verbessern. Dieses Logikoptimierungswerkzeug ist besonders zur Implementierung von unregelmäßigen und unvollständig spezifizierten Booleschen Funktionen geeignet. In diesem Fall bedeutet die Unregelmäßigkeit einer Booleschen Funktion, dass ihre Eingaben, deren Abbild ‘1’ ist, zufällig über dem Definitionsraum verteilt sind. Unvollständige Spezifizierung beruht auf Inputs, für die es gleichgültig ist, ob sie auf ‘0’ oder ‘1’ abgebildet werden. Beispiele für diese Art von Funktionen sind: BFF, BFX [Tou96] und die so genannte X-Maskierungsfunktion (XMF) [Tan04]. Alle diese Beispiel-funktionen werden in verschiedenen *test set embedding* Verfahren verwendet.

Für solche Funktionen werden effiziente mehrstufige Logikimplementierungen erzeugt. Diese Logikimplementierungen können sehr gut mit Hilfe ungeordneter BDDs (FBDDs) modelliert werden. Das Problem wird auf die Synthese eines minimalen FBDD reduziert. Dies wird durch den Ansatz zweier verschiedener Methoden erreicht: (a) auf DC-basierte Knotenzahlreduzierung und (b) Verteilung des Definitionsraumes der Zielfunktion in eine reduzierte Zahl von Subräumen, die entweder zu ‘0’ oder zu ‘1’ abgebildet werden können. Heuristiken werden verwendet, um fast optimale Teilungen des Definitionsraums in solchen Subräumen zu finden und folglich die Anzahl der Knoten und Pfade der resultierenden FBDD-artigen Implementierungen zu minimieren. Außerdem ist diese Näherung auch im Stande, unter Anwendung des DC-Raumes, die Gatteranzahl zu reduzieren, die in der Implementierung jeder Knotenfunktion erscheint.

Verglichen mit den im CUDD-Paket [Cudd] enthaltenen Methoden (*restrict*-Operator und Umstellung von BDD Variablen), liefert das FBDD-basierte Verfahren

Logikimplementierungen, deren Schaltungsbeschreibungen ungefähr 70% weniger Logikoperatoren benötigen. Diese Schaltungsbeschreibungen, erzeugt mit beiden Verfahren, wurden mit dem Synopsys Design Compiler synthetisiert. Infolgedessen konnte man erkennen, dass das FBDD-basierte Verfahren den zusätzlichen Bedarf an Schaltungsfläche um einen Faktor zwischen zwei bis drei verbessert und die Laufzeit bedeutend reduziert wird. Des Weiteren kann beobachtet werden, dass das vorgeschlagene Verfahren besser skaliert und einen größeren Nutzen aus der DC-Menge zieht als das bekannteste mehrstufige Synthesewerkzeug SIS [Sen92].

Insgesamt zeigen die experimentellen Ergebnisse für große industrielle Schaltungen, dass das *bit-flipping* DLBIST Verfahren für verschiedene Segmente der IC Tests verwendet werden kann, z.B. für die jenigen die sich mit Sicherheitschips (z.B. Smart-Cards) oder mit *Hardcores* befassen.

Am Ende dieser Arbeit werden einige Ideen vorgeschlagen, um die hier präsentierte Forschung fortzusetzen.

Table of Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

∞	Infinite
$\varnothing$	Empty set
$\mathbf{\Omega}$	Ohm – measurement unit of the electrical resistance
$\neg$	Logic negation operator
$\cdot$	Logic conjunction operator or, depending on the context, multiplication operator
+	Logic disjunction operator or, depending on the context, addition operator
$\oplus$	EXOR operator
$\cup$	Set union operator
$\cap$	Set intersection operator
_	Concatenation operator
$\downarrow$	*Constrain* operator
$\Downarrow$	*Restrict* operator
$\exists_x$	Existential quantification with respect to the variable x
$\mathbf{f}\vert_{\mathbf{l}}$	Cofactor of the Boolean function f with respect to the literal l
$\vert\mathbf{h}\vert$	Absolute value of an algebraic expression h
$\vert\mathbf{G}\vert$	Size (number of nodes) of the graph G
$\vert\vert\mathbf{f}\vert\vert$	Number of input assignments mapped by the Boolean function f to 1
$\mathbf{F(f_{on}, f_{off})}$	Representation of an incompletely specified function F defined by its ON-set, represented by f_{on}, and OFF-set, represented by f_{off}.
ADI	Analogue detectability interval
AND	Boolean conjunction operator
ATE	Automated test equipment
ATPG	Automatic test pattern generator
BCU	BIST control unit
BDD	Binary Decision Diagram
BFF	Bit-flipping function
BFX	Bit-fixing function

BILBO Built-in logic block observer

BIST Built-in self-test

CAD Computer-Aided Design

ADI_C Covered ADI

ADI_G Global ADI

Cov(F) Cover of the incompletely specified function F

CRL Correction logic

CUDD CU (Colorado University) decision diagram package

CUT Core (or circuit) under test

DC Don't care

DC-set Set of input assignments of a Boolean function that can be mapped either to 1 or to 0

DC-space Similar to DC-set

DLBIST Deterministic LBIST

ESPRESSO Heuristic approach used for optimizing two-level representations of Boolean functions

EXOR Boolean exclusive-OR operator

EXPAND Operator used by ESPRESSO (transforms a cover into a prime and irredundant cover)

FBDD Free (also called unordered) BDD

FC Fault coverage

FDD Functional Decision Diagram

FE Fault efficiency

FF Flip-flop

FIX-set On-set $\cup$ Off-set

IC Integrated circuit

ID Identity matrix

IDDQ Power supply current (IDD) in a CMOS circuit when all nodes are quiescent (static)

ISCAS International Symposium on Circuits and Systems

LBIST Logic BIST

LFSR Linear feedback shift register

LHCA Linear hybrid cellular automata

MISR Multiple input shift register

MLFSR Modular LFSR

MUX Multiplexer

NAND	Boolean negation and conjunction operator
NOR	Boolean negation and disjunction operator
OBDD	Ordered BDD
OFF-set	Set of input assignments mapped to 0 by a Boolean function
OFF-BDD	ROBDD-based representation of the OFF-set
ON-set	Set of input assignments mapped to1 by a Boolean function
ON-BDD	ROBDD-based representation of the ON-set
OR	Boolean disjunction operator
PC	Pattern counter
PLA	Programmable logic array
PRPG	Pseudo-random pattern generator
PS	Phase shifter
REDUCE	Operator used by ESPRESSO *(*transforms a prime and irredundant cover into a new irredundant but usually not prime cover)
ROBDD	Reduced OBDD
ROM	Read only memory
RPR	Random pattern resistant
RTL	Register transfer level
SC	Shift (also called bit) counter
SE	Scan enable signal
SG	Already synthesized sub-graph
SLFSR	Standard LFSR
SOC	System on a chip
STUMPS	Self-test using MISR and parallel shift register sequence generator
Th	Threshold voltage of a gate input
TPG	Test pattern generator
TRE	Test response evaluator
TRP	Test resource partitioning
V	Volt – measurement unit of the voltage
VHDL	VHSIC (very high speed integrated circuit) hardware description language
VDD	Positive supply voltage for field effect transistors (FET)
X	Unspecified or unknown bit value
XMF	X-masking function
ZBDD	Zero-suppressed BDD

Chapter 1

Introduction

1.1 Motivation and Goal of the Work

The sustained improvement of deep-submicron technologies has led to an explosion in the number of transistors that may be integrated on a chip and further to the possibility of putting a whole *system on a chip* (SOC). Core-based design is one paradigm of the new trends used to reduce complexity and costs of chip development. Nevertheless, test-related costs are problems still far away from having a unitary and satisfactory solution.

The external testing of integrated circuits (ICs) is a traditional approach in which *automated test equipment* (ATE) provides all the necessary test data. This may set high requirements on the storage capacity and speed of the ATE. Furthermore, the ever increasing transistor count per I/O pin and the low accessibility of internal blocks are affecting the tradeoff between the final fault coverage and the test application time. All of these, combined with the necessity of specially tuned testers for different types of cores and the growing need for periodic in-field maintenance and on-line testing capabilities make the external testing difficult, costly and insufficient.

All the above mentioned problems demand *built-in self-test* (BIST) solutions. In this context, BIST for random logic (LBIST) is becoming an attractive alternative in IC testing.

The standard BIST architecture [Bar82][Eic83] uses an LFSR that feeds pseudo-random patterns into the scan paths. It is easy to implement and minimizes both hardware overhead and impact on the system performance. However, due to random-pattern-resistant (RPR) faults, pseudo-random patterns cannot always achieve sufficient fault coverage within an acceptable test time.

The fault coverage can be increased by biasing the pseudo-random test sequence towards the RPR faults [Brg89][Wun88]. Conflicting input values required by different RPR faults may need different weighting sets. Unfortunately, the control logic and the storage requirements for the weighting sets can increase unacceptably.

Pseudo-exhaustive testing [Mcc81] achieves the benefits of exhausting testing while usually requiring less test patterns. This reduction is obtained by splitting the circuit into various segments that are tested exhaustively. The efficiency of the method is limited by the size of the largest segment that has to be tested.

An alternative approach for increasing the fault coverage is the insertion of test points, which has been proposed for both LBIST and external testing [Geu00][Hay74] [Sei91][Vra02]. While the area increase due to test point insertion may be tolerable, they can introduce additional signal delays, which could require a complete re-synthesis and a new timing verification [Vra04].

Deterministic LBIST (DLBIST) guarantees higher or complete fault coverage by embedding deterministic test *cubes* (test patterns with unspecified bits) into the pseudo-random sequence. There is a wide range of deterministic logic BIST methods that apply deterministic test patterns and hence improve the low fault coverage often obtained by pseudo-random patterns. In an initial deterministic BIST scheme, additional external patterns were applied on top of the pseudo-random test [Het99]. Unfortunately, the very last percentages of fault coverage require the largest amount of deterministic patterns. For instance, it has been reported in [Bas89] that detecting the last 10% of undetected faults typically requires 70% or more of the test patterns in an automatic test pattern generated set. Consequently, the benefits of deterministic BIST are severely reduced by this approach.

Compression and decompression methods in which a small amount of external test data is continuously fed into the circuit [Koe91][Koe01][Raj02] are more efficient. However, this approach is no longer a BIST method; it may still require a relatively expensive ATE and lose some benefits of BIST like in-field testing.

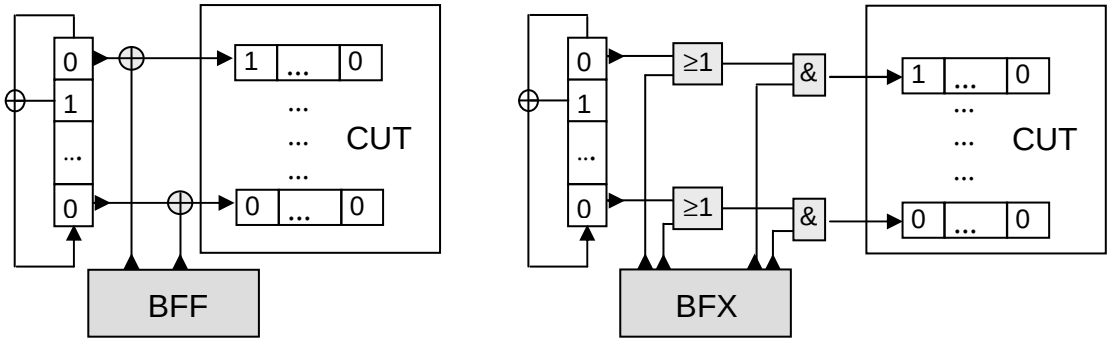
In contrast to the above mentioned BIST methods, pure DLBIST schemes try to avoid both the modification of the *core under test* (CUT) and the application of additional external test data. These methods can be classified into *store and generate* schemes and *test set embedding* schemes.

Store and generate schemes consist of hardware structures which store the test patterns on-chip in a compressed form and implement a decompression algorithm. Widely known representatives of this method are LFSR-reseeding [Koe91], multi-polynomial reseeding [Hel92][Hel95] and folding counter based-LBIST [Lia02].

Test set embedding schemes rely on a pseudo-random test pattern generator plus some additional circuitry that modifies the pseudo-random sequence in such a way that a set of deterministic cubes is embedded. Widely known test set embedding techniques are *bit-flipping* [Kie00][Wun96][Kie97][Kie98] and *bit-fixing* [Tou96].

In the bit-flipping approach, the output sequence of an LFSR is inverted at a few bit positions in order to increase the fault coverage (Figure 1.1.a), while in the bit-fixing approach constant values are applied (Figure 1.1.b). The test generation process is controlled by a *bit-flipping function* (BFF) or a *bit-fixing function* (BFX), respectively.

The term *pattern mapping* will be used for referring to the assignment of a pseudo-random pattern to a given deterministic cube. The synthesis procedure of a DLBIST scheme consists of pattern mapping and generation of the hardware structure used to implement the mapping, e.g. by means of a BFF or BFX. The synthesis procedure for generating the BFX as published in [Tou96] is based on rectangle covering, while the synthesis procedure for generating the BFF as published in [Wun96][Kie97][Kie98] is based on manipulating sets of test cubes. In both cases, the procedures use heuristics that generally require at least cubical, but often exponential, effort in terms of memory consumption and computation time.



(a) (b)

Figure 1.1: (a) Bit-flipping and (b) bit-fixing BIST schemes.

In this work, a novel *pattern mapping* approach is proposed that has nearly linear complexity in terms of both computation time and memory consumption. The used algorithms are based on Binary Decision Diagrams (BDDs). The efficiency of the new algorithms is demonstrated by experimental results obtained with ISCAS benchmarks and industrial designs containing up to 2M gates.

The embedded test sequences obtained by mapping deterministic cubes to pseudo-random sequences are also evaluated with respect to their coverage of non-target defects. Moreover, possible tradeoffs between the test length, hardware overhead, fault coverage and non-target defect coverage are analyzed.

All the methods discussed so far mainly refer to the test of stuck-at faults. Unfortunately, the steady increase of the clock rate and the integration density in today's IC designs enhance the significance of the timing accuracy defects [Cha96], which are difficult to be covered by the classical stuck-at fault model. Consequently, delay fault models and, implicitly, delay fault testing become more and more important.

Here, an extension of the bit-flipping DLBIST approach to the test of transition faults is also presented. The scheme is based on functional justification and on an efficient pattern embedding. A special module, the *correction logic* (CRL), is introduced to further improve the pattern embedding. Due to the rather low random-pattern testability of transition faults, the saturation of their random fault coverage requires significantly longer test sequences, which in turn is beneficial for both limiting the hardware overhead and improving the coverage of modeled and non-modeled defects [Tan04].

A major concern of the test set embedding schemes is their hardware overhead. Reducing the hardware overhead of the DLBIST scheme considered here is equivalent to optimizing the logic synthesis of the BFF. Two properties of this function are relevant for its logic implementation: the *irregularity*, defined by the random distribution over the definition space of the input assignments mapped to '1' and the *incomplete specification*, defined by the existence of input assignments for which it does not matter whether they are mapped to '0' or '1'. Other examples of such functions are the BFX [Tou96], the function implemented by the CRL and the so-called *X-masking function* (XMF) [Tan04].

This work proposes an efficient and innovative way to implement irregular Boolean functions with large *don't care* sets. Reduced ordered BDDs (ROBDD) are used for representing and manipulating the involved functions. Multi-level representations are

obtained based on free BDDs (FBDD). The problem is reduced to the construction of an efficient BDD-based representation by using the *don't care* space to perform node reduction and to partition the definition space of the considered function into a minimum number of sub-spaces which may be mapped either '0' or '1'. Heuristics are used to find near-optimal partitions of the definition space into such sub-spaces and, consequently, to minimize the path and node count of the resulting FBDD. Furthermore, this approach is also able to use the *don't care* set to reduce the average gate count per node. Experimental results show that for all the considered functions, implementations are found with a significant reduction of the gate count compared to the well known multi-level synthesis tool, SIS [Sen92], or to methods offered by a state-of-the-art BDD package. This performance is due to a reduction of the node count in the corresponding FBDDs and a decrease in the average number of gates needed to implement the FBDD nodes.

1.2 Outline

Chapter 2 briefly describes the three logical fault models which will be used in this work. Section 2.1 introduces the stuck-at fault model. In Section 2.2, the resistive bridging fault model is described. Section 2.3 presents two delay fault models: the transition and the path delay fault models. Only the transition fault model will be used later in the work. The path delay fault model is briefly mentioned in order to better understand specific aspects of the delay fault testing.

Basic BIST concepts are reviewed in Chapter 3. *Test-per-scan* and *test-per-clock* BIST schemes are described in Section 3.1 and Section 3.2, respectively. State-of-the-art methods for test pattern generation and test response evaluation are analyzed in Section 3.3 and Section 3.4, respectively.

Chapter 4 compares two of the basic approaches that are used for the representation and the manipulation of Boolean functions. Section 4.1 introduces the cube-based, also called disjunctive two-level representation. The generalization of this representation to the multi-level representation and implementation is described in Section 4.2. Section 4.3 presents the representation, manipulation and logic synthesis of Boolean functions based on Binary Decision Diagrams (BDDs).

Chapter 5 presents a new algorithm for mapping deterministic test cubes to a pseudo-random test sequence. The algorithm is based on BDDs and outperforms the previously published cube-based approach [Wun96] by several orders of magnitude. It has been applied to the bit-flipping Deterministic Logic LBIST (DLBIST) architecture which is presented in Section 5.1. The pattern mapping problem is formally defined in Section 5.2. Sections 5.2 and 5.3 provide a detailed description of a prior cube-based and of the new BDD-based mapping algorithms, respectively. Section 5.5 reports the experimental results obtained with a set of industrial designs containing up to 2M gates, ISCAS-85 and combinational parts of ISCAS-89 benchmark designs. These results prove that significant improvements can be achieved with the help of the BDD-based mapping approach. In Section 5.6, the embedded test sequences generated for single stuck-at faults are evaluated with respect to the coverage of non-target defects. Resistive bridging faults are used as a surrogate of non-target defects [Eng05]. This is the first time when the results of such a study are presented. This

investigation especially adresses the impact of the test sequence length on the non-target defect coverage and on the hardware overhead. The chapter is concluded in Section 5.7.

Chapter 6 extends the approach introduced in Chapter 5 to make it also available for the test of transition faults. Due to the fact that pairs of test patterns are required, transition faults are more difficult to test than stuck-at faults. In Section 6.1, a qualitative comparison of stuck-at and transition faults is made with respect to their pseudo-random testability. The extension of the bit-flipping DLBIST scheme for transition fault testing is described in Section 6.2. Relevant experimental results for large industrial benchmark designs are reported in Section 6.3. The chapter is summarized in Section 6.4.

In Chapter 7, an innovative BDD-based logic synthesis method is described that improves the implementation of the BFF. This approach is especially suited for the logic implementations of *irregular* functions that have large *don't care* sets. Some examples of such functions are: the BFF, the BFX [Tou96] and the function XMF introduced in [Tan04], etc. Two of these examples are analyzed in Section 7.1. Section 7.2 presents a new heuristic method to find efficient logic implementations for such functions. In Section 7.3, experimental results are used to compare the new approach with SIS [Sen92] and methods available in the CUDD-package (like *restrict* [Cou90]). Furthermore, the outcome of the proposed method is evaluated as input to Synopsys Design Compiler. The chapter is concluded in Section 7.4.

Chapter 8 summarizes the work and suggests some related research directions that look promising and may be investigated in a future work.

Chapter 2

Basic Fault Models

This chapter describes the three logical fault models used in this work. Logical faults represent the effect of physical defects on the logic behavior of the modeled system. Restricting the analysis of physical defects to the level of the logic behavior has several advantages. The complexity is reduced by transforming a physical problem into a logical problem. The space of physical defects is larger than the space of logical faults, such that a fault model can cover several physical defect types. Moreover, tests derived for certain logical faults may cover physical defects for which no accurate fault model is known. Most of the logical fault models are technology-independent and hence testing and diagnosis methods developed for such fault models are applicable to many technologies [Abr90].

A distinction is made between faults that affect the logic correctness of a circuit and delay faults that affect the operating speed of the system. Depending upon the type of modeling used for the system, the former faults may be divided in structural and functional faults. Structural fault models are usually defined at the gate level net-list and assume that components are fault-free and only their interconnections are affected. Functional faults are usually defined at RTL or higher levels (like behavioral or system level) and they affect the proper execution of the operations used at these levels.

Shorts and *opens* are two examples of structural faults. A *short* is formed by connecting points not intended to be connected while an *open* results by breaking a connection.

In this work only structural, permanent and single faults of combinational logic are considered. Intermittent, transient, or multiple-faults are not taken into account. The analog and the memory elements that may be present in the circuit under test are not considered.

Under the single-fault assumption one assumes that in a system at most one logical fault is present. This assumption is justified by the fact that in most of the cases a multiple fault can be detected by the tests designed for the individual single faults that compose the multiple-fault [Abr90].

Section 2.1 introduces the stuck-at fault model. In Section 2.2, the resistive bridging fault model is briefly described. Section 2.3 introduces two representative delay fault models: the transition and the path delay fault model. Only the transition fault model will be used later in this work. The path delay fault model is mentioned in order to better understand specific aspects of the delay fault testing.

2.1 Stuck-at Faults

The logical fault corresponding to a signal line being *stuck at* a fixed logic value (0/1) is referred to as a single stuck-at 0/1 fault (Figure 2.1). Physical defects which can be modeled with the help of a stuck-at 0/1 fault on the signal line *i* include an open on the fan-out lines driven by the line *i*, a short to power/ground or an internal error in the component driving the line *i*.

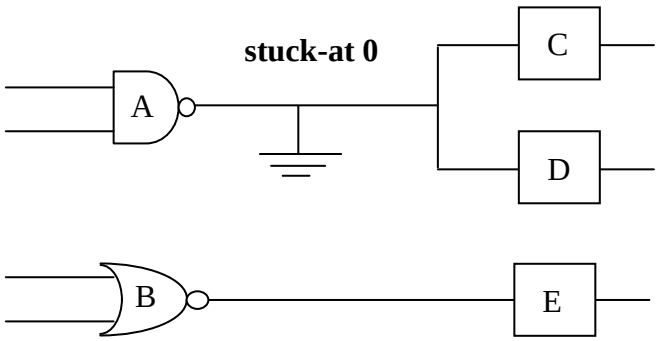


Figure 2.1: Example of a stuck-at fault.

Despite the fact that the single stuck-at fault model does not cover all the physical defects that can appear in a digital circuit, it is very useful due to the following properties:

- It is very simple. As compared to other fault models, the number of single stuck-at faults in a circuit grows linearly with its size. Moreover, the number of these faults that have to be explicitly considered can be reduced by fault collapsing. Techniques like structural-based and dominance-based fault collapsing can reduce the number of faults to be explicitly analyzed by 50% and 40%, respectively [Abr90].
- It models many different physical defects [Tim83]. Test sets generated for single stuck-at faults may detect many faults belonging to other fault models.
- It is technology independent.
- The single stuck-at fault model and its analysis can be used to construct and analyze other types of fault models, like the transition fault model (Section 2.3.3).

A combinational circuit that contains an undetectable stuck-at fault is said to be *redundant*, since such a circuit can always be simplified by removing at least one gate or input. The test generation problem for stuck-at faults belongs to the class of NP-complete problems (worst-case behavior) [Iba75]. Undetectable (redundant) faults are usually the ones that cause test generation algorithms to exhibit their worst-case behavior [Abr90].

A straightforward extension of the single stuck-at fault model is the multiple stuck-at fault model. This fault model is more difficult to handle. The list of faults for a circuit having N possible sites for single stuck-at faults can contain up to $2N$ single and $3^N–1$ multiple stuck-at faults [Abr90]. Fortunately, the importance of the multiple stuck-at fault model is reduced due to the fact that tests with complete detection of the single stuck-at faults would usually also detect most of the multiple stuck-at faults [Hug84].

For all fault models introduced in this chapter, whose description does not depend on a continuous parameter, the following metrics are used to characterize the quality of a test set.

Definition 2.1: The *fault coverage* (FC) is the percentage of detected faults with respect to the total number of faults.

Definition 2.2: The *fault efficiency* (FE) is the percentage of detected faults with respect to the total number of testable faults.

2.2 Resistive Bridging Faults

A logical fault representing an electrical connection between a pair of signal lines (nets) is referred to as a *bridging fault*. The non-resistive bridging fault model considers a short between the two nets. The logic value of the shorted nets may be modeled as 1-dominated (OR bridge), 0-dominated (AND bridge) or intermediate, depending upon the implementation technology [Bus00][Mal92].

More general and realistic is the resistive bridging fault model, in which the connection between the two nets is characterized by an arbitrary electrical resistance [Ren95]. The resistive bridging fault model will be used in the following chapters to account for non-target defects.

The main difficulty when dealing with resistive bridging faults is that, unlike the non-resistive case, there is an unknown value to be taken into account – the bridging resistance. This is due to the fact that the cause which generated of the bridging fault cannot be known in advance. Topological and physical parameters like shape, size, electrical conductivity, exact location on the die, evaporation behavior, electron-migration and environmental temperature can influence the resistance of the short defect [Eng03].

A test pattern may detect a bridging defect for one resistance value and not for another resistance value. This fundamentally changes the meaning of standard testing concepts, like testability, redundancy, fault coverage, etc [Ren95].

In order to illustrate this, consider the example sketched in Figure 2.2 [Eng03]. The nets *a* and *b* in this example are bridged by a short defect with the resistance R_{sh}. The voltage V_a on *a* and the voltage V_b on *b* both depend not only on the input pattern, but also on the bridge resistance R_{sh}.

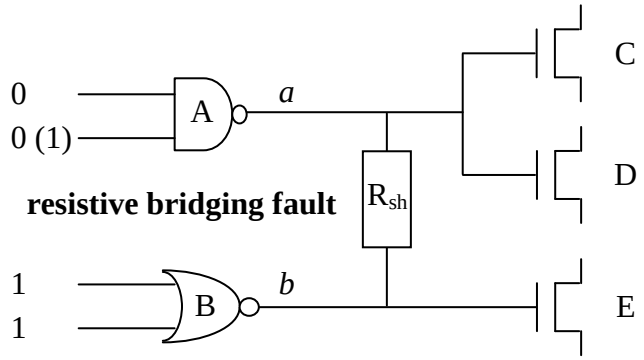


Figure 2.2: Example of a resistive bridging fault [Eng03].

Consider the input assignment 0011. Here, it is considered that logic values '1' and '0' are encoded by a high-, respectively a low-voltage. A possible voltage dependence on the R_{sh} values is depicted by the solid curves in Figure 2.3. For $R_{sh} = 0\Omega$, there is an intermediate voltage identical for both lines. With increasing R_{sh}, V_a and V_b diverge with V_a approaching VDD and V_b approaching 0. The transistors succeeding the bridge will interpret these voltages as logic-0 or logic-1, depending on their *input threshold voltages* Th. In Figure 2.3, the threshold voltages for transistors C, D and E are shown as horizontal lines labeled by Th_C, Th_D and Th_E, respectively. Hence, the resistive bridging fault may be observed at the drain of the transistors C or E and eventually at the output of the gates containing these transistors iff $R_{sh} \in [0, R_C]$, respectively $R_{sh} \in [0, R_E]$. For transistor D, the threshold voltage Th_D is below the curve, implying that transistor D will recognize the voltage on *a* as a logic-1 for any R_{sh}. Consequently, the fault effect is visible at one of the outputs iff $R_{sh} \in [0, R_C] \cup \varnothing \cup [0, R_E] = [0, R_E]$.

Next, consider the input pattern 0111 that sets a high-voltage on the second input of the NAND gate. In this case, only one p-transistor will pull up the voltage on the net *a* to the power supply. Thus, the net *a* is still driven with logic-1, but with less strength, while the logic-0 on the net *b* has the same strength as before. One possible voltage characteristic for V_a and V_b is described in Figure 2.3 by the dashed curves situated underneath the solid ones. Hence, the fault effect is visible at one of the transitor drains and eventually at the outputs of the corresponding gates iff $R_{sh} \in [0, R_C'] \cup [0, R_D'] \cup [0, R_E'] = [0, R_C']$. Consequently, a resistive bridging fault with $R_{sh} \in [R_C', R_E]$ may be detected by the pattern 0011, but not by the pattern 0111, although the logic values on all internal lines of the fault-free circuit are identical for these two patterns.

In order to handle this ambiguity, the concept of *analogue detectability interval* (ADI) and probabilistic fault coverage are introduced [Ren95][Ren99].

Definition 2.3: The interval $[R_1, R_2]$ $(0 \le R_1 \le R_2 \le \infty)$ in which a resistive bridging fault f_R is detected by a pattern P at one output (at least) is called the *analogue detectability interval* (ADI) of the pattern P with respect to f_R [Ren99].

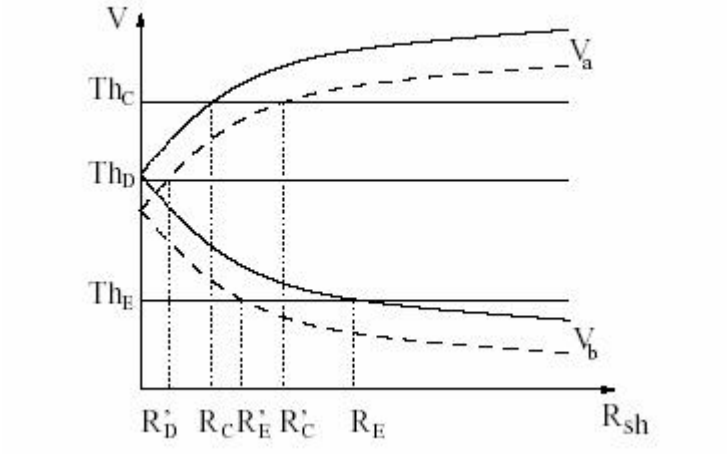


Figure 2.3: R_{sh}-V – diagram [Eng03].

The ADI of the patterns 0011 and 0111 with respect to the bridging fault between the nets *a* and *b* are given by the intervals [0, R_E] and [0, R_C'] ($\subset$ [0, R_E]), respectively.

The fault simulation for *classical* fault models determines whether a fault can be detected or not. In contrast to this, resistive bridging fault simulation determines the ADI for a given fault and test pattern, i.e. the values of the bridging resistance for which the considered fault can be detected by the specified pattern.

Given the resistive bridging fault f_R and a set of test patterns S, the following definitions can be given [Ren99]:

Definition 2.4: The ADI_C (C stands for *covered*) of the test set S with respect to f_R is defined as the union of the ADIs of each individual test pattern in S corresponding to f_R.

Definition 2.5: ADI_G (G means *global*) with respect to f_R is the maximum ADI_C corresponding to f_R.

ADI_C characterizes the testability of a resistive bridging fault with respect to the patterns in a given test set S. ADI_G measures the testability of a resistive bridging fault independently of the test set – it corresponds to an exhaustive test set. A bridging fault with an empty ADI_G ($ADI_G = \varnothing$) is *untestable* (at least if effects on delay and IDDQ testing or on reliability are not considered).

Definition 2.6: The *global fault coverage* (FC_G) [Ren99][Eng03] of a test set S with respect to a resistive bridging fault f_R is defined as:

$$FC_G(f_R) = \frac{\left(\int_{ADI_C} \rho(R)dR \right)}{\left(\int_{ADI_G} \rho(R)dR \right)},$$

where $\rho(R)$ is the probability density function of the short resistance *R* obtained from manufacturing data. $\rho(R)$ is chosen such that the second integral is equal to 1.

If for any considered bridging fault f_R, for which the ADI_C is different from the empty set, ADI_G is set equal to ADI_C, then the non-probabilistic case associated to the non-resistive bridging models is obtained ($FC_G(f_R) = 1$).

Definition 2.7: For N bridging faults f_i, $1 \le i \le N$, the *average resistive bridging fault coverage* [Eng03] is defined as:

$$FC_G = \frac{1}{N} \sum_{i=1}^{N} FC_G(f_i)$$

Up to now there is no known method to determine the ADI_G, and implicitly the FC_G, without simulating all 2^n test patterns, where n is the number of inputs. Approximation methods for computing ADI_G and FC_G are given in [Eng03].

2.3 Delay Faults

Delay fault testing is used to prove and estimate the performance of the *core under test* (CUT) and has become a standard option in today's technology. *Path-delay*, *segment-delay* and *gate-delay fault* models have been proposed so far [Her96][Sha00] [Smi85][Krs98][Iye90]. These models have different complexity in both test generation and test application. A special case of the gate-delay fault model is the *transition fault* model [Krs98][Lev86][Wai87], also called *gross-delay fault* model, in which the gate-delay fault is assumed to be of the same order of magnitude as the clock period.

In order to test delay faults, two patterns are required, an *initialization pattern* V_1 that sets the circuit to a predefined state, and an *activation pattern* V_2 that launches the appropriate transition and propagates the fault effect to a (pseudo-)primary output.

2.3.1 Path-Delay Faults

Path-delay faults are used to model defects that are correlated along a path from a (pseudo-)primary input to a (pseudo-)primary output of the CUT. Both the switching delays of devices and the transport delays of the interconnects may preturbate the propagation of a signal transition along the considered path.

Path-delay faults may be robustly and non-robustly tested. A test that guarantees to detect a path-delay fault, only if no other path-delay faults are present, is called a non-robust test [Lin87][Bus00]. Besides the application of the right input transition, the other requirement for the non-robust test of a path is that all its off-path input signals assume non-controlling in the steady state following the application of the activation pattern V_2.

A robust path-delay test guarantees to detect a path-delay fault, irrespective of the delay distribution in the circuit [Lin87][Bus00]. In addition to the requirements of the non-robust test, the robust test of a path requires that all the off-path inputs must have a steady non-controlling value in both V_1 and V_2 when the on-path event is a transition from non-controlling value to controlling value.

Unfortunately, in the worst case the number of path-delay faults may increase exponentially with the number of the signal lines in the CUT. Consequently, for large industrial designs simpler delay fault models like the gate-delay and the transition fault models are usually considered.

2.3.2 Transition Faults

The transition fault model is used to cover delay effects which are generated by localized (spot) defects and whose sizes are in the order of magnitude of the clock cycle or of the test pattern period. A *slow-to-rise* and *slow-to-fall* transition fault may be associated to each signal line in the CUT. Consequently, the number of transition faults increases linearly with the number of the signal lines in the CUT. The upper bound of the number of transition faults is twice the number of signal lines in the CUT. Moreover, the similarity to the test of stuck-at faults implies that (1) tests for transition faults can be easily generated by modifying a stuck-at test generator [Krs98][Lev86] and (2) circuits with high stuck-at fault coverage usually also have large transition fault coverage [Bus00][Wai87].

Due to its limited complexity, the transition fault model is most widespread. For an efficient delay testing, it is recommended to augment transition fault testing by path delay testing performed for a sub-set containing at least the critical paths [Bus00].

Chapter 3

Basic Concepts of Built-In Self-Test

Built-in self-test (BIST) is a technique in which additional circuitry is added to a *core under test* (CUT) in order to make it able to test itself with minimum external help. Figure 3.1 sketches the general structure of a self-testable circuit composed of a *test pattern generator* (TPG), a *test response evaluator* (TRE) and a *BIST control unit* (BCU).

This technique is especially preferable when it is difficult to access the CUT externally. It also helps to protect *intellectual property* (IP) and to reduce cost of the *external test equipment* (ATE) by minimizing the amount of test data that has to be stored off-chip. Its implementation can result in an improvement in the test quality due to its better support for at-speed testing, which is essential for detecting delay faults. BIST supports in-field and on-line testing [Kar98], which helps to reduce the cost of system maintenance. It also offers the opportunity to improve reliability by means of burn-in testing.

BIST approaches can be divided into *test-per-scan* and *test-per-clock* schemes [Wun98], which are described in Section 3.1 and Section 3.2, respectively. State-of-the-art methods for test pattern generation and test response evaluation are analyzed in Section 3.3 and Section 3.4, respectively.

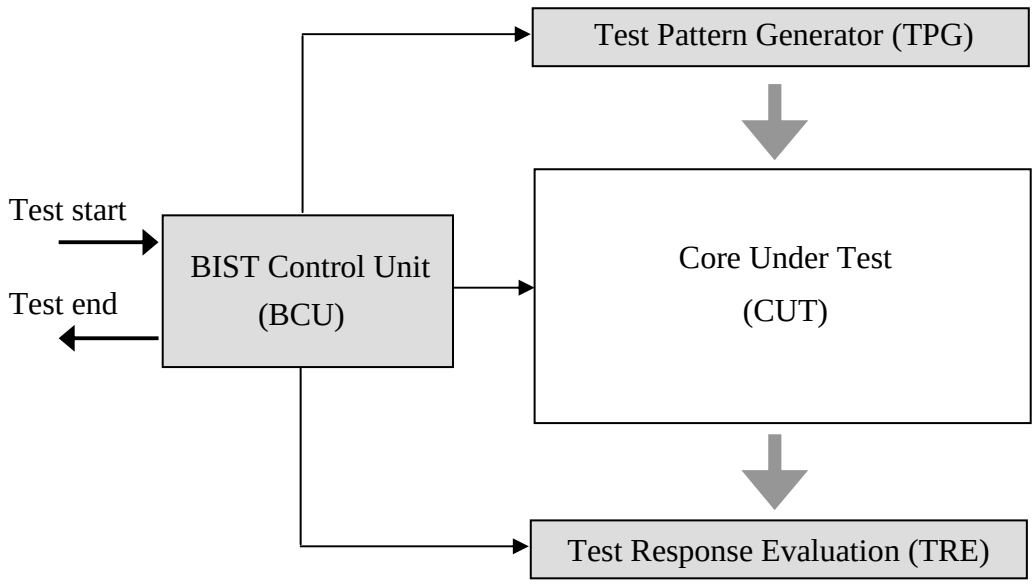


Figure 3.1: Built-in self-test (BIST) (adapted from [Hua03]).

3.1 Test-per-scan Schemes

Test-per-scan BIST schemes require scan-based design. In the case of sequential circuits, this means that all the storage cells can be configured as one or several scan paths (chains), which are used as serial shift registers in test mode (Figure 3.2). In this way, each storage device of the CUT becomes easily controllable and observable. The test stimuli/responses are shifted into/out of the scan paths [Abr90][Eic83][Tri80]. Scan-based design helps to reduce the problem of testing sequential circuits to the simpler problem of testing combinational circuits.

The BCU in Figure 3.2 must contain at least a shift counter and a pattern counter. The shift counter controls the bit stream which is generated and shifted into the scan path by a TPG. The pattern counter controls the length of the test sequence. A *system* clock cycle (also called *capture* or *functional* clock cycle) is applied to load the CUT response to the current test pattern into the scan path. During the so-called *shift* mode (also called *scan* or *test* mode) a new test pattern is shifted into the scan path, while the CUT response to the previous pattern is shifted out and compressed by a TRE.

A very common and effective parallel-serial mixed scheme is obtained by partitioning a full scan path into multiple scan chains (Figure 3.3).

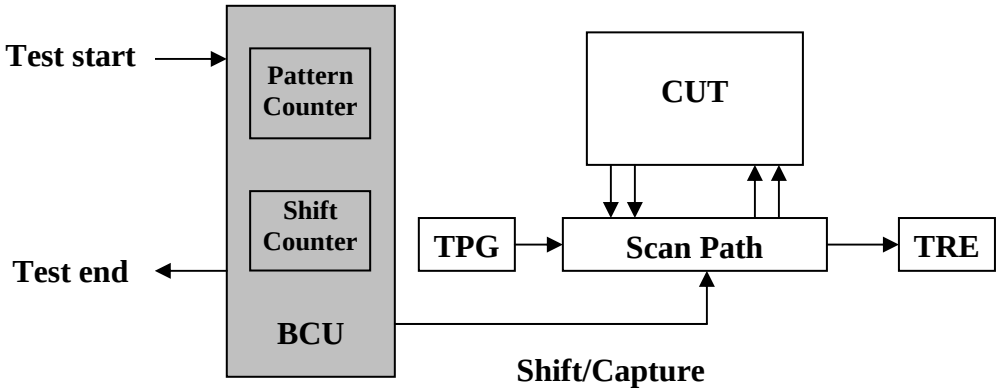


Figure 3.2: Test-per-scan scheme (adapted from [Wun98]).

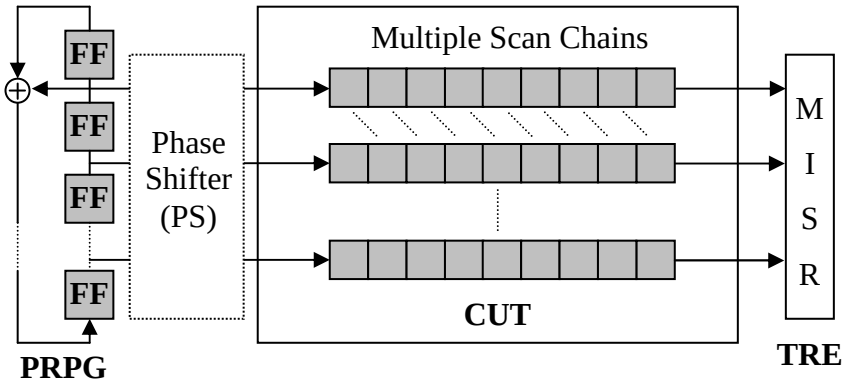


Figure 3.3: STUMPS architecture for parallel-serial mixed scheme (adapted from [Wun02]).

In Figure 3.3, the test patterns are generated by a *pseudo-random pattern generator* (PRPG) and the responses are compacted by a *multiple input shift register* (MISR). Both the PRPG and the MISR are typically implemented as *linear feedback shift registers* (LFSRs) (Section 3.3.1). Such a scheme is called *Self-Test Using MISR and Parallel Shift register sequence generator* (STUMPS) [Bar82].

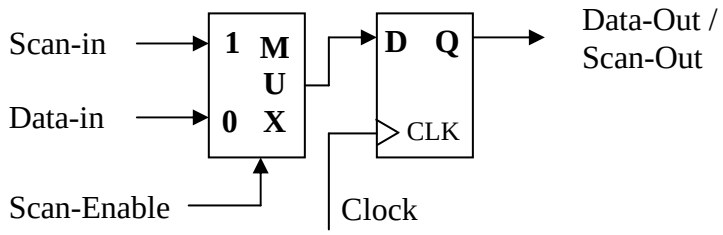
The basic design with multiple scan chains suffers from highly correlated patterns (Section 3.3.1). To solve this problem, XOR-trees (phase shifters (PS)) may be inserted between the LFSR and the scan chains inputs (Figure 3.3) [Bar90][Raj98]. This logic transforms the LFSR outputs into several uncorrelated signals. In order to reduce test time, power consumption and storage requirement, other scan structures like *scan forest* [Xia03] or *Illinois scan* [Hsu01] may be used.

There are several approaches to transform the storage elements of the CUT into scan elements. For example, edge-triggered D-type flip-flops can be transformed into so-called scan flip-flops by adding a multiplexer (Figure 3.4 (a)) in front of them. A *scan-enable* signal is used to switch between shift and capture modes and the same clock signal can be used for both modes [Abr90].

An example of level-triggered storage element transformed into scan element is shown in Figure 3.4 (b). Here, the switching between shift and capture modes is made with the help of two clock signals that control the first of the two latches.

Test-per-scan schemes have several advantages: (a) high fault/defect coverage; (b) reduced test data size (compared to sequential test patterns); (c) relatively low test generation time; (d) reduced test costs (no special requirement for costly ATEs for functional testing); (e) low impact on the system behavior, as only scan paths are included into the mission logic and (f) separation of the pattern generator from the CUT, so that it can be synthesized at a later step of the design flow.

(a) Edge – Triggered Scan Element (Scan Flip-Flop)



(b) Level Sensitive Scan Element (Shift Register Latch)

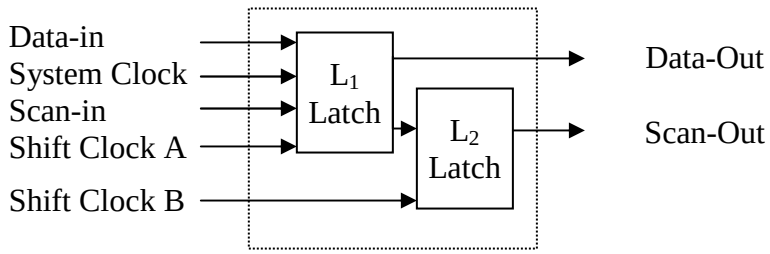


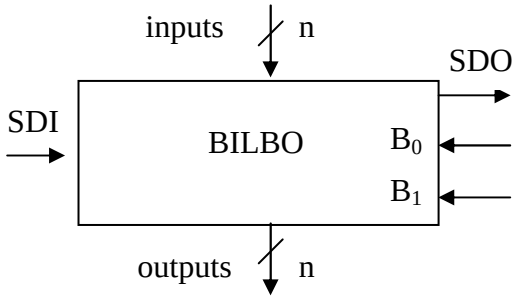
Figure 3.4: Storage cells for scan design (adapted from [Wun02]).

The drawbacks of test-per-scan schemes are: (i) long test application time required by the scan mode; (ii) functionally untestable faults can be activated[4]; (iii) reduced testability for faults whose detection necessitates pairs of test patterns and (iv) reduced system performance if scan elements are introduced into the critical paths. If partial scan paths [Jou95][Tri80] are used, such problems can be reduced and more test patterns may be applied within the same test time.

3.2 Test-per-clock Schemes

In a test-per-clock scheme [Koe79][Kra89][Str94][Wan86], a test pattern is applied to the CUT every clock cycle. This scheme is best suited for register-based design. This kind of scheme employs a specific BIST architecture using the *built-in logic block observer* (BILBO) [Koe79], which is a more sophisticated register that can function as a normal state register, scan register, PRPG or MISR All functionality of the BILBO depends on the mode input signals B_0 and B_1. Signal B_0 controls all the registers to switch between the global and local modes (Figure 3.5). The global mode covers the functional and scan modes. In the local mode the registers may act as pattern generators or response evaluators. In order to select each of these sub-modes associated with the global or local mode, the signal B_1 is used. In contrast to signal B_0, which is unique for all registers, the signal B_1 depends upon the addressed register.

In Figure 3.6, it can be seen how to facilitate testing by changing the functionality of the BILBO registers. Initially, the registers R_1 and R_2 are initialized in scan mode. Then register R_1 is set to a PRPG mode for the combinational logic C_1 and the test responses are observed by register R_2 that functions in response evaluation mode as MISR. The combinational logic C_2 is tested after the test outcome contained in R_2 is shifted out and the functionalities of R_1 and R_2 are interchanged. In the end, the new test outcome contained in R_1 has to be shifted out.



B_0 B_1	Mode	
0 0	Shift	global
0 1	System	
1 0	Pattern generation	local
1 1	Response evaluation	

Figure 3.5: Control signals of a BILBO (adapted from [Wun98]).

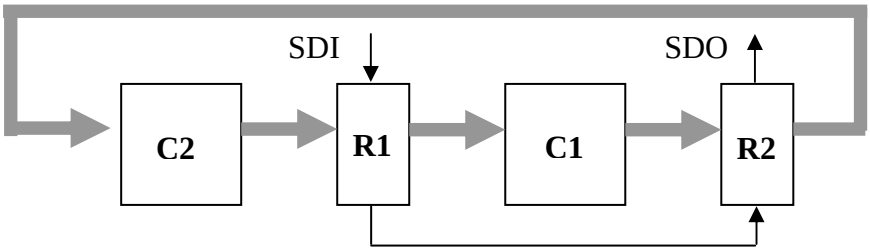


Figure 3.6: Test-per-clock scheme (adapted from [Wun02]).

[4] The test of faults (paths) that cannot be functionally activated may result in a yield loss.

Compared to test-per-scan schemes, the test-per-clock schemes have both advantages and disadvantages. The advantages of test-per-clock schemes are: (a) shorter test times and better support for two-pattern testing [Coc98], as a new pattern can be applied in each clock cycle; and (b) better support for at-speed testing, as no pattern shifting is required, which generally is done at a lower speed.

The disadvantages of the test-per-clock schemes may be the following: (i) larger hardware overhead and (ii) stronger impact on the system behaviour and design flow. The overhead can also be affected by the increased complexity in the test-per-clock schedule that requires the synthesis of a rather complex BCU. One reason for these disadvantages is that additional test registers have to be included, due to the fact that normal BILBO registers cannot work as TPG and TRE simultaneously. In [Wan86], a special type of BILBO register, also called *concurrent BILBO*, has been introduced, which is able to perform signature analysis and pattern generation concurrently.

3.3 Test Pattern Generation

Test pattern generation for both *test-per-scan* and *test-per-clock* BIST schemes can be classified into the following groups: pseudo-random, weighted, exhaustive, pseudo-exhaustive, deterministic and mixed-mode schemes.

3.3.1 Pseudo-Random Pattern Generation

Pseudo-random pattern testing is an attractive approach for BIST. Possible choices for pseudo-random pattern generators (PRPGs) are one-dimensional linear hybrid cellular automata (LHCAs), linear feedback shift registers (LFSRs) or different accumulator based structures [Gup96][Wu98]. As processor kernels or programmable units are integrated into SOCs, they can also be used for pattern generation [Hel96].

An LHCA [Cat96][Kha87] is a collection of memory cells x_1, x_2,…, x_{j-1}, x_j, $x_{j+1, \ldots}$ connected in such a way that each cell is restricted to local neighborhood interactions. The next state of each cell is determined based on the states of the cells with each the considered cell interacts. For example, if cell j can communicate only with the neighbor cells, j-1 and j+1, one of the following two rules can be employed: $x_j(t+1) = x_{j-1}(t) \oplus x_{j+1}(t)$ or $x_j(t+1) = x_{j-1}(t) \oplus x_j(t) \oplus x_{j+1}(t)$, where $x_j(t)$ represents the state of cell j at time t.

An LFSR is a Moore finite state machine that consists of interconnected memory elements, also referred to as *stages* or *cells*, and linear logic elements such as exclusive-OR (XOR) or exclusive-NOR (XNOR) gates. Several LFSR configurations are used in a variety of *design for testability* (DFT) schemes. In this sub-section, the basic theory and the operation of two basic LFSR types will be briefly discussed.

The canonical form of an LFSR, also called standard LFSR (SLFSR), is sketched in Figure 3.7. Here, h_i is a binary constant. h_i=1 implies that a connection exists, while h_i=0 implies that no connection exists. In the latter case the corresponding XOR gate can be replaced by a direct connection between the gate input and its output.

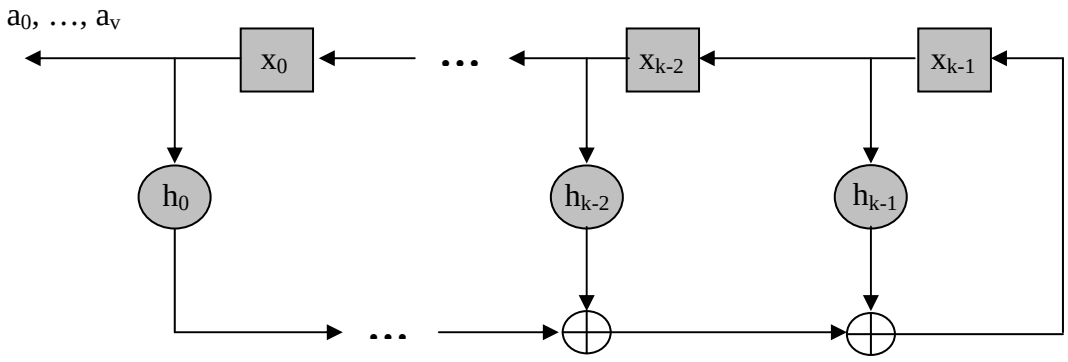


Figure 3.7: Standard linear feedback shift register (adapted from [Wun98]).

The behavior of an SLFSR is completely determined by the feedback coefficients h_0, ..., h_{k-1}, which define a polynomial $h(x) = x^k + \sum_{j=0}^{k-1} h_j x^j$ called the *characteristic* or *feedback* polynomial. The output sequence a_v of the SLFSR has to satisfy the following recurrence equation:

$$\text{for } v = 0, ..., k\text{-}1: \quad a_v = x_v \text{ and for } v \geq k: \; a_v = \sum_{j=0}^{k-1} a_{v-k+j} h_j$$

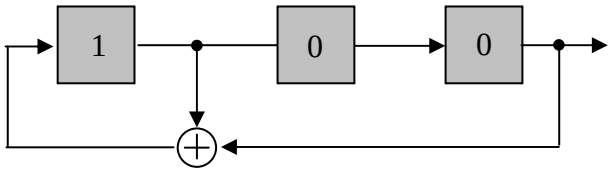
The state transition matrix H of the SLFSR is shown in Figure 3.8. Given the state transition matrix H, the characteristic polynomial h(x) is equal to det(H+x·ID). If the initial state is an all-0 state, the subsequent states can only be all-0 states. Consequently, the all-0 state will lock up the SLFSR in a degenerated sequence. If the initial state of an SLFSR is different from the all-0 state, the SLFSR will produce a non-degenerated sequence of states/outputs, which is periodic and its period cannot be greater than 2^k-1 (k is the length of the SLFSR).

The period of the non-degenerated output sequence $(a_v)_{v\geq 0}$ produced by an SLFSR of length k is the smallest integer p ($\leq 2^k$-1) such that the polynomial $(1-x^p)$ is divided by the reciprocal of the characteristic polynomial [Gol82]: $x^k \cdot h\left(\frac{1}{x}\right) = 1 + \sum_{j=1}^{k} h_{k-j} x^j$.

For any length k of an SLFSR, feedback polynomials exist which can generate state/output sequences of maximum length (2^k-1). The characteristic polynomial corresponding to such an SLFSR is referred to as primitive polynomial. A primitive polynomial is irreducible [Gol82], which means that it cannot be factored out. Figure 3.9 illustrates an example of a maximum length SLFSR.

$$\begin{pmatrix} x_0(t+1) \\ x_1(t+1) \\ . \\ . \\ x_{k-2}(t+1) \\ x_{k-1}(t+1) \end{pmatrix} = H \cdot \begin{pmatrix} x_0(t) \\ x_1(t) \\ . \\ . \\ x_{k-2}(t) \\ x_{k-1}(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 0 & 1 \\ h_0 & h_1 & h_2 & \dots & h_{k-2} & h_{k-1} \end{pmatrix} \cdot \begin{pmatrix} x_0(t) \\ x_1(t) \\ . \\ . \\ x_{k-2}(t) \\ x_{k-1}(t) \end{pmatrix}$$

Figure 3.8: State transition matrix of an SLFSR (adapted from [Wun98]).



	x_0	x_1	x_2
s_0	1	0	0
s_1	1	1	0
s_2	1	1	1
s_3	0	1	1
s_4	1	0	1
s_5	0	1	0
s_6	0	0	1
$s_7 = s_0$	1	0	0

Figure 3.9: Example of maximum length SLFSR (adapted from [Wun02]).

The output sequence $(a_v)_{v\geq 0}$ generated by an SLFSR with a primitive polynomial has several random properties [Gol82] and it is called a pseudo-random sequence. Test patterns that are pseudo-randomly generated are used in many BIST schemes. The main limitation of this test pattern generation approach is that, in most of the cases, insufficient fault coverage is achieved due to linear dependencies. This happens when the initial state of an LFSR, that has to generate a particular output sequence $(a_v)_{v\geq 0}$, is defined by the solution of an unsolvable systems of equations.

In the example of Figure 3.10, each specified bit a_i in the test sequence corresponds to a linear equation in the variables describing the initial state of the LFSR: x_0, x_1, x_2. The detection of the stuck-at 0 fault at the output O_2 requires $a_2 \oplus a_4 \oplus a_5 = 1$, for which no solution exits.

If s is the number of specified bits in the output sequence of a k-bit SLFSR, then the probability *P* that the system of equations determined by the s entries is linearly dependent is given by the following expression [Che88]:

$$P = 1 - \prod_{i=0}^{s-1} \frac{2^k - 2^i}{2^k - i - 1}$$

For example, the selection of 20 entries from the output sequence produced by a 32-bit SLFSR leads to a probability $P = 0.000244$ that these 20 bits are dependent and cannot be set randomly.

The linear dependency problem is enhanced in the case of designs with multiple scan chains. As already mentioned in Section 3.1, this problem can be solved by inserting XOR-trees (phase shifters) [Bar90][Raj98] between the LFSR and the scan chains inputs (Figure 3.3). Besides reordering the memory elements in the scan paths, another way to improve the encoding efficiency of an LFSR or of any other linear test pattern generator or decompressor is to insert inversion logic (invertors or XOR-gates) between the scan elements [Bal04][Lai04].

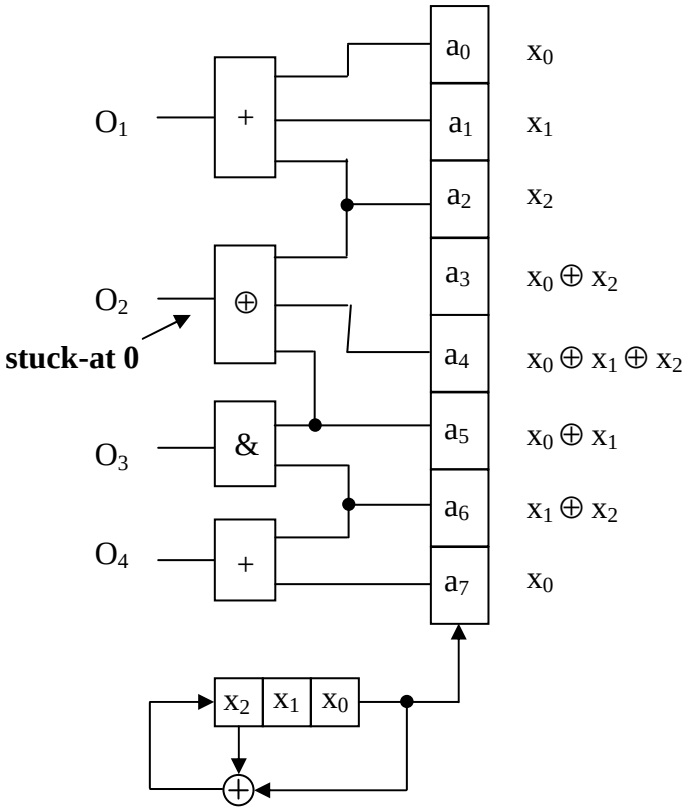


Figure 3.10: LFSR-based testing (adapted from [Wun98]).

An alternative way to implement an LFSR is the so-called modular linear feedback shift register (MLFSR) as illustrated in Figure 3.11. The XOR-gates are connected between the stages of the MLFSR. MLFSRs are faster than SLFSRs as the maximum delay is one XOR gate. Moreover, the difference between successive internal states is enhanced in the case of an MLFSR, which is especially useful for BIST applications.

It can be easily proven that for each MLFSR (SLFSR), an SLFSR (MLFSR) with an equivalent state transition matrix can be found which is expressed by the relations: $H_{SLSFR} = T \cdot H_{MLSFR} \cdot T^{-1}$ and $H_{MLSFR} = T^{-1} \cdot H_{SLSFR} \cdot T$ (Figure 3.12). Hence, all results derived for SLFSRs also hold for MLFSRs.

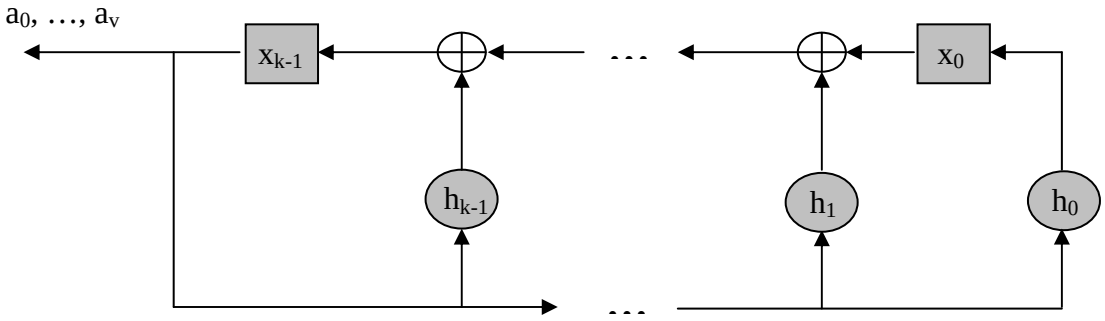


Figure 3.11: Modular linear feedback shift register (adapted from [Wun98]).

$$H_{SLFSR} = T \cdot H_{MLFSR} \cdot T^{-1}, where: \quad H_{SLFSR} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ . & . & . & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & . & \dots & 0 & 1 \\ h_0 & h_1 & h_2 & \dots & h_{k-2} & h_{k-1} \end{pmatrix}$$

$$H_{MLFSR} = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 & h_0 \\ 1 & 0 & . & \dots & 0 & h_1 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & . & \dots & 0 & h_{k-2} \\ 0 & 0 & . & \dots & 1 & h_{k-1} \end{pmatrix}, \quad T = \begin{pmatrix} h_1 & h_2 & h_3 & \dots & h_{k-2} & h_{k-1} & 1 \\ h_2 & h_3 & h_4 & \dots & h_{k-1} & 1 & 0 \\ . & . & . & \dots & . & . & . \\ . & h_{k-1} & 1 & \dots & . & . & . \\ h_{k-1} & 1 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & . & \dots & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.12: Equivalence between the transition matrices of MLFSRs and SLFSRs (adapted from [Hua03]).

3.3.2 Weighted-Random Pattern Testing

Although LFSRs, LHCAs or other linear TPGs can generate a large set of pseudo-random test patterns with very simple hardware, this seldom provides sufficient fault coverage for a CUT. A way to address this problem is to use weighted-random pattern testing techniques.

The TPG used in weighted-random pattern testing is composed of an LFSR and additional combinational logic to modify the probability of ones and zeros in the output sequence. This *weighting* circuitry is used to bias the pseudo-random patterns towards those that detect random pattern resistant faults, such that the fault coverage is increased and the test length can be reduced.

Several techniques have been proposed for computing weight sets [Wun85][Bar87]. In [Wun90] it has been shown that for most circuits, multiple weight sets are required to achieve sufficient fault coverage. For this reason, the weight sets have to be stored on-chip and additional control logic is needed to switch between them during the test time. This increases the BIST overhead a lot.

Extensions of the weighted-random pattern testing are presented in [Tsa00][Lai04], among others.

3.3.3 Exhaustive and Pseudo-Exhaustive Testing

Exhaustive testing applies all possible 2^n test patterns to an n-input combinational circuit [Mcc81], so that a high quality test can be obtained and no particular fault model is used. The test pattern generator can be a binary counter or an LFSR with a primitive feedback polynomial, in which the all-zero pattern may be generated by a reset signal. As the number of test patterns increases exponentially with the number of the circuit inputs, this approach is usually not feasible for circuits with a large number of inputs (n>30).

Pseudo-exhaustive testing relies on the partition of the CUT into output cones which are tested exhaustively [Mcc81][Hel90][Abr90]. As compared to exhaustive testing, far fewer test patterns are required. Nevertheless, the feasibility of pseudo-exhaustive testing depends on the size of the largest output cone.

3.3.4 Deterministic Testing

Deterministic testing applies a pre-computed set of test *cubes* (test patterns with unspecified bits) to the CUT. Thus, any coverage of the testable faults can be achieved. The patterns may be stored on-chip, e.g. using a ROM, or off-chip in which case they have to be loaded from an ATE. In both approaches the data volume to be stored tends to be extremely large.

In the case of the ATE-based approach this may also have a strong impact on the required bandwidth. In order to reduce the storage and bandwidth requirements, special algorithms for generating compact test sets can be used [Cha01][Gon02] [Kaj95][Red92][Tro91][Wue04]. Similar approaches can also be used with (ROM-based) BIST schemes to reduce the storage requirements. Such methods are often called *store and generate* [Agr81].

An intensively investigated *store and generate* technique uses LFSR-reseeding. It is based on storing pre-computed LFSR seeds that can be used to generate deterministic test cubes [Koe91]. Reseeding-based encoding provides a higher compression ratio than any other entropy-based compression method [Tou04]. As seeds are smaller than the test patterns themselves, they require less ROM storage. A small LFSR with a single feedback polynomial may not always have a seed that will generate all the required deterministic test cubes. Multiple-polynomial LFSR schemes [Hel92][Hel95] can fix this problem. The LFSR can operate corresponding to a limited number of different feedback polynomials and produce all the deterministic cubes. Both polynomial and seed identifiers need to be stored.

A different class of reseeding techniques is based on special counters that generate a deterministic set of test cubes. *Twisted-ring counter* [Cha00] and *folding counter* [Lia02] are approaches which embed deterministic cubes into counter sequences. They can efficiently reduce test data storage with full fault coverage, but the approaches are not compatible with standard scan design.

More efficient compression and decompression methods are those in which a small amount of external test data is continuously fed into the chip [Koe91][Raj02] [Wue04]. As long as these methods are based on the use of an external ATE and not

on an internal memory, they are no longer BIST methods and lose some specific benefits of BIST like in-field and on-line testing.

3.3.5 Mixed-Mode Testing

Mixed-mode approaches can achieve more efficient test data compression and hardware implementation than pure deterministic test schemes. Mixed-mode testing combines pseudo-random testing with various deterministic testing schemes so that the test storage requirements can be significantly reduced and high levels of fault coverage can be obtained within a reasonable test application time.

Usually in mixed-mode approaches, the pseudo-random patterns produced by LFSRs are used to test easy-to-detect faults. To increase the number of detected faults, test points can be inserted into the CUT [Geu00][Hay74][Sei91][Vra02]. While the area increase due to the test points may be tolerable, they may also introduce additional delays, which could require a complete resynthesis and a new timing verification [Vra04]. For the remaining faults, deterministic test patterns can be generated by an automatic test pattern generator (ATPG) and stored in a ROM.

In other mixed-mode approaches, often called *test set embedding* schemes, deterministic test patterns are embedded in pseudo-random sequences with the help of some additional combinational logic [Tou96][Wun96][Ghe04]. In the bit-flipping approach, the output sequence of an LFSR is inverted at a few bit positions in order to increase fault coverage [Wun96][Ghe04], while the bit-fixing approach applies constant values [Tou96].

The so-called *Star Test* approach introduced in [Tsa97][Tsa00] uses deterministic test patterns which are surrounded at a limited Hamming distance by clusters of child patterns. Based on the use of parent patterns, the *Star Test* approach can be considered a deterministic method. Due to the way in which the clusters of child patterns are produced, this scheme can also be classified as a generalized weighted-random pattern testing.

Processor kernels or programmable units integrated into the system containing the CUT may also be used to emulate deterministic or mixed-mode schemes [Hel96].

3.4 Test Response Evaluation

Besides test pattern generation, BIST architectures should also be able to compress/evaluate test responses. As the number of test patterns applied to the CUT is usually very large, it is infeasible to store all the expected values on-chip and compare them with the response values. It is much cheaper in terms of storage requirement and compacting circuitry to compress the test responses to short sequences, called *signatures*, which are delivered for analysis at the end of the test session [Abr90].

A signature is obtained as the final state of a finite state machine whose inputs are fed with test responses. This type of compression which addresses the length of the test response sequence is also known as *time compression*. Examples of time compressors are accumulator, LFSR- and counter-based compactors [Abr90][Raj93].

The other type of test response compaction, called *space compression*, is used to transform n test outputs into $m<n$ signals, which may be connected to the primary outputs of the chip or, eventually, to the inputs of a time compressor. Linear space compactors are built with XOR or XNOR gates [Mit04]. Consequently, they may mask out bits carrying the information about the CUT errors. For example, any combination of an odd number of errors on the inputs of a XOR tree propagates to its output, but a combination of an even number of errors remains undetected.

A reduced number of test outputs helps to reduce the ATE storage and bandwidth requirement. In the case of a BIST scheme, the space compression can be also used to reduce the size of the time compressor by limiting the number of its parallel inputs. Limitations of space compression may be the loss of information and fault coverage, if the CUT output includes joint cones [Mit04].

This sub-section considers the LFSR-based time compression and the related signature analysis. An LFSR has the property that it divides the input data (in this case, the test responses) by the characteristic polynomial. The signature is obtained as final remainder of such successive divisions. Instead of comparing a large set of test outputs, only the signature defined as the final state of the LFSR obtained at the end of the testing needs to be compared.

MLFSR and SLFSR-based time compressors are shown in Figure 3.13. Their input signals come from the outputs of the scan paths. The output stream is not observed, and only the final state of the LFSR is used.

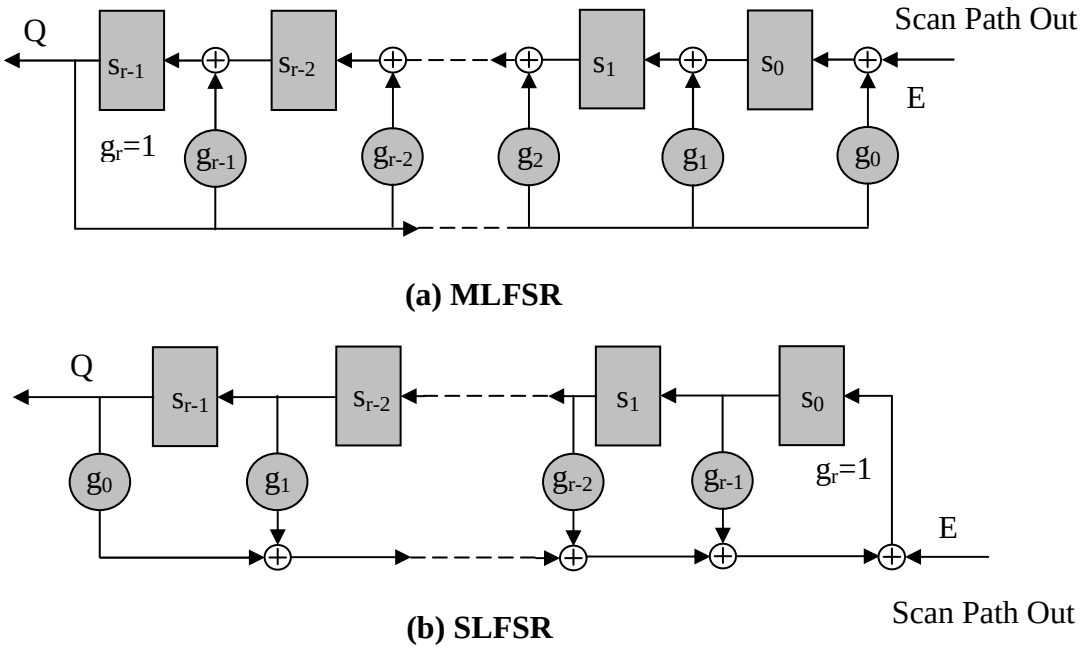


Figure 3.13: LFSR-based time compressors (adapted from [Wun98]).

Figure 3.14 shows an example of the polynomial division performed by an MLFSR-based time compressor. The operation of the time compressor is defined by its feedback polynomial $g(x) = g_r x^r + g_{r-1} x^{r-1} + ... + g_0 = x^4 + x^2 + x + 1$, the input sequence $e(x) = e_n x^n + e_{n-1} x^{n-1} + ... + e_0 = x^7 + x^3 + x^2 + x = 10001110$, the output sequence $q(x) = q_n x^n + q_{n-1} x^{n-1} + ... + q_0 = x^3 + x^1 + 1$ and the remainder polynomial $s(x) = s_{r-1} x^r + ... + s_0 = x^3 + x^2 + x^1 + 1$. Among these polynomials the following relation exists:

$$\frac{e(x)}{g(x)} = q(x) + \frac{s(x)}{g(x)}$$

The signature S = 1111 can be derived from the expression of the remainder polynomial s(x).

The LFSR-based time compressor discussed above had only one single input. It is straightforward to extend the number of inputs of an LFSR-based time compressor and to obtain a so-called *multi-input shift register* (MISR), which can be used for parallel signature analysis (Figure 3.15).

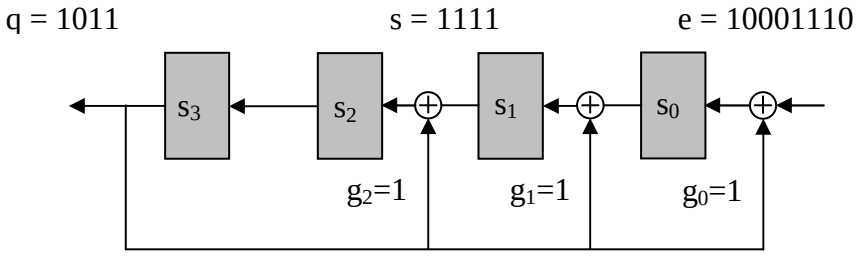
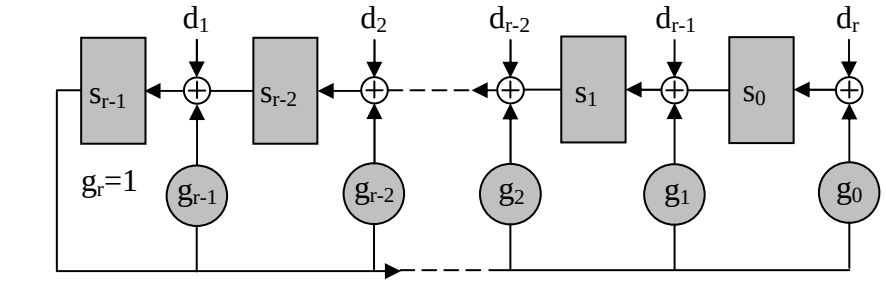
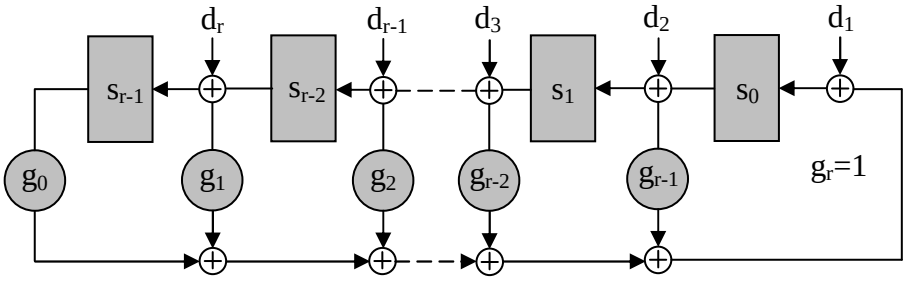


Figure 3.14: LFSR performing division (adapted from [Hua03]).



(a) MLFSR



(b) SLFSR

Figure 3.15: Parallel signature analysis (adapted from [Wun02]).

An ideal compaction algorithm has the following features: (a) it should be easy to implement it as a part of the on-chip DFT circuitry; (b) it should not be a limiting factor with respect to test time; (c) it should provide a logarithmic compression of the test data; and (d) it should not lose information concerning the tested faults. However, there is no known compaction algorithm that satisfies all the above criteria. In particular, it is difficult to ensure that the compressed output obtained from a faulty circuit is not the same as the output of the fault-free circuit. This phenomenon is often referred to as *error masking* or *aliasing* and is measured in terms of the likelihood of its occurrence.

Aliasing occurs because many compaction operations have an inherent filtering effect. Methods to design test response compactors with minimum aliasing probability are available in [Dac90][Dam89][Str90][Zor90], among others. They use primitive feedback polynomials and assume that errors occur randomly.

The probability of aliasing for MISR-based compression has been theoretically proven to be 2^{-k}, where k is the signature length. We can note that the result is independent of the size and complexity of the CUT and a long signature can provide low aliasing.

The use of accumulator based structures for test response compaction leads to aliasing probabilities comparable to the MISR-based methodology [Raj93]. In the counter-based time compression approach the number of ones or the number of 0-1 and 1-0 transitions in the test response sequences are counted. Depending upon the situation, either ones counting, transition counting or MISR-based time compression is a better solution [Abr90].

Due to its low aliasing, high speed, small hardware overhead and better scalability (for improving the aliasing probability only the characteristic polynomial or the length of the register needs to be changed), the MISR-based solution is chosen for BIST.

Chapter 4

Representation, Manipulation and Implementation of Boolean Functions

This chapter discusses two basic approaches that are used for the representation and manipulation of Boolean functions. These two approaches rely on the cube- and Binary Decision Diagram (BDD)-based representations, respectively. Logic implementation styles using the two representations are also analyzed.

Here, a distinction has to be made between completely and incompletely specified Boolean functions. In the sequel, lowercase letters will be used to indicate completely specified functions (e.g. *f, g*), while uppercase letters will be used to denote incompletely specified functions (e.g. *F, G*).

Definition 4.1: Given an incompletely specified function $F:\{0,1\}^n \rightarrow\{0,1,X\}$ (the symbol 'X' indicates a *don't care*), its definition space is partitioned into 3 sets: *ON*-set, *OFF*-set and *DC*-set containing all the input assignments mapped to '1', '0' and 'X', respectively. Depending whether the DC-set is empty or not, the functions are classified into *completely specified* (DC-set = $\varnothing$) and *incompletely specified* (DC-set $\neq \varnothing$).

In order to define an incompletely specified function, at least 2 of the 3 sets above in their true or negated form should be specified. All over this work, the ON-set and the OFF-set are chosen to represent incompletely specified functions. Consequently, an incompletely specified function $F:\{0,1\}^n \rightarrow\{0,1,X\}$ will be represented by 2 completely specified functions f_{on} and f_{off} that have the following properties:

- $f_{on}:\{0,1\}^n\rightarrow\{0,1\}$ defines the input assignments mapped by F to '1': f_{on}(ON-set) = 1, f_{on}(OFF-set) = f_{on}(DC-set) = 0.
- $f_{off}:\{0,1\}^n\rightarrow\{0,1\}$ defines the input assignments mapped by F to '0': f_{off}(OFF-set) = 1, f_{off}(ON-set) = f_{off}(DC-set) = 0.

In the sequel, $F(f_{on}, f_{off})$ will denote an incompletely specified function $F:\{0,1\}^n \rightarrow\{0,1,X\}$, represented by the functions f_{on}, and $f_{off}:\{0,1\}^N\rightarrow\{0,1\}$.

Definition 4.2: A completely specified function $Cov(F)$ is called a *cover* of $F(f_{on}, f_{off})$ iff the following holds: $f_{on} \cdot Cov(F) = f_{on}$ and $f_{off} \cdot Cov(F) = 0$. Here, $Cov(F)$ will be assimilated to a possible implementation of $F(f_{on}, f_{off})$.

Section 4.1 introduces the cube-based, also called disjunctive two-level, representa-

tion. The generalization of this representation to the multi-level representation and implementation is described in Section 4.2. Section 4.3 presents the representation, manipulation and logic synthesis of Boolean functions based on BDDs.

4.1 Two-level (Cube-based) Representations of Boolean Functions

The definitions below are given for a better understanding of the following discussion.

Definition 4.3: A literal is a variable in its true or negated form. Examples of literals are: a, $\neg a$, b, $\neg b$, c, $\neg c$

Definition 4.4: The cofactor of a Boolean function f by a literal $l \in \{x, \neg x\}$ is a Boolean function, $f|_l$, which is equal to f evaluated at x, if $l = x$, or at $\neg x$, if $l = \neg x$.

Definition 4.5: A product-term, also called cube, is a set of literals and it is used to represent the function obtained by the product of the literals in the set. Examples of cubes are: $a \cdot b \cdot \neg c$, $\neg b \cdot c \cdot \neg d \cdot e$

Using the cube-based representation, a function can be expressed in the sum-of-products form, also called *disjunctive form*. Examples of cube-based representations are:

$$f_1 = a \cdot b \cdot \neg c + \neg b \cdot c \cdot \neg d \cdot e$$

$$f_2 = \neg a \cdot b \neg c + c \cdot d + \neg b \cdot \neg e + a \cdot \neg d$$

Besides the disjunctive form, the two-level representation also has a *conjunctive form*, in which the considered function is represented as a product-of-sums. Examples of *conjunctive forms* are:

$$f_3 = (a + b + \neg c)(\neg b + c + \neg d + e)$$

$$f_4 = (\neg a + b + \neg c)(c + d)(\neg b + \neg e)(a + \neg d)$$

Definition 4.6: Each cube in a sum-of-products (also called cube)-based representation of the function f is also called product-term or *implicant*.

Definition 4.7: Given a Boolean function $f:\{0,1\}^n \rightarrow \{0,1\}$, an implicant of it that contains n literals is called *minterm*. A *minterm* corresponds to a completely specified input assignment mapped by f to '1'.

Two-level representations are especially suitable for a design style based on *programmable logic arrays* (PLAs). This is due to the fact that each product/sum of the two-level representation is implemented as a row/column of the PLA.

All the following considerations can be applied to both the disjunctive and the conjunctive two-level representations, assuming a few modifications. For the sake of simplicity, only the disjunctive two-level (also called cube)-based representations will be considered from now on.

Definition 4.8: An implicant (or cube) *c* of a sum-of-products expression *f* is *prime* if none of the *c*'s literals can be removed such that the function represented by *f* remains unchanged. A sum-of-products is *prime* if it contains only prime implicants.

Definition 4.9: A sum-of-products expression *f* is *irredundant* if the removal of any implicant (or cube) $c \in f$ produces a non-equivalent expression.

The goal of the cube-based implementation is to find disjunctive two-level representations with a (near-)minimal number of product terms and literals. Consequently, for an efficient manipulation and implementation, the used sum-of-products has to be prime and irredundant. Exact minimization techniques for the cube-based logic implementation involve two steps [Bra97]:

- generation of all prime implicants
- extraction of a minimum prime and irredundant cover

Two well known methods for the generation of all prime implicants are based on the covering of Karnaugh-Veitch maps or on the Quine-McCluskey algorithm [Mcc65]. Unfortunately, the number of all prime implicants of a Boolean function can be very high. It can be shown that this number can be as large as $3^n/n$ for a function with *n* inputs [Bra97].

Exact extraction of a minimum prime cover involves the solution of a minimum covering problem that is known to belong to the class of NP-complete problems [Bra97].

Most known heuristics to deal with the minimization of the two-level covers are included in the program ESPRESSO [Bra97], which is especially suitable for the implementation of incompletely specified functions. Unfortunately, many of the employed algorithms have an exponential worst-case complexity [Bra97]. Consequently, only relatively small problems can be efficiently handled with ESPRESSO.

The poor scalability of ESPRESSO is also a consequence of the poor scalability of the cube-based implementation of the Boolean operators. A way to circumvent this problem was given by Minato in [Min97] where a new representation, called Zero-Suppressed Binary Decision Diagram (ZBDD), had been introduced.

4.2 Multi-level Implementations of Boolean Functions

A generalization of the two-level representation is the multi-level representation, which is used to obtain more compact implementations of Boolean functions. As an example, consider the implementation of the Boolean function *f* below, which is given in its minimal cube-based representation:

$$f = a \cdot b \cdot c + a \cdot b \cdot \neg e + a \cdot b \cdot g + c \cdot d + d \cdot \neg e + d \cdot h \qquad (4.1)$$

By using an intermediate variable *p*, the function *f* can be rewritten as follows:

$$f = p \cdot c + p \cdot \neg e + a \cdot b \cdot g + d \cdot h, \quad p = a \cdot b + d$$

By using another two intermediate variables q and r, the function f can be rewritten as follows:

$$f = p \cdot q + r \cdot g + d \cdot h, \quad p = r + d, \quad q = c + \neg e, \quad r = a \cdot b \tag{4.2}$$

Multi-level implementations are especially useful for standard cell design. In such a case the implementation of the expression (4.1) requires 14 2-input logic gates, while the implementation of the expression (4.2) needs only 8 2-input logic gates.

The operation used to simplify the expression (4.1) to (4.2) is called factorization. Factorized forms can be achieved by performing one of the two types of division [Bra87]: the *algebraic-division,* also called *weak-division*, or the *Boolean-division*. The algebraic-division is relatively easier to implement, but the Boolean-division provides better results.

For multi-level logic synthesis based on factorization, the quality of the results greatly depends on the choice of the divisors. Divisor extraction methods for both algebraic-division and Boolean-division are described and successfully used in [Bra87]. A simple and fast divisor extraction method together with a fast algebraic-division approach is presented in [Min97].

A way to represent multi-level forms is to use Boolean networks:

Definition 4.10: A Boolean network is a net-list of connected components, where each individual component may implement an arbitrary Boolean function.

Relevant examples of multi-level synthesis tools which are able to handle *don't cares* are MIS [Bra87], SIS [Sen92] and Minato's multi-level logic synthesizer [Min97]. In MIS and SIS, the DC-based optimization relies on ESPRESSO or simpler variants of it, which can act only on the two-level representation of the functions implemented by each node of the target Boolean network. This DC-based optimization does not necessarily guarantee a reduction of the size of the Boolean network [Bra87].

In Minato's multi-level logic synthesizer, the algorithm of Minato and Morreale is used to generate a prime-irredundant cube cover of the target incompletely specified function [Min97]. The cube cover is transformed into a multi-level circuit with the help of a heuristic for fast algebraic-division.

In both multi-level synthesis approaches, the DC-set is only used for optimizations of two-level representations.

4.3 BDD-based Representations of Boolean Functions

Reduced ordered BDDs (ROBDDs) [Akr78][Lee59] offer an efficient way for manipulating and representing a large variety of Boolean functions [Bec95][Bry86]. Moreover, the internal structure of BDDs provides the basis for logic synthesis solutions that can be considered as a compromise between two- and multi-level logic implementations (Section 4.3.3).

Definition 4.11 A BDD is a rooted, directed, acyclic graph {V, E} with an edge set E and a vertex (node) set *V* containing two types of vertices. A *non-terminal* vertex v has two attributes: an argument index $index(v) \in \{1,\ldots, n\}$, which indexes an input variable, and two children (siblings) $low(v)$, $high(v) \in$ V. A *terminal* (leaf) vertex v has as attribute a value $value(v) \in \{0,1\}$. Each non-terminal node v is connected to its $high(v)$, $low(v)$ children by a *then-edge*, *else-edge* $\in$ E, respectively.

Figure 4.1 shows the BDD-based representation of the parity function *parity(a,b,c)* that operates on the input variables *a*, *b*, and *c*. The function result is 0 if there is an even number of input variables that have the value 1, while the function result is 1 if there is an odd number of input variables that have the value 1. For instance, *parity*(011) = 0 and *parity*(010) = 1. The labels at the edges correspond to the variable value of the parent vertex. The BDD-based representation of the parity function with *n* input variables contains $2n+1$ vertices, while a cube-based representation of the same function would require 2^{n-1} cubes. When evaluating the compaction of the cube- and BDD-based representations, one should look not only at the numbers of cubes and nodes, but one should also notice that the cube size may grow linearly with the number of input variables, while the size of a BDD node stays constant.

This example illustrates that a BDD may be a very compact representation for certain logic functions. A second advantage of BDDs is that the complexity of many logic operations performed by using BDD-based representations scales linearly with the number of input variables [Bry86].

A few special BDD types and their properties are presented in the following.

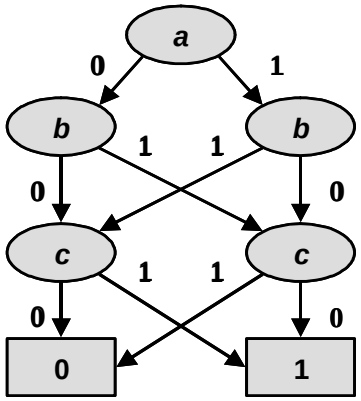


Figure 4.1: BDD representation of the parity function with three input variables (adapted from [Bry86]).

4.3.1 Types of Binary Decision Diagrams

BDDs can be compacted (*reduced*) with the help of two rules: *merging* and *deletion* (*elimination*). *Merging* unifies pairs of BDD nodes with the same index and identical low- and high-children. Such a pair of nodes is called *isomorphic*. *Deletion* removes a node whose children represent the same function and replaces it by one of its children [Bry86]. If each node in a BDD represents a different function, then this BDD is said to be a *reduced* BDD (RBDD). For example, the BDD presented in Figure 4.1 is reduced.

ZBDDs have a different *deletion* rule according to which only those non-terminal nodes are eliminated whose then-edge points to a terminal node with the *value* 0.

Each non-terminal node v of a BDD implements a Shannon decomposition (expansion) of the Boolean function $f(v)$ represented by the sub-graph rooted at v:

$$f(v) = x \cdot f(high(v)) + \neg x \cdot f(low(v))$$

where x is the input variable indexed by $index(v)$. The functions implemented by the children of v are the $f(v)$ cofactors by the input variable x: $f(high(v)) = f(v)|_{x=1}$ and $f(low(v)) = f(v)|_{x=0}$.

If, instead of the Shannon decomposition, each non-terminal node implements the Reed-Muller (Davio) expansion, a new type of decision diagram is obtained: the functional Decision Diagram (FDD) [Keb92][Keb93]. Each non-terminal node v of a FDD may implement a positive or a negative Davio decomposition of the Boolean function $f(v)$ represented by the sub-graph rooted at v:

$$f(v) = f(high(v)) \oplus x \cdot f(low(v)) \text{ (positive Davio)}$$

$$f(v) = f(high(v)) \oplus \neg x \cdot f(low(v)) \text{ (negative Davio)}$$

where $f(low(v)) = f(v)|_{x=0} \oplus f(v)|_{x=1}$, $f(\text{high}(v)) = f(v)|_{x=0}$ (positive Davio) or $f(\text{high}(v)) = f(v)|_{x=1}$(negative Davio).

In order to improve the BDD-based manipulation of Boolean functions, a restriction is introduced that the input variable (node index) order is fixed on all the paths starting from the root node and ending at a terminal node [Bry86]. With respect to this restriction the following types of BDDs can be defined.

Definition 4.12 A *free* BDD (FBDD) is a BDD in which: (a) each node index can appear at most once on a given path from root to a terminal node (for reasons explained in Section 7.2, this restriction will be disabled here) and (b) different paths can have different orderings of the node indices. An *ordered* BDD (OBDD) [Bry86] is a free BDD where the node indices can appear at most once and only in the same order on all the paths from root to a leaf node. For example, the BDD presented in Figure 4.1 is a reduced OBDD (ROBDD).

For a given variable order, ROBDDs provide only one canonical representation of the Boolean functions. ROFDDs may provide 2^n different canonical representations for a given variable order, where n is the number of input variables. The number 2^n is due

to the fact that for each variable, either the positive or the negative Davio decomposition can be used.

In order to improve performance, these BDD types can be combined to obtain hybrid structures with better properties (e.g. better compaction) [Ger96][Dr98].

4.3.2 ROBDD-based Manipulation of Boolean Functions

One of the benefits of the ROBDD-based representation is the efficient manipulation of Boolean functions. The time complexities of the ROBDD-based implementations of the basic logic operations [Bry86][Sie93] are provided in Table 4.1, using the notations given below:

- n is the number of input variables of the considered Boolean functions.
- G denotes the graph of the ROBDD-based representation of the considered Boolean function for a given variable order. Only the *reduce* operator, which transforms an OBDD into a ROBDD, receives an unreduced G as input. |G| represents the node count of the graph G.
- S_f denotes the *satisfying set* of the Boolean function *f*, which is set of completely specified input assignments mapped to '1' by the function *f*.
- $||f||$ represents the cardinality of S_f.

Procedure	Result	Time Complexity
Equivalence check	$f_1 == f_2$ or $f_1 != f_2$	Constant
Negation	$\neg f$	Constant
Reduce	G reduced to canonical form	O(|G|)
Apply	F_1 <operator> f_2	$O(\|G_1\| \cdot \|G_2\|)$
Compose	$f_1\|_{x = f2}$	$O(\|G_1\|^2 \cdot \|G_2\|)$
Cofactor computation	$f\|_{x = b}$	O(|G|)
Satisfy-one	Some element of S_f	O(n)
Satisfy-all	S_f	$O(n \cdot \|\|f\|\|)$
Satisfy-count	$\|\|f\|\|$	O(|G|)

Table 4.1: Time-complexity of basic logic operations performed with ROBDD-based representations [Bry86].

Due to the fact that for a given variable order the ROBDD-based representation is canonical, the ROBDD-based implementations of *equivalence* ($f_1 = f_2$), *tautology* ($f = 1$) and *satisfiability* ($f = 0$)) have constant complexity in BDD-packages where all the nodes are stored in a so-called unique table[5]. The use of *complemented edges* [Cudd] enables to implement the BDD-based *negation* with constant complexity. The ROBDD-based implementation of the *apply* operator (e.g. logic OR or AND operators) can be made very efficient by the use of *hash-tables*[Cudd], as long as the

[5] A hash table in which each BDD-node represents a different function.

size of the ROBDD-based representation does not explode due to the dependence of the *apply* operator on |G|.

The complexity of the cube-based *apply* operator has a lower bound given by the product of the cardinalities of the cube-based representations of the involved operands. The cube-based implementations of the *equivalence* check, *tautology* check, *negation* and *compose* operators are very expensive. The cube-based implementation of these operators can require exponential space and time in terms of the number of input variables more often than in the case of the other operators.

The last four operators in Table 4.1 are specific to the ROBDD-based representation. Two other operators also specific to the ROBDD-based representation are *constrain* [Cou89] and *restrict* [Cou90]. One of these operators (*restrict*) will be used here as a reference for the experimental evaluations.

Definition 4.13 For the functions f and g (with n primary inputs), the function f *constrained* by g, written $f \downarrow g$, is defined by:

$$(f \downarrow g)(r) = \begin{cases} f(r) & \text{if } g(r) = 1 \\ f(s) & \text{if } g(r) = 0 \end{cases}$$

where s is the input assignment such that:

$$g(s) = 1 \text{ and } \sum_{i=1}^{N} |r_i - s_i| \cdot 2^{n-i} \text{ is minimum.}$$

The pseudo-code for *constrain* is given in Figure 4.2.

```
constrain(f, g){
    if (g = 1 or f is constant) return f;
    if (f = g) return 1;
    if (f = ¬g or g = 0) return 0;
    let v be the top variable of {f, g};
    if (g|v = 0) return constrain(f|¬v, g|¬v);
    if (g|¬v = 0) return constrain(f|v, g|v);
    return v · constrain(f|v, g|v) + (¬v) · constrain(f|¬v, g|¬v);
}
```

Figure 4.2: Procedure *constrain* without hash table (adapted from [Cou90]).

In general the ROBDD-based representation of $f \downarrow g$ (*constrain* (f, g)) has fewer nodes than the ROBDD-based representation of f. In most cases of incompletely specified functions $F(f_{on}, f_{off})$, the ROBDD-based representation of the cover $Cov(F) = f_{on} \downarrow (f_{on} + f_{off})$ has fewer nodes than the cover $Cov(F) = f_{on}$. Sometimes the reverse may occur: the ROBDD for $f \downarrow g$ can have more nodes than the ROBDD for f. This frequently occurs when the ROBDD for g depends on many variables that f does not depend on. These variables may be introduced in $f \downarrow g$, causing an undesirable growth of the corresponding ROBDD. Sometimes, this inconvenience can be avoided if the procedure returns $f \downarrow (\exists_x g)$[6], when the top variable x of g has a lower index than the top variable of f. The resulting algorithm implements the so-called *restrict* operator

[6] The operator $\exists_x g = g|_x + g|_{\neg x}$ is called the existential quantification with respect to the variable x.

[Cou90] (Definition 4.14). Normally, the ROBDD for *restrict*(*f*, *g*) is more compact than the ROBDD for *constrain*(*f*, *g*), because *restrict* does not increase the support of the result with respect to the support of *f*.

Definition 4.14 For the functions *f* and *g* (with *n* primary inputs), the function *f restrict* by *g*, written $f \Downarrow g$, is defined by:

$$(f \Downarrow g)(r) = \begin{cases} f(r) & \text{if } g'(r) = 1 \\ f(s) & \text{if } g'(r) = 0 \end{cases}$$

where g'= *fit*(*f, g*) (Figure 4.3) and *s* is the input assignment such that:

$$g'(s) = 1 \text{ and } \sum_{i=1}^{N} |r_i - s_i| \cdot 2^{n-i} \text{ is minimum.}$$

```
fit (f, g){
    if (f = ¬g or g = 0 or f = g or g = 1 or f is constant) return g;
    let v be the top variable of {f, g};
    if (g|v != 0 and g|¬v != 0 and v is not the top variable of f)
      return fit (f, ∃v g); // return fit (f, g|v + g|¬v);
    return v · fit (f|v, g|v) + (¬v) · fit (f|¬v, g|¬v);
}
```

Figure 4.3: Procedure *fit* without hash table.

The pseudo-code for *restrict* is given in Figure 4.4.

```
restrict(f, g){
    if (g = 1 or f is constant) return f;
    if (f = g) return 1;
    if (f = ¬g or g = 0) return 0;
    let v be the top variable of {f, g};
    if (g|v = 0) return restrict(f|¬v, g|¬v);
    if (g|¬v = 0) return restrict(f|v, g|v);
    if (v is not the top variable of f)
      return restrict(f, ∃v g); // return restrict(f, g|v + g|¬v);
    return v · restrict(f|v, g|v) + (¬v) · restrict(f|¬v, g|¬v);
}
```

Figure 4.4: Procedure *restrict* without hash table (adapted from [Cou90]).

4.3.3 BDD-based Implementation of Boolean Functions

The internal structure of a BDD offers the basis for logic synthesis solutions that can be considered as a compromise between two- and multi-level logic implementations. If each non-terminal node of a BDD is substituted by a multiplexer (MUX), a multi-level circuit can easily be generated [Bec92] (Figures 4.5 and 4.6). This also happens if each node of a FDD is implemented with the help of a 2-input AND gate and a 2-input XOR gate [Keb92].

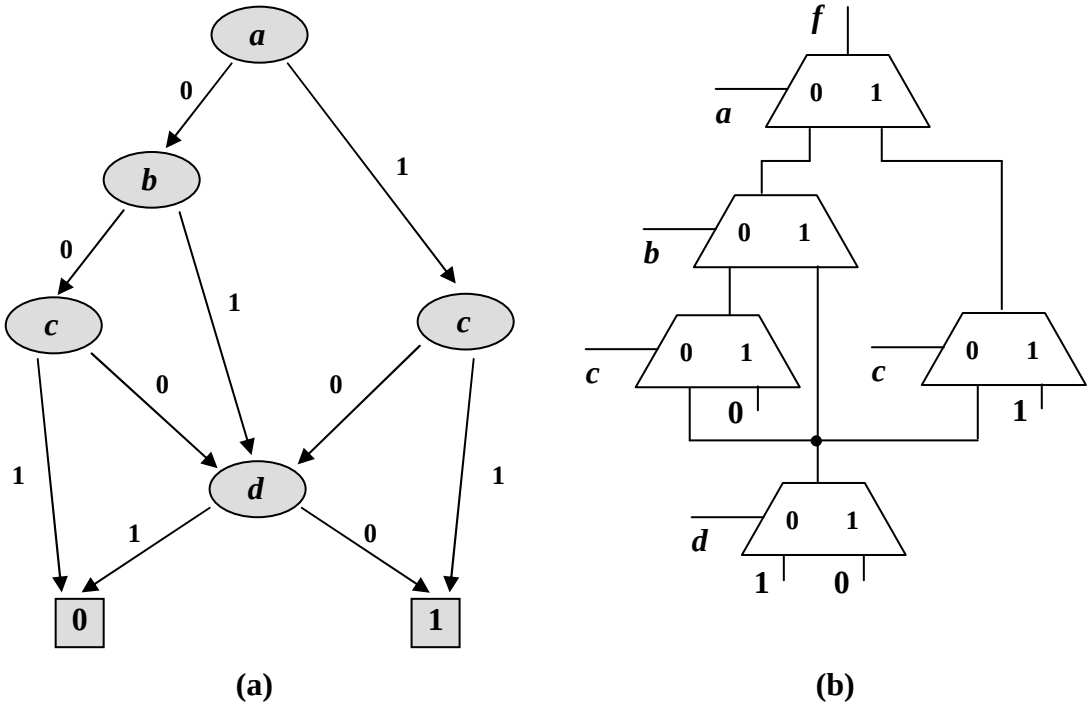


Figure 4.5: (a) BDD for the function f = ¬a· (¬b· (¬c· ¬d) + b· ¬d) + a· (c + ¬d). (b) MUX-based implementation of the function f.

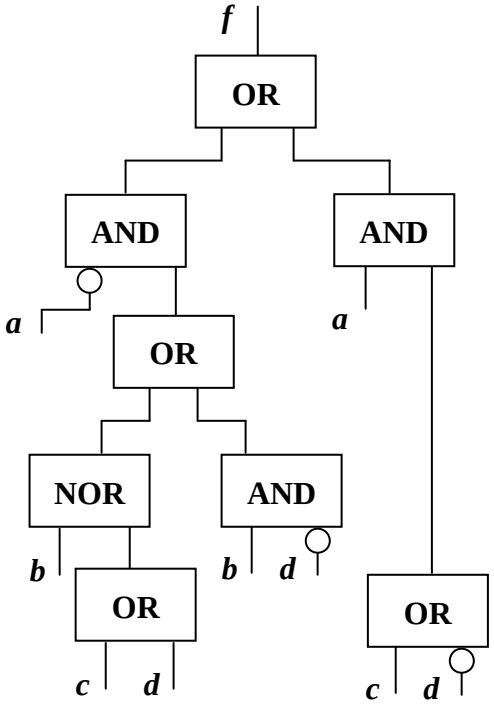


Figure 4.6: Non-redundant implementation of the circuit from Figure 4.5 (b).

A minimal BDD/FDD can offer an efficient implementation or at least a good starting point for a multi-level logic synthesis tool. The limitation of the resulting multi-level representations is that they contain factorized forms in which at most one of the factors is not a literal (Figure 4.6).

The size of a ROBDD depends on the used order of variables. Due to the direct correspondence of a BDD to a combinational logic circuit, saving nodes in a BDD by using good variable orders already pays off. Dynamic reordering heuristics which try to improve a given order of variables can be found in [Fel93][Ish91][Pan94][Rud93], among others.

The situation is more complex if *don't cares* (DC) are involved. In the case of Minato's approach [Min97], the OBDD-based representation of the target circuit and of its DC-set are transformed with the help of Minato-Morreale's algorithm into a prime-irredundant cube cover implicitly represented by a ZBDD. However, mapping BDDs to cubes and applying known algorithms based on two-level representations (e.g. ESPRESSO) may destroy all the benefits of the BDD-based representation.

The problem of minimizing the size of an OBDD-based implementation using the DC-set has been proven to be NP-hard [Sau96]. In [Oli98], an exact OBDD minimization algorithm based on the DC-set is presented. Nevertheless, due to the NP-hardness of the problem, this approach has a limited applicability.

Some of the first heuristics that take advantage of the DC-set for the minimization of the OBDD-based implementations have been introduced by Coudert and Madre based on the operators *constrain* and *restrict* (Section 4.3.2) [Cou89][Cou90]. A cover for an incompletely specified function $F(f_{on}, f_{off})$, can be calculated using the operators *constrain* and *restrict* as shown below:

$$Cov(F) = constrain(f_{on} , f_{on} + f_{off}) \tag{4.3}$$

$$Cov(F) = \neg\, constrain(f_{off} , f_{on} + f_{off}) \tag{4.4}$$

$$Cov(F) = restrict(f_{on} , f_{on} + f_{off}) \tag{4.5}$$

$$Cov(F) = \neg\, restrict(f_{off} , f_{on} + f_{off}) \tag{4.6}$$

where '+' represents the logic disjunction operator.

The ROBDD-based representation of the covers obtained by applying the operators *constrain* and *restrict*, according to the expressions 4.3 – 4.6, is normally more compact than the ROBDD-based representations of f_{on} or f_{off}, as long as the same order of variables is considered. This is due to the fact that f_{on} or f_{off} are expanded towards the DC-set, described by $f_{DC} = \neg(f_{on} + f_{off})$, such that new opportunities are created for the application of the *deletion* rule presented in Section 4.3.1. In this way the ROBDD-based representation of the expanded f_{on} or f_{off} becomes usually more compact.

The ROBDD minimization methods developed in [Cha94][Shi94] exploit the DC-set for sibling matching or, more generally, for matching BDD nodes below cut lines through the BDD, which enables a more aggressive reduction of the BDD size. None of these methods is safe, which would require that the resulting BDD is always smaller than the original one. The compaction algorithm of [Hon97][Hon00] avoids this problem by using a preprocessing step to identify the nodes that can make the minimization *unsafe*. Compared to *restrict* or *constrain* this compaction algorithm

gives better results on the average, but it is considerably slower. Moreover, none of the heuristics analyzed in [Shi94] succeeds to outperform *restrict* by more than a few percents.

All the ROBDD-based minimization methods discussed above consider only ROBDDs with a fixed variable order. In [Sch99], the concept of variable reordering based on symmetries has been extended to incompletely specified functions such that a ROBDD can be minimized by means of *don't care* assignments combined with variable reordering. This method cannot handle large problem instances.

The additional degree of freedom of the FBDD-based representations (Definition 4.12) allows them to have a larger compaction potential than the OBDD-based representations. The same holds in the case of free FDD-based representations [Bec95]. There are functions, like the *hidden weighted bit* function, which require OBDDs and OFDDs of exponential size [Bec95][Bry91], independent of the variable order, while FBDD-based representations of polynomial size are known [Sie95].

In the case of completely specified functions, an exact algorithm for the minimization of FBDD-based representations is described in [Gue99]. Unfortunately, despite sophisticated pruning techniques, such an approach is inherently bound to very small problems (with a maximum of 8 input variables). Heuristics for the minimization of FBDDs have been proposed in [Gue00][Gue99], among others. A complexity analysis of the FBDD minimzation is given in [Sie99].

The first FBDD-based logic synthesis method for incompletely specified functions will be presented in Chapter 7. The new method improves considerably all the synthesis parameters of the Boolean functions which will be introduced in the next chapters, as compared to other synthesis approaches, like SIS or OBDD-based methods.

Chapter 5

Scalable Pattern Mapping for Deterministic Logic BIST

In this chapter, a new algorithm is introduced for mapping deterministic test cubes to a pseudo-random test sequence. The approach is based on BDDs and outperforms the previously published cube-based algorithm [Wun96] by several orders of magnitude. It has been applied to the bit-flipping Deterministic Logic LBIST (DLBIST) architecture which is presented in Section 5.1. In Section 5.2, the pattern mapping problem is formally defined. Sections 5.2 and 5.3 provide a detailed description of a prior cube-based and of the new BDD-based mapping approaches, respectively. Section 5.5 reports the experimental results obtained with a set of industrial, ISCAS-85 and combinational parts of ISCAS-89 benchmark designs. These results prove that significant improvements can be achieved with the help of the BDD-based mapping method. In Section 5.6, the embedded test sequences generated for single stuck-at faults are evaluated with respect to the coverage of non-target defects. Resistive bridging faults are used as a surrogate of non-target defects [Eng05]. This is the first time when the results of such a study are presented. This investigation especially adresses the impact of the test sequence length on the non-target defect coverage and on the hardware overhead. The chapter is concluded in Section 5.7.

5.1 Bit-Flipping DLBIST Architecture

The *bit-flipping* DLBIST scheme is a mixed-mode technique (Section 3.3.5) in which an LFSR and, eventually, a phase shifter (PS) are used to generate the pseudo-random test sequence. If the achieved pseudo-random fault efficiency (Definition 2.2) is not enough, deterministic test patterns are embedded into the pseudo-random sequence with the help of a XOR gate inserted in front of each output of the pseudo-random pattern generator (LFSR + PS). The XOR gates are controlled by a combinational module that implements a so-called *bit-flipping* function (BFF) to selectively flip bits of the pseudo-random test sequence. The pseudo-random pattern generator together with the BFF module and the XOR gates form the pattern generator of this BIST architecture.

From now on, in order to keep the presentation simple, the core under test (CUT) will be assumed to fullfill the following design for test (DFT) constraints, even though these requirements are not mandatory for the implementation of the proposed scheme.

- Test shell around: a flip-flop is associated to each primary input and output.
- Full scan design: all flip-flops (CUT + test shell around) are transformed into scan flip-flop and connected together in one or several (balanced) scan chains.
- BIST readiness: the test responses do not contain unknown bits (Xs).

A scan enable signal is used to switch the scan flip-flops between two modes:

- In *shift* mode (also called *scan* or *test* mode), the scan flip-flops can store only the signal coming from the previous flip-flop in the scan chain. The first scan flip-flop in each scan chain stores the signal coming from the test pattern generator.
- In *functional* mode (also called *capture* or *system* mode), the scan flip-flops can store only signals coming from the CUT. The scan flip-flops in the test shell associated to the primary inputs will store the signals coming from the corresponding primary inputs.

The test application process is managed with the help of a finite state machine, the so-called BIST control unit, which must contain at least a shift counter (SC) and a pattern counter (PC). The SC controls the bit stream corresponding to each test pattern. The PC is used to control the length of the test sequence. In functional mode, the CUT response to the current test pattern is loaded into the scan paths. During the shift mode a new test pattern is shifted into the scan paths, while the CUT response to the previous pattern is shifted out and compressed by a multi-input shift register (MISR). At the end of the test, the MISR contains a signature with the information about the correctness of the CUT.

As shown in Figure 5.1, the state bits of the LFSR, the PC and the SC are connected to the BFF inputs, while the BFF outputs are connected to the XOR-gates at the scan inputs. The operation of the BFF module is controlled by the state bits of the LFSR, the PC and the SC. In the case where a phase shifter (PS) is introduced, it is highly recomandable to use also the output of the PS to control the bit-flipping.

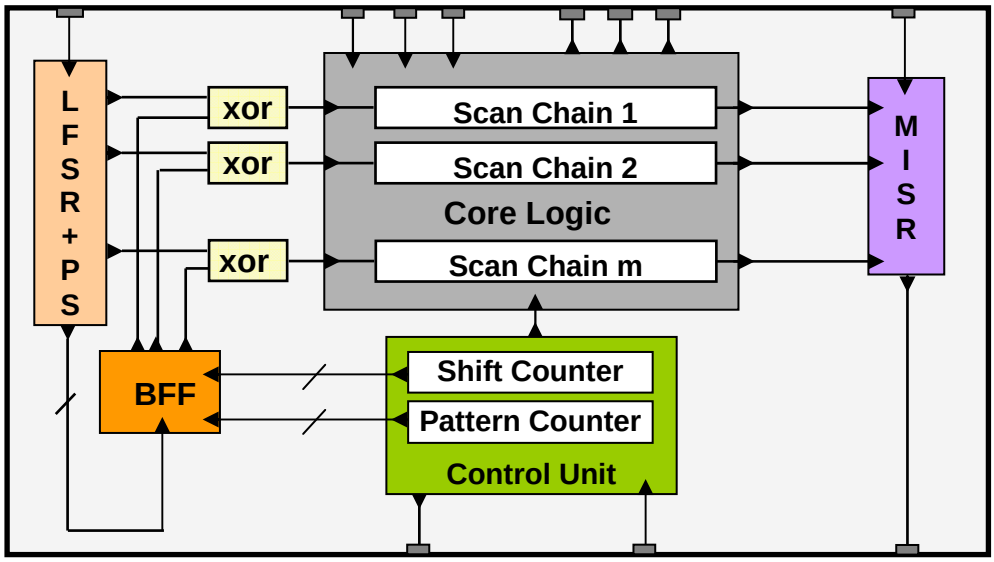


Figure 5.1: Bit-flipping DLBIST architecture.

The LFSR and the SC are updated in every clock cycle, while the PC is updated after applying a new test pattern. In the current implementation, both the SC and the PC are decremented. The all-zero state of the SC indicates that a new test pattern has been shifted in. The new test pattern is applied to the CUT with the help of one clock cycle in functional mode. In order to shift in a new test pattern, the SC is reloaded from a shadow register with a state which corresponds to the length of the longest scan chain of the CUT. The all-zero state of the PC indicates that all the patterns of the test session have been applied and the signature stored in the MISR can be shifted out.

5.2 The Pattern Mapping Problem

Most of the pseudo-random test patterns used in a mixed-mode BIST scheme do not contribute to the fault coverage, since they can only detect faults that are already detected by the previous pseudo-random test patterns. Such *useless* pseudo-random test patterns may therefore be skipped or modified in any arbitrary way. The key idea of the bit-flipping DLBIST scheme is to modify some *useless* pseudo-random patterns into useful deterministic test patterns to improve the fault coverage. In order to do so, an ATPG tool determines test cubes that target those faults not detected by the pseudo-random test sequence. In such a deterministic test cube, only a few bits are actually specified, while most of the bits are *don't care* and hence can be arbitrarily set to '0' or '1'.

In the bit-flipping DLBIST approach, the modification of the pseudo-random patterns is realized by inverting *(flipping)* some of the LFSR outputs, such that deterministic test stimuli are obtained [Wun96]. In the bit-fixing approach, the modification of the pseudo-random patterns is realized by *fixing* some of the LFSR outputs to either '1' or '0', such that deterministic test patterns are produced [Tou96]. In [Wun96], it has been shown that the expected number of bits to be flipped in order to embed a precomputed test cube is significantly smaller than the number of specified bits.

From now on, only pattern modification by means of bit-flipping will be considered. Nevertheless, the considerations presented here can be applied to both the bit-flipping and the bit-fixing approaches, assuming a few modifications.

The bit-flipping is realized by combinational logic implementing a so-called bit-flipping function (BFF). The BFF realizes the mapping of a set of deterministic test cubes to a (larger) set of pseudo-random patterns. Every specified bit (i.e. care bit) in a deterministic test cube either matches the corresponding bit in the associated pseudo-random pattern, in which case bit-flipping should not be performed, or the bit does not match, in which case bit-flipping is required. For all unspecified bits (i.e. *don't care* bits) in a deterministic test cube, the corresponding bits in the associated pseudo-random pattern may be flipped or not. The BFF must provide that (1) all conflicting bits are flipped, (2) all matching bits are not flipped, while (3) the *don't care* bits may be flipped or not. The BFF can be kept quite small by carefully selecting the candidates for each deterministic test cube in the large set of *useless* pseudo-random patterns.

Without any loss of generality, consider a CUT with a single scan chain. Let *S* denote the set of all possible combinations of the states of the LFSR, the PC, the SC and the PS output (if any). The ON-set is the sub-set of *S* that corresponds to the clock cycles in which the LFSR (or PS) output must be flipped. Similarly, the OFF-set is the sub-

set of *S* that corresponds to the clock cycles in which the LFSR (or PS) output must not be flipped. Obviously, the ON-set and OFF-set are disjoint (ON-set $\cap$ OFF-set = $\varnothing$). The *don't care* set (DC-set) contains those states of *S* that corresponds to the clock cycles in which the LFSR (or PS) output may be flipped or not, i.e. the states that are neither in the ON-set nor in the OFF-set (DC-set = S - {ON-set $\cup$ OFF-set}). The DC-set may be exploited to minimize the logic implementation of the BFF.

The ON-set, OFF-set, and DC-set specify an incompletely specified function BFF:$\{0,1\}^n \rightarrow \{0,1,X\}$, where the symbol 'X' indicates a *don't care* and *n* corresponds to the total number of state bits of the LFSR, the PC, the SC and output bits of the PS (if any). For instance, consider the simple example of a DLBIST scheme with a 2-bit LFSR, a 2-bit PC, a 2- bit SC and no PS ($n = 6$). Considering that the symbol '_' stands for the concatenation of the LFSR, the PC and the SC states. Then BFF(01_10_01) = 1 indicates that the pseudo-random bit must be flipped when the LFSR state is 01, the PC state is 10, and the SC state is 01. The state 01_10_01 is therefore part of the ON-set. BFF(01_10_11) = 0 indicates that the pseudo-random bit must not be flipped when the LFSR state is 01, the PC state is 10, and the SC state is 11. The state 01_10_11 is therefore part of the OFF-set. BFF(10_01_01) = 'X' indicates that the pseudo-random bit may be flipped or not when the LFSR state is 10, the PC state is 01, and the SC state is 01. The state 10_01_01 is therefore part of the DC-set.

In a CUT with *m* scan chains, each scan chain *i* ($1 \leq i \leq m$) has its own ON-set$_i$, OFF-set$_i$ and DC-set$_i$. In this case, the BFF is a multi-output function consisting of *m* single-input functions BFF$_i$, one for each scan chain. The size of the BFF implementation can be minimized by sharing logic between the implementations of the BFF$_i$ corresponding to the individual scan chains.

Any pattern mapping approach should take into account the following two constraints:

- Generate a BFF that can be efficiently implemented into logic.
- Require limited run-time and memory resources.

Two fundamentally different pattern mapping approaches are presented in Section 5.3 and Section 5.4. The first approach has been previously introduced in [Wun96], while the second approach is an original contribution of this work. These two pattern mapping solutions are compared on the basis of experimental results in Section 5.5.

5.3 Cube-based Pattern Mapping

The original pattern mapping algorithm presented in [Wun96] and further improved in [Kie97][Kie98] uses the cube-based representation and manipulation of the BFF. The output of this initial approach is a two-level cover of the BFF, optimized using ESPRESSO-like algorithms [Bra97].

Besides the underlying cube-based representation, the other characteristics of the original pattern mapping approach are as follows:

- The mapping process is incremental and coupled with the optimization of the two-level (cube-based) implementation of the resulting BFF.
- The whole pseudo-random test sequence is used to embed deterministic test cubes. Useful pseudo-random test patterns that detect faults not detected by previous pseudo-random patterns are not protected from being corrupted by the bit-flipping logic. They could be protected only by explicitly considering them during the logic optimisation of the BFF, which might be very expensive.

The cube-based mapping approach is explained in the following subsections.

5.3.1 Mapping Cost-Function

Assume V is the set of test patterns generated by the LFSR and the partly generated BFF. Let T be a set of deterministic test cubes to be mapped. For each cube $t \in T$, a test pattern $p_0 \in V$ has to be selected such that t can be *efficiently* mapped to p_0 [Wun96]. Deterministic test cubes with only a few specified bits correspond to faults that are relatively easy to test and might be detected by patterns modified in some later iteration of the algorithm. So, initially those cubes $t \in T$ are selected for mapping, whose number of specified bits is large.

Let the DLBIST hardware states be the concatenated states of the LFSR, the PC, the SC and the output of the PS (if any). Given a deterministic test cube $t \in T$ and a test pattern $p \in V$, let on(t, p)/off(t, p) be the set of DLBIST hardware states which *correspond*[7] to those bits of p that are conflicting/identical to the corresponding specified bits of t.

Let FIX-set denote the set of DLBIST hardware states that correspond to those bits of the modified test sequence that are not allowed to be changed anymore due to previous assignments. In the beginning, the pseudo-random test sequence is not modified yet and FIX-set = $\varnothing$. The cube t can only be mapped to the pattern p if the relation on(t, p) $\cap$ FIX-set = $\varnothing$ holds.

The cost for assigning the cube t to the pattern p is estimated by the increase in the number of product terms required by a 2-level implementation of the BFF. An element c of on(t, p) can be *efficiently expanded* and therefore does not cause any new product term, if there is a cube c_0 in the On-set of BFF (Definition 4.1) such that:

$$(\text{FIX-set} \cup \text{off}(t, p)) \cap (\text{EXPAND}(c, c_0) - \{c, c_0\}) = \varnothing,$$

where the term EXPAND(c, c_0) denotes the smallest Boolean sub-space covering both c and c_0 as used in ESPRESSO [Bra97][Wun96].

The cost of an assignment, cost(t, p), is defined as the number of minterms that cannot be efficiently expanded:

$$\text{cost}(t, p) = \text{cardinality of } \{c \in \text{on}(t, p) \,/\, c \text{ cannot be efficiently expanded}\}$$

[7] A state of the DLBIST hardware is said to *correspond* to a bit of the test sequence, if the DLBIST hardware is in this state when the considered bit of the test sequence is scanned in.

That test pattern $p_0 \in V$, which minimizes the cost function cost(t, p_0), is assigned to the deterministic test cube t.

5.3.2 The Algorithm

The cube-based algorithm used to map deterministic test patterns to a pseudo-random sequence is outlined in Figure 5.2. This mapping is modeled by a BFF which is generated incrementally. The construction process begins with $BFF^0 = 0$ and ends with BFF^R which provides the required fault coverage. In each iteration r, $1 \le r < R$, BFF^{r-1} is enhanced to BFF^r, such that new deterministic test cubes are embedded into the test sequence produced by the LFSR and the BFF^{r-1}, while certain useful test patterns are protected from being corrupted. The individual steps are detailed below:

1. Identify the set F of all the non-redundant faults of the CUT.

   The following steps are repeated until the required fault coverage is achieved. Here, r represents the index of the current iteration and V is the set of test patterns generated by the LFSR and the implementation of BFF^{r-1}, which will be represented by $Cov(BFF^{r-1})$ (Definition 4.2).

   Initialize $r = 1$, ON-set = ∅, OFF-set = ∅ (Definition 4.1) and $Cov(BFF^0) = ∅$.

2. Determine the set F^r_{hard} of non-redundant faults not detected by the current test sequence.

   Compute the set of faults $F_{crit} = \bigcup_{i=0}^{r-1} F^i_{hard}$.

   Given F_{crit}, all the patterns $p \in V$ of the current test sequence are simulated in several permutated orders, until a small sub-set $P = \{p_0, ..., p_k\}$ of *essential* patterns is found which still detects all faults in F_{crit}. In order to guarantee complete fault detection, not all the bits of p_i, $0 \le i \le k$, need to be specified. Don't cares are inserted into each essential pattern as long as the fault coverage is preserved. In this way, P is transformed into a set $P' = \{p_0', ..., p_k'\}$ of patterns that contain as many *don't cares* as possible and can still detect all faults in F_{crit}.

   Let FIX(p_i') be the set of DLBIST hardware states corresponding to the specified bits (also called, *essential* bits) in the pattern $p_i' \in P'$.

   Let FIX-set = $\bigcup_{i=0}^{k} FIX(p_i')$.

3. An ESPRESSO-like REDUCE operator [Bra97] is applied to $Cov(BFF^{r-1})$:

   ON-set = $REDUCE_{\text{FIX-set}}(Cov(BFF^{r-1}))$, OFF-set = FIX-set - ON-set

   REDUCE transforms the prime and irredundant cover $Cov(BFF^{r-1})$ into a new cover ON-set, which is irredundant but usually not prime. This is done by replacing each cube in $Cov(BFF^{r-1})$ by a new and, in general, smaller cube that covers the same number of minterms in FIX-set. After the replacement of each cube, all minterms in FIX-set covered by the replaced cube are removed from FIX-set. Consequently, the result of REDUCE depends on the order in which the cubes in $Cov(BFF^{r-1})$ are replaced.

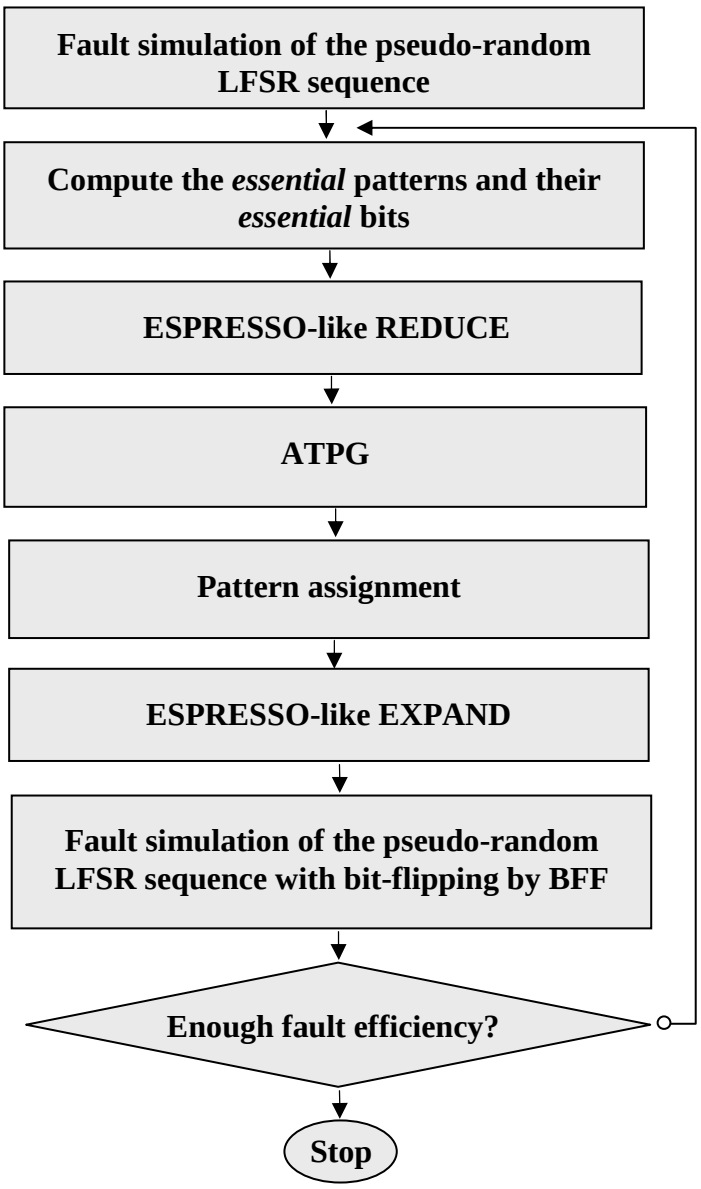


Figure 5.2: Cube-based pattern mapping by means of bit-flipping.

REDUCE allows the cube-based algorithm to move away from locally optimal solutions towards a better one

4. Find a set of deterministic test cubes T with as many *don't cares* as possible that detect as many faults from F_{hard} as possible.
5. For each deterministic cube $t \in T$ find an appropriate pattern $p_0 \in V$ that minimizes cost(t, p_0) (Section 5.3.1) and compute:

$$\text{ON-set} = \text{ON-set} \cup \text{on}(t, p_0), \qquad \text{OFF-set} = \text{OFF-set} \cup \text{off}(t, p_0)$$

6. An ESPRESSO-like EXPAND operator [Bra97] is applied to ON-set:

$$\text{Cov}(\text{BFF}^r) = \text{EXPAND}_{\text{OFF-set}}(\text{ON-set})$$

EXPAND transforms the cover ON-set of BFF^r into a prime and irredundant cover $Cov(BFF^r)$. The goal of EXPAND is to remove as many cubes as possible from ON-set and to remove as many literals as possible from the remaining cubes. The result of EXPAND depends on the order in which the cubes in ON-set are expanded.

7. Simulate the test sequence generated by the LFSR and $Cov(BFF^r)$.
8. Return to step 2 if the required fault coverage has not been achieved, else the iterative mapping process is stopped.

There is a trade-off between the computation time which is smaller for a few loop iterations and the quality of the result which is better if there are only a few assignments per iteration. Usually, the number of assignments per iteration is increased progressively with the iteration number.

Using the DC-set for the optimisation of the bit-flipping logic, modeled by $Cov(BFF^r)$, makes the number of modified pseudo-random patterns larger than the number of embedded deterministic cubes. While this increases the chance of detecting additional previously undetected faults not targeted by the ATPG tool in the previous iterations [Wun96], useful pseudo-random patterns can also be corrupted. Due to this fact, the number of iterations cannot be controlled and the run-time may explode.

5.3.3 An Example

Consider a scan path containing 5 memory elements (flip-flops), which is fed by the output of the LFSR sketched in Figure 5.3. Table 5.1 shows the state sequence of the LFSR. The resulting pseudo-random patterns and the corresponding DLBIST hardware states are listed in Table 5.2. In this particular case, only the LFSR states are considered.

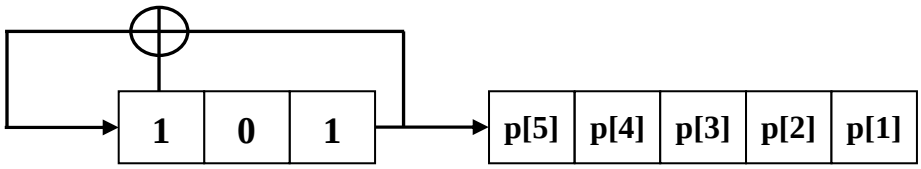


Figure 5.3: LFSR used in the example (adapted from [Wun96]).

s_0	**101**
s_1	**010**
s_2	**001**
s_3	**100**
s_4	**110**
s_5	**111**
s_6	**011**
$s_7=s_0$	**101**
…	…

Table 5.1: States of the LFSR (adapted from [Wun96]).

#	Patterns p[1] … p[5]	States
1	10100	s_0, s_1, s_2, s_3, s_4
2	11101	s_5, s_6, s_0, s_1, s_2
3	00111	s_3, s_4, s_5, s_6, s_0
4	01001	s_1, s_2, s_3, s_4, s_5
5	11010	s_6, s_0, s_1, s_2, s_3

Table 5.2: Pseudo-random patterns and corresponding LFSR states (adapted from [Wun96]).

First, the following initializations are performed:

$$r = 1,\ \text{ON-set} = \varnothing,\ \text{OFF-set} = \varnothing \text{ and } \text{Cov}(\text{BFF}^0) = \varnothing$$

Consider that all the faults included in F_{crit} can be detected by the patterns 11XXX and 0XX1X. The procedure for extracting the *essential* patterns $p_i \in P$ returns the patterns 2 and 3 in Table 5.2. Consequently, one obtains: $P = \{p_1, p_2\} = \{11101, 00111\}$. Consider that the analysis of *essential* bits transforms P to $P' = \{p_1', p_2'\} = \{11XXX, 0XX1X\}$. Table 5.2 can be used to look up for sets: FIX (p_1') and FIX (p_2').

$$\text{FIX-set} = \text{FIX}(p_1') \cup \text{FIX}(p_2') = \{s_5, s_6\} \cup \{s_3, s_6\} = \{s_3, s_5, s_6\} = \{100, 111, 011\}$$

Due to the fact that $\text{Cov}(\text{BFF}^0)$ is equal to $\varnothing$, it cannot be reduced anymore:

$$\text{ON-set} = \text{REDUCE}_{\text{FIX-set}}(\text{Cov}(\text{BFF}^0)) = \varnothing$$

$$\text{OFF-set} = \text{FIX-set} - \text{ON-set} = \text{FIX-set}$$

Let us assume that the deterministic test cube t = 00X00 has been generated and has to be mapped to one of the five pseudo-random patterns. Using the information in Table 5.2, one can derive the sets on(t, p) and off(t, p) of states in which the bit-flipping logic must be on or off. For every pattern, the condition on(t, p) $\cap$ FIX-set = $\varnothing$ is verified and cost(t, p) is computed. Table 5.3 shows the results.

#	p[1]…p[5]	on(t, p)	off(t, p)	cost(t, p)
1	10100	s_0	s_1, s_3, s_4	1
2	11101	s_5, s_6, s_2	s_1	∞
3	00111	s_6, s_0	s_3, s_4	∞
4	01001	s_2, s_5	s_1, s_4	∞
5	11010	s_6, s_0, s_2	s_3	∞

Table 5.3: Finding a pattern for mapping t = 00X00 (adapted from [Wun96]).

All patterns except the first one cannot be selected for mapping without violating the condition: on(t, p) $\cap$ FIX-set = $\varnothing$. The only way of mapping t is to modify the first pattern. So, the BFF should be accordingly extended:

$$\text{ON-set} = \text{ON-set} \cup \text{on}(t, p_0) = \{s_0\} = \{s_0\} = \{101\}$$

$$\text{OFF-set} = \text{OFF-set} \cup \text{off}(t, p_0) = \{s_1, s_3, s_4, s_5, s_6\} = \{010, 100, 110, 111, 011\}$$

Finally, the bit-flipping function is expanded in such a way that none of the terms in *OFF-set* is covered:

$$\text{Cov}(\text{BFF}^1) = \text{EXPAND}_{\text{OFF-set}}\ (\text{ON-set}) = \{\text{X01}\}$$

Figure 5.4 shows the corresponding pattern generator including the bit-flipping logic (Cov(BFF^1)). The set of patterns produced by the new test pattern generator differs considerably from the original one (Table 5.4). Nevertheless, patterns 2 and 3 are still compatible with the fixed patterns 11XXX and 0XX1X, and pattern 1 is now compatible with the deterministic test cube t = 00X00.

In general, the ON- and OFF-sets are very irregular and their cardinalities increase with the total number of specified bits in the embedded test cubes.

Unfortunately, the experimental results (Section 5.5) prove that the cube-based bit-flipping mapping approach scales poorly with the CUT size, more precisely, with the size of the ON- and OFF-sets. This is due to the very high (exponential) complexity of the cube-based methods used for the generation and the implementation of the BFF.

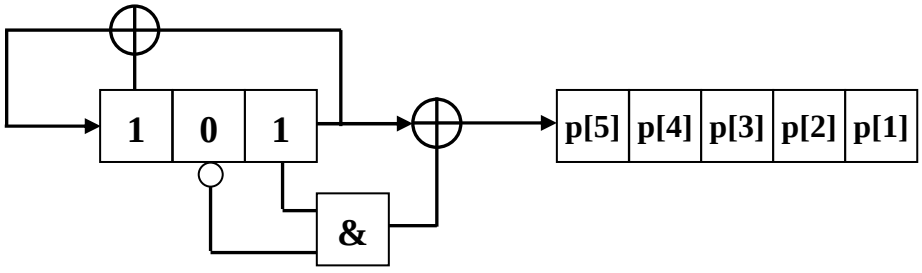


Figure 5.4: New pattern generator including bit-flipping logic (adapted from [Wun96]).

#	Old	New
1	10100	00000
2	11101	11000
3	00111	00110
4	01001	00001
5	11010	10000

Table 5.4: Old and new set of patterns (adapted from [Wun96]).

5.4 BDD-based Pattern Mapping

A scalable pattern mapping approach based on bit-flipping can be implemented only if a more efficient way to represent and manipulate the BFF is found. Such a way is described in this section. It is based on the use of ROBDDs (Section 4.3) for the represention and the manipulation of the *characteristic functions*[8] of the involved sets. For example, an ON-BDD and an OFF-BDD are employed to represent the ON-set and the OFF-set of the BFF. The ON-BDD will output the value '1' if the input is taken from the ON-set, otherwise the output is '0'. Similarly, the OFF-BDD will output '1', only if the input is selected from the OFF-set. In the sequel, the acronym BDD will be used in the sense of ROBDD.

As explained in Section 4.3.2, the BDD-based representation offers a more efficient way to manipulate Boolean functions than the cube-based representation. The complexity of the logic operations used here is at-maximum linear in the size of the BDD operands. In contrast to this, the cube-based logic manipulations can have up to an exponential complexity in the size of the operands, which may grow linearly with the number of embedded deterministic patterns and the CUT size.

In the worst case, the size of BDDs may grow exponentially with the number of input variables. Nevertheless, in practice the size of the BDD-based representation of the BFF has always been within practical limits to be handled by state-of-the-art computers and BDD software packages (e.g. [Cudd]).

Besides the underlying BDD-based representation, the other characteristics of the new pattern mapping approach are as follows:

- The pattern mapping is performed in a one-pass process and it is decoupled from the logic optimization of the resulting BFF.
- Only a sub-sequence of the whole pseudo-random test sequence is used to map deterministic test cubes. All useful pseudo-random test patterns not included in this sub-sequence are protected from being corrupted by the bit-flipping logic.

In the new approach, the test sequence is partitioned into two regions. The first part of the test sequence is used only for pseudo-random fault detection, and no deterministic stimuli are embedded into this part. In general, most of the faults of the CUT can be quickly detected by the first few hundred or thousand pseudo-random test patterns. The DLBIST hardware states associated to this first part are included into the DC-set, since increasing the DC-set gives more room for optimizing the bit-flipping logic. Consequently, the implemented BFF can arbitrarily flip bits of the *essential* pseudo-random patterns from this region, and some previously detected faults might no longer be detected. In order to prevent this, the outputs of the BFF are disabled during the first part of the test sequence. Disabling the BFF outputs is achieved with the help of only one single AND gate per scan chain controlled by a combination of the most significant bits of the PC.

[8] The characteristic function of an arbitrary set S is a completely specified Boolean function, whose ON-set is equal to S.

The second part of the test sequence is used only for the mapping of deterministic test cubes. The outputs of the BFF are enabled to modify only this last region of the test sequence, whose length is usually set to one fourth of the total test length.

The splitting of the test sequence into a pseudo-random and an embedded part together with the more efficient BDD-based representation and manipulation of the involved Boolean functions enable a decoupling of the pattern mapping from the synthesis of the resulting BFF. This is not the case with the cube-based approach that requires an iterative algorithm in order to take into account the potential corruption of *essential* pseudo-random patterns (Section 5.3.2) and to limit the overhead of the resulting bit-flipping logic. Consequently, with the BDD-based approach it is possible to use a one-pass algorithm that needs significantly lower run-time and memory requirements, while the overhead of the bit-flipping logic becomes much smaller (Section 5.5). The BDD-based algorithm for the generation and the implementation of the BFF is outlined in Figure 5.5.

The individual steps of the BDD-based flow are decribed below:

1. The sequence of pseudo-random test patterns generated by an LFSR and, optionally, a phase shifter (PS) is simulated to determine which non-redundent stuck-at faults of the CUT remain undetected.

2. An ATPG tool is used to generate a limited number of deterministic test cubes for a sub-set of the non-redundant stuck-at faults that remained undetected by the pseudo-random patterns. The deterministic cubes contain a large number of *don't care* bits. The number of new faults tested by these cubes depends on the size of the CUT, the pseudo-random fault efficiency, the required fault efficiency and the maximum number of deterministic cubes allowed for embedding.

3. A pseudo-random test pattern is assigned to each deterministic test cube. Each assigned pseudo-random test pattern is modified by bit-flipping to become compatible with the corresponding deterministic test cube. The mapping of the deterministic test cubes is done with the goal that the subsequent implementation of the BFF can be efficiently optimized. For each deterministic cube only a limited number of pseudo-random patterns are checked starting with the ones at the minimum Hamming distance. For these mapping candidates a combination of the following two objectives is used:

   - Minimize the number of clock cycles in which both matching and conflicting bits appear. This tries to make the outputs of the BFF_i's corresponding to different scan chains switch in phase. In this way, the logic sharing among the logic implementations of the corresponding BFF_i's can be maximized.

   - Minimize the number of scan chains which contain both matching and conflicting bits. This attempts to make some combination of the variables corresponding to the state bits of the PC to appear only in the satisfying set of either the ON-BDD or the OFF-BDD of a certain scan chain. This increases the degrees of freedom for optimizing the implementation of the corresponding BFF_i's.

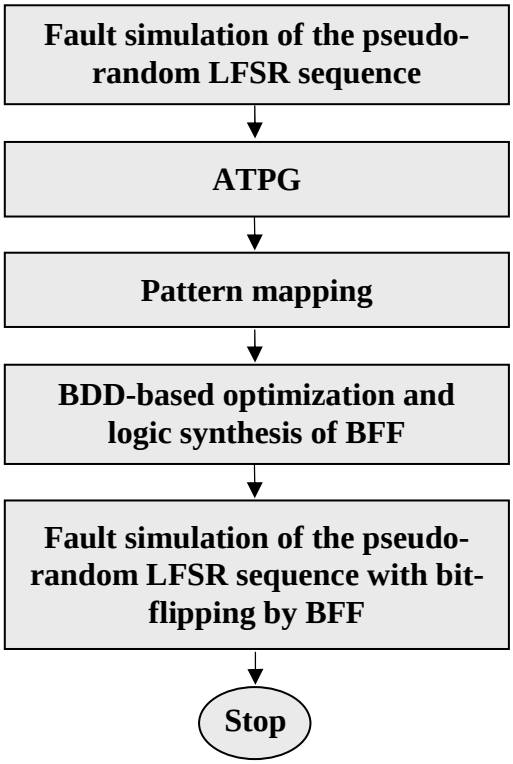


Figure 5.5: BDD-based pattern mapping by means of bit-flipping. A description of the program implementing this algorithm is given in Appendix 2.

For example, consider the simplified case where the PC counter has 4 state bits (x_0, x_1, x_2, and x_3) and only 3 deterministic test cubes have to be mapped. Assume that these cubes are mapped to the pseudo-random patterns which are generated when the PC state $x_0x_1x_2x_3$ is equal to 1001, 1100 and 0100, respectively. Consider also that with respect to the i^{th} scan chain the first pseudo-random pattern has only conflicting bits, the second pseudo-random pattern has both conflicting and matching bits and the third pseudo-random pattern has only matching bits as compared to the specified bits of the corresponding test cubes. If one neglects the other state bits of the DLBIST hardware (e.g. LFSR, SC, PS), the combination $x_0x_1x_2x_3$ = 1001 appears only in the satisfying set of ON-BDD_i while the combination $x_0x_1x_2x_3$ = 0100 is included only in the satisfying set of OFF-BDD_i. The combination $x_0x_1x_2x_3$ = 1100 appears in the satisfying sets of both BDDs. Consequently, the variables x_0, x_1 and x_3 can help in the implementation of BFF_i. One can choose $Cov(BFF_i) = x_3 + x_0 \cdot bff_i(x_4,\ldots, x_{n-1})$, where $x_4,\ldots, x_{n-1}$ are the variables corresponding to the other state bits of the DLBIST hardware and the function $bff_i(x_4,\ldots, x_{n-1})$ is used to implement the bit-flipping necessary for embedding the second test cube.

If all the assigned pseudo-random patterns had both matching and conflicting bits with respect to their corresponding test cubes, then the variables x_0, x_1, x_2, and x_3 could not be used for the optimization of $Cov(BFF_i)$. In such a case, the resulting $Cov(BFF_i)$ might be more complicated.

4. The BDD-based representation of the BFF is optimized by efficiently exploiting its DC-set and transformed into a RTL VHDL circuit description (Chapter 7). The circuit description can be synthesized using commercial logic synthesis tools, e.g. Synopsys Design Compiler.
5. In the end, a simulation of the embedded test sequence generated by the LFSR, the PS (if any) and the bit-flipping logic determines the final stuck-at fault coverage.

5.5 Experimental Evaluation of the BDD-based Approach vs. the Cube-based Approach

Experiments have been performed to evaluate the new BDD-based pattern mapping approach with respect to the original cube-based approach [Wun96]. The experimental setup and results are described in Appendix 1 (Table 5.5 – 5.9).

In the case when industrial benchmark designs are considered, the use of the BDD-based approach instead of the cube-based approach reduces the pattern mapping time from several days down to a few minutes. The run-times of the two other tasks, ATPG and fault simulation, are considerably improved as well. This is due to the fact that the BDD-based approach uses a single pass algorithm which involves ATPG and fault simulation less often than the original cube-based approach which uses an iterative algorithm, where ATPG, pattern mapping and fault simulation alternate [Wun96]. The BDD-based approach reduces the total run-time from more than a week down to several hours, while also the memory requirements scale quite well with the circuit size.

These amazing improvements are not achieved at the cost of fault efficiency and hardware overhead. The BDD-based approach outperforms the cube-based approach also with respect to these parameters.

In the experiments discussed so far, the fault efficiency of the BDD-based approach has been limited to the maximum reachable with the cube-based approach. It is shown that all relevant parameters of the BDD-based mapping approach, total run-time, memory consumption and cell area overhead scale very well also when the target fault efficiency is increased to the highest levels allowed by the ATPG tool. The logic overhead decreases significantly for the larger designs.

The experimental results reported till now prove the capability of the new mapping approach in achieving the scalability goal for which it has been devised. Nevertheless, it is still interesting to investigate how the BDD-based approach performs on smaller, but still difficult to test designs for which the cube-based approach is still efficient. ISCAS designs have been choosen for this purpose and the experimental results of this investigation are presented in Appendix 1 (Table 5.10 – Table 5.11).

For all the designs, it has been possible to reach higher final fault efficiencies with the BDD-based approach. With few exceptions, the total run-time, the memory consumption and the cell area overhead of the BDD-based approach are much lower.

Sometimes, the total run-time is reduced by even one order of magnitude. For the larger ISCAS designs the difference between the two approaches is more obvious. This proves the better scalability of the BDD-based algorithm, just like the experimental results for the large industrial designs.

5.6 Non-Target Defect Coverage and Overhead Dependence on Sequence Length

This section presents a study of the non-target defect coverage of the embedded test sequences obtained with the bit-flipping DLBIST scheme. This is the first time when such a study has been carried out. Resistive bridging faults are used as a surrogate of non-target defects [Eng05]. They accurately represent pattern dependency, Byzantine behaviour and other complex phenomena that are not considered by the stuck-at fault model. Due to the fact that the embedded deterministic test cubes target only the stuck-at faults, resistive bridging faults are a valid non-target defect surrogate. This investigation especially adresses the impact of the test sequence length on the non-target defect coverage and on the hardware overhead.

The algorithm used for these evaluations is outlined in Figure 5.6. Given the CUT, an LFSR is used to generate a pseudo-random test sequence. This sequence is simulated to determine its stuck-at and resistive bridging fault coverage. Subsequently, deterministic test cubes are produced for all non-reduntant stuck-at faults remained undetected. These cubes are mapped to the pseudo-random sequence and a BFF is generated. Next, the BDD-based representation of the BFF is optimized and transformed into a RTL VHDL circuit description. Finally, the test sequence generated by the LFSR and the bit-flipping logic is simulated for resistive bridging faults. Note that this embedded test sequence detects all non-redundant stuck-at faults not aborted by the ATPG tool.

The experimental data reported in Appendix 1 (Table 5.12 – 5.13) illustrates the impact of embedding deterministic test cubes for stuck-at faults and of the test sequence length on the coverage of resistive bridging faults and on the logic overhead of the bit-flipping logic.

The simulation results show that the resistive bridging fault coverage of the pseudo-random test sequences is consistently higher than their stuck-at fault coverage. On the other hand, the validity of stuck-at fault coverage in identifying circuits with many random pattern resistant resistive bridging faults appears to be limited. Circuits with many random pattern resistant resistive bridging faults may have a relatively reduced number of random pattern resistant stuck-at faults.

These results clearly demonstrate the importance of the embedded deterministic cubes, as the resistive bridging fault coverage increases significantly due to embedding. However, the pseudo-random patterns also seem to contribute to the detection of non-target defects. This is implied by the fact that applying more pseudo-random patterns results in appreciably higher resistive bridging fault coverage.

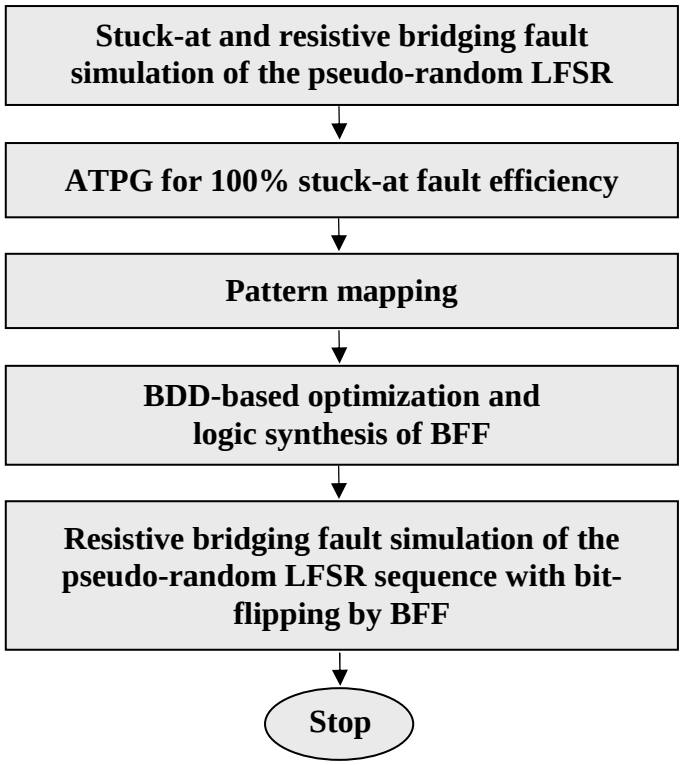


Figure 5.6: Evaluation of the effect of embedding deterministic test cubes into a pseudo-random sequence on non-target defect coverage.

Finally, the longer sequences require less logic overhead. This is due to two facts. First, the pattern embedding process has more degrees of freedom that can be exploited. Second, more stuck-at faults are covered by the pseudo-random sequence before pattern mapping. These faults do not have to be considered by the ATPG. Overall, there is a three-dimensional trade-off. Longer DLBIST sequences mean a larger test application time, but also less area cost and an enhanced coverage of both target and non-target defects.

5.7 Conclusion

In this chapter, a new pattern mapping algorithm has been proposed for bit-flipping and more generally for *test set embedding* DLBIST schemes. The new mapping method exploits the maneuverability and the compactness of the BDD-based function representation. Evaluations performed in the case of stuck-at fault testing have revealed that both run-time and memory requirements are improved by several orders of magnitude as compared to the original cube-based approach. Moreover, the proposed generation and implementation of the BFF does not require more run-time and memory resources than the ATPG or the fault simulation steps. This efficiency gain can be used to obtain even better solutions in terms of logic overhead and fault coverage.

For the first time, the effectiveness of the embedded test sequences obtained by mapping deterministic test cubes to pseudo-random test sequences has been evaluated with respect to the coverage of non-target defects. The resistive bridging fault model has been used to model non-target defects. The experimental results reveal that both deterministic test cubes and pseudo-random test sequences are useful for detecting non-target defects. Furthermore, it has been shown that increasing the length of the test sequences enhances their non-target defect coverage and significantly reduces the logic overhead. This increases the competitivity of the proposed DLBIST scheme and reduces the need for expensive *automated test equipment* (ATE).

Chapter 6

Deterministic Logic BIST for Transition Fault Testing

This chapter presents an extension of the bit-flipping DLBIST scheme described in Chapter 5 to make it also available for the test of transition faults. Some specific aspects of delay fault testing are considered below.

In order to test delay faults, two patterns are required, an *initialization pattern* that sets the circuit to a predefined state, and an *activation pattern* that launches the appropriate transition and propagates the fault effect to a (pseudo-)primary output. There are two approaches for the application of pattern pairs to a standard scan design [Sav92][Sav94][Wai87]: *functional justification*, also called *broadside*, and *scan shifting*, also called *skewed-load*. In the functional justification approach, the circuit response to the first pattern is used as the second pattern. In order to apply pattern pairs, the circuit is operated two consecutive clock cycles in functional mode after the initialization pattern has been scanned in. In the scan shifting approach, the second pattern is generated by operating the scan path for one additional scan clock cycle after the first pattern has been applied. Since scan shifting may require additional efforts for a consistent clocking scheme beyond the BIST hardware, only the functional justification approach will be considered. Another advantage of this approach is the expected limitation of the scan-induced overtesting, as long as the activation pattern is computed by the CUT itself and not scanned in, like in the scan shifting approach. So, the impact on the yield should be less than in the case of the scan shifting approach.

Here, the bit-flipping DLBIST scheme used for the test of stuck-at faults will be adapted to transition fault testing based on functional justification. LBIST approaches for the test of delay faults (especially for path delay faults) have been presented in [Che96][Duf97][Fur91][Gir97][Kei99][Li03][Muk98][Wur95], among others. Nevertheless, this is the first time when a DLBIST scheme is used to test delay faults in circuits with scan design.

The extension of the bit-flipping DLBIST scheme for transition fault testing requires the modification of the test control unit such that the *scan enable* signal allows two consecutive functional clock cycles and not only one as in the case of stuck-at fault testing.

Since pairs of test patterns are required, transition faults are more difficult to test than stuck-at faults. Consequently, the pseudo-random transition fault coverage is significantly lower than the pseudo-random stuck-at fault coverage. The final effect is an increase of the mapping effort and the logic overhead.

As a solution, this work proposes a trade-off between these costs and the test application time. Slightly larger test application times reduce logic overhead and enhance test quality in terms of both transition fault and non-target defect coverage (Chapter 5).

A quantitative estimation of the random testability of the stuck-at and transition faults is made in Section 6.1. The extension of the bit-flipping DLBIST scheme for transition fault testing is described in Section 6.2. Relevant experimental results for large industrial benchmarks, containing up to 2M gates, are reported in Section 6.3. The chapter is summarized in Section 6.4.

6.1 Random Testability of Transition and Stuck-at Faults

The probability that a stuck-at i fault ($i \in \{0,1\}$) on the signal line J is tested by a random pattern is equal to the probability P(J sa i) that the line J can be controlled to the logic value $\neg i$ and observed at a (pseudo-)primary output [Brg84][Sav84]:

P(J sa i) = P(J is controllable to $\neg i$ and J is observable)

The probability that a transition fault from i to $\neg i$ on the signal line J is tested by a pair of independent random patterns is equal to the probability P(J slow from i to $\neg i$) that the first pattern controls the line J to the logic value i multiplied by the probability that the second pattern controls the line J to the logic value $\neg i$ and can make the signal line J visible at a (pseudo-)primary output:

$$P(J \text{ slow from } i \text{ to } \neg i) = P(J \text{ is controllable to } i) * P(J \text{ sa } i) \quad (6.1)$$

From the relation (6.1), it results that if the stuck-at i fault on a signal line is random pattern resistant, then the transition fault from i to $\neg i$ on the same line is random pattern resistant as well. Consequently, a digital circuit contains at least as many random pattern resistant transition faults as random pattern resistant stuck-at faults.

The following expression gives the number N_f of random patterns required to test a fault f having the detection probability P_f (<<1) with the *test confidence*[9] C [Wun85].

$$N_f \approx - \ln(1-C) / P_f$$

Consequently, for two faults f and g with $P_f = P*P_g$ (P_f <<1, P<1) we have:

$$N_f \approx N_g / P \quad (6.2)$$

From the relations (6.1) and (6.2), one can observe that in the case when controllability to i of a signal line is close to 0, a slow transition fault from i to $\neg i$ on the same line may require a much larger number of random test patterns than the corresponding stuck-at fault, if the same test confidence is the objective.

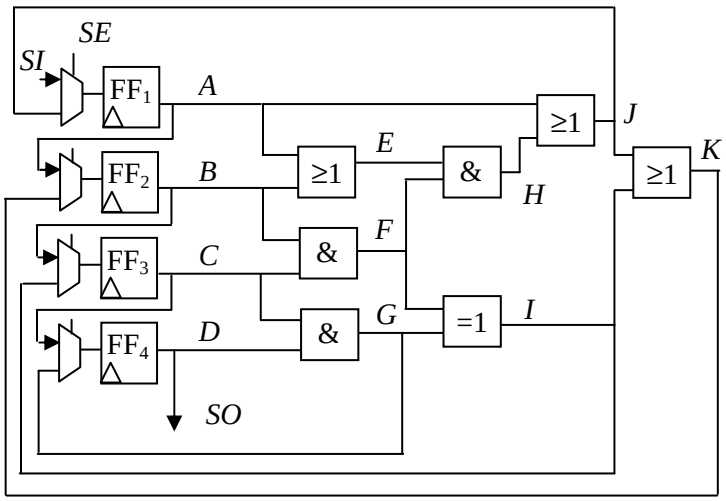
[9] Probability that the considered fault is tested by at least one pattern of the test sequence.

As an example, consider the circuit sketched in Figure 6.1. The inputs of this circuit are driven by four scan flip-flops. A *scan enable* signal (SE) is used to switch the scan flip-flops between scan and functional modes. The possible patterns to test the stuck-at '0' and '1' faults on the net *J* are *ABCD* ∈ {1XXX, X11X} and {00XX, 0X0X}, respectively. Based on functional justification, the initialization pattern *ABCD* = 0011 will generate an activation pattern 0111 such that a slow-to-rise fault on the net *J* can be tested. The slow-to-fall transition fault on the net *J* cannot be tested.

Note that there is only 1 initialization pattern for the slow-to-rise fault on the net *J* which has 4 specified bits, while each of the 2 corresponding stuck-at faults has 2 test patterns with 1 or 2 specified bits. The probabilities to randomly detect the slow-to-rise, the stuck-at '0' and the stuck-at '1' faults on the net *J* are 1/16, 5/8 and 3/8, respectively. ($1/16 < (3/8)^2 < (5/8)^2$). The reduced testability of the transition fault is due to the fact that the initialization pattern must not only set the required initial logic value on the target line, but it must also generate appropriate logic values at the CUT outputs in order to define an useful activation pattern.

Figure 6.2 presents a comparison between the cumulative stuck-at and transition fault coverage of a pseudo-random sequence applied to an industrial benchmark design with 5116 flip-flops arranged into 11 scan chains and 127K nodes in the net list. The lower level and the slower saturation of the transition fault coverage is due to the larger number of random pattern resistant transition faults and to the larger number of pseudo-random patterns required by these faults to achieve the same test confidence as in the case of the random resistant stuck-at faults.

This slow saturation is expected for any type of delay fault testing and may be enhanced in the case of robust delay fault testing. It increases the necessity of having long test sequences when DLBIST is used. Moreover, a new DLBIST architecture is necessary that is able to use the whole test sequence for pseudo-random fault detection and not only one fraction of it as in the case of the architecture presented in Chapter 5.



Patterns which can test the stuck-at 0, stuck-at 1 and slow-to-rise faults on the wire *J*:

	A	*B*	*C*	*D*
stuck at-0	1	X	X	X
stuck at-1	0	0	X	X
slow-to-rise	0	0	1	1

slow-to-fall is untestable

Figure 6.1: Specified bits for testing stuck-at and transition faults.

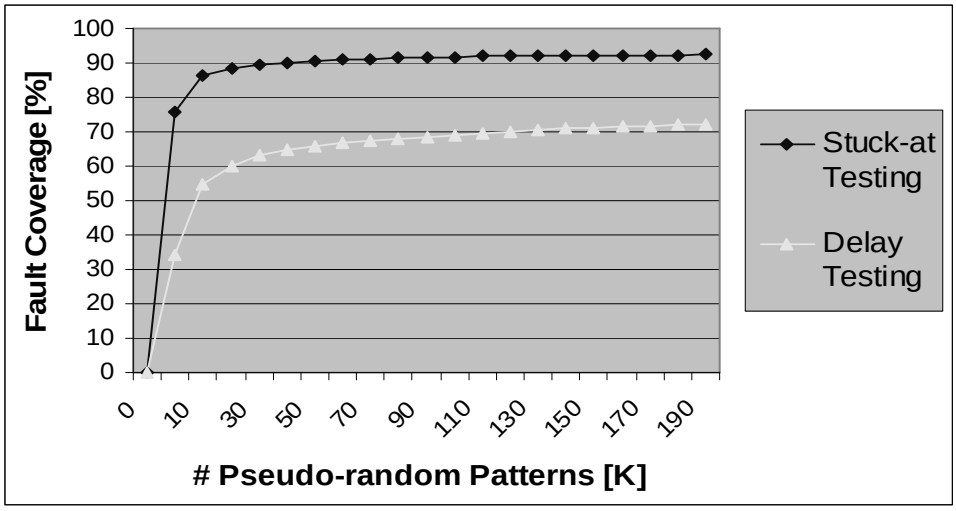


Figure 6.2: Cumulative stuck-at and transition fault coverage of a pseudo-random sequence applied to an industrial benchmark design that contains 5116 flip-flops arranged into 11 scan chains. The transition fault testing was based on functional justification.

6.2 Bit-flipping Deterministic Logic BIST for Transition Fault Testing

Applying a DLBIST scheme to transition fault testing is a challenge due to the lower random testability of the transition faults. This requires more deterministic cubes with more specified bits to be embedded into the pseudo-random sequence as compared to the case of stuck-at fault testing.

In the case of transition fault testing based on functional justification, only the initialization pattern of each pair of test patterns should be generated by the DLBIST hardware. The activation pattern is generated by the CUT as a response to the first pattern and only single test cubes have to be embedded into the pseudo-random sequence. Consequently, the DLBIST synthesis flow for transition fault testing based on functional justification may be derived by adapting the flow used for stuck-at fault testing, provided that the ATPG and the fault simulation are correspondingly modified. This is also true in the case of transition fault testing based on scan shifting, with the observation that in this case the test control unit should generate one additional shift clock cycle instead of the first clock cycle of each pair of functional clock cycles. For this reason, the approach presented here can also be applied to scan shifting approaches.

Here, each pattern of a test sequence that can detect faults not detected by any of its precedent patterns is referred to as an *essential* pattern (Section 5.3.2). The embedding of deterministic test cubes into a pseudo-random test sequence may corrupt the *essential* pseudo-random patterns even if they are not assigned to deterministic cubes. This is due to the fact that the logic synthesis and optimization of the BFF are intensively using its DC-space (Chapter 7). Consequently, the resulting bit-flipping

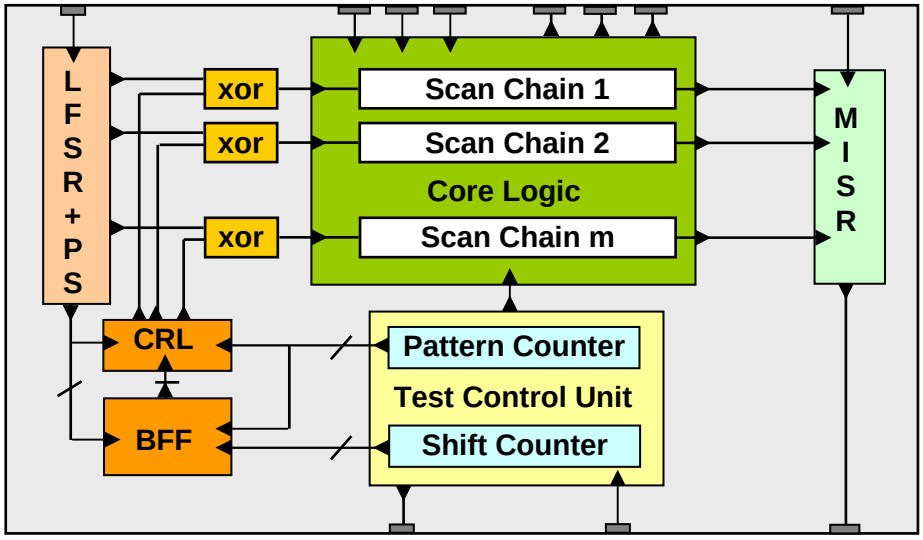


Figure 6.3: Architecture of the bit-flipping DLBIST.

logic flips more bits than necessary for the embedding of the target deterministic test cubes and the consequence is that *essential* pseudo-random test patterns may be corrupted .

In order to limit the number of corrupted pseudo-random patterns in the case of the architecture presented in Chapter 5, the set of deterministic cubes is embedded only at the end of the pseudo-random test sequence. The length of the pseudo-random sequence that can be modified is a fraction given by a negative power of 2 of the total test length. The first part of the test sequence is used only for pseudo-random fault detection and it is protected from being flipped by disabling the outputs of the BFF with the help of an AND gate per scan chain, which is controlled by a combination of the most significant bits of the pattern counter. Nevertheless, the *essential* pseudo-random patterns of the embedded test sequence are not protected.

Due to the slower saturation of the random transition fault coverage discussed in Section 6.1, the application of longer test sequences and the protection of all the *essential* pseudo-random patterns become critical. A way to prevent the corruption of the *essential* pseudo-random patterns without increasing the complexity of the BFF implementation is to utilize an additional combinational module, here referred to as *correction logic* (CRL), to enable/disable the outputs of the bit-flipping logic (Figure 6.3). The partition of the test sequence into pure pseudo-random and embedded sequences is preserved to limit the CRL size.

The algorithm used for the generation of the bit-flipping DLBIST for transition fault testing based on functional justification is outlined in Figure 6.4. The individual steps of the flow are described below:

1. Initial fault simulation is used to detect the transition faults that cannot be tested by the pseudo-random test sequence produced by an LFSR and, optionally, a phase shifter (PS). During this step a list L_1 is generated containing the indices of all the *essential* pseudo-random patterns of the test sequence part where deterministic test cubes will be embedded.
2. An ATPG tool is used to generate a limited number of deterministic initialization test cubes for all or a sub-set of the transition faults that remained undetected by pseudo-random test patterns.

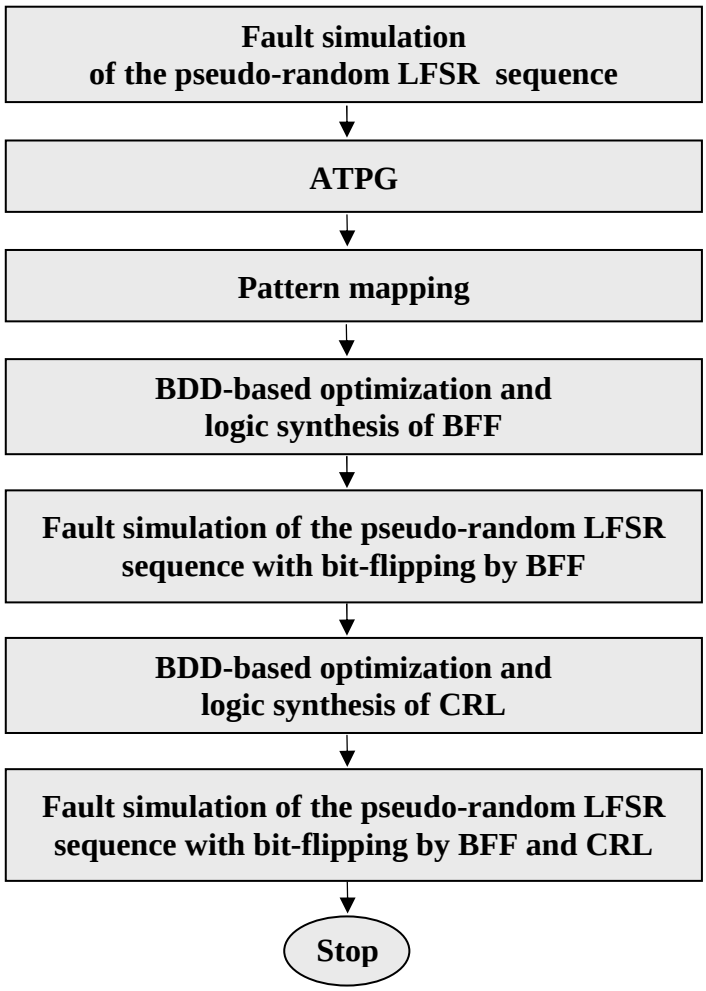


Figure 6.4: Implementation flow of the bit-flipping DLBIST for transition fault testing based on functional justification. A description of the program implementing this algorithm is given in Appendix 2.

The number of new faults tested by the deterministic cubes depends on the size of the CUT, the pseudo-random fault efficiency, the required fault efficiency and the maximum number of deterministic test cubes allowed for embedding.

3. A pseudo-random test pattern is assigned to each deterministic initialization test cube. The same mapping costs are utilized as in the flow used for stuck-at fault testing. Nevertheless, this time the *essential* pseudo-random test patterns are not allowed to be assigned. Each assigned pseudo-random test pattern is modified by bit-flipping to become compatible with the corresponding deterministic test cube.

   During this step, a BDD-based representation is generated for the resulting BFF. The BFF is only partly specified and has a large DC-set.

4. The BDD-based representation of the BFF is optimized and transformed into a RTL VHDL circuit description (Chapter 7). The logic optimization procedure exploits the DC-set left by the incomplete specification of the BFF. Consequently, the optimized bit-flipping logic flips additional bits besides the conflicting bits in the assigned pseudo-random test patterns, so that the *essential* pseudo-random test patterns (with the index included in L_1) may be corrupted.

5. In order to determine which of the patterns with the index in L_1 must be protected from being corrupted, the embedded test sequence produced by the LFSR, PS (if any) and the bit-flipping logic, which is not allowed to act on the test patterns with the index in L_1, is simulated to generate a second list L_2 of indices corresponding to the *essential* patterns in the embedded test sequence. The test patterns whose indices are included in $L_1 \cap L_2$ need to be protected from being corrupted by the bit-flipping logic. On the other hand, the test patterns whose indices are included in $L_2 - L_1$ should remain modified by the bit-flipping logic.

6. A combinational logic called *correction logic* (CRL) is built (Figure 6.3). The CRL prevents the bit-flipping logic from flipping the patterns with the index in $L_1 \cap L_2$ and allows it to modify the patterns with the index in $L_2 - L_1$. In this way, the *essential* pseudo-random test patterns (referenced in $L_1 \cap L_2$) will not be corrupted, while the *useless* test pseudo-random patterns that become useful by means of bit-flipping can still be generated. The CRL implements an incompletely specified function whose ON-set and OFF-set contain the states of the pattern counter, LFSR and the output of the PS (if any) corresponding to the first scan clock cycle of the test patterns with the index in $L_2 - L_1$ and $L_1 \cap L_2$, respectively. The DC-set of the CRL function is used to optimize its implementation, exactly as in the case of the BFF implementation (Chapter 7).

   The output of the CRL has to be kept constant during the scan clock cycles corresponding to each test pattern. Due to the fact that some of the input signals to the CRL (the state bits of the LFSR and the output bits of the PS (if any)) change during the scan clock cycles, a latch is used to store the output of the CRL (Figure 6.5). The operation of the latch is controlled with the help of the SE signal, so that its state can be changed only before a new test pattern is scanned in.

   Without the CRL, all the inputs of the BFF corresponding to the *essential* pseudo-random patterns should be included into the OFF-set of the BFF. Consequently, the use of the CRL leaves more DC-space to optimize the logic implementation of the BFF. On the other hand, storing the output of the CRL into a memory element that cannot be written during the shift cycles increases significantly the degrees of freedom for the optimization of the CRL.

   While a BFF function has to be implemented for each scan chain, the CRL is common for all the scan chains. The circuit description of the BFF and CRL can be synthesized using commercial logic synthesis tools (e.g. Synopsys Design Compiler).

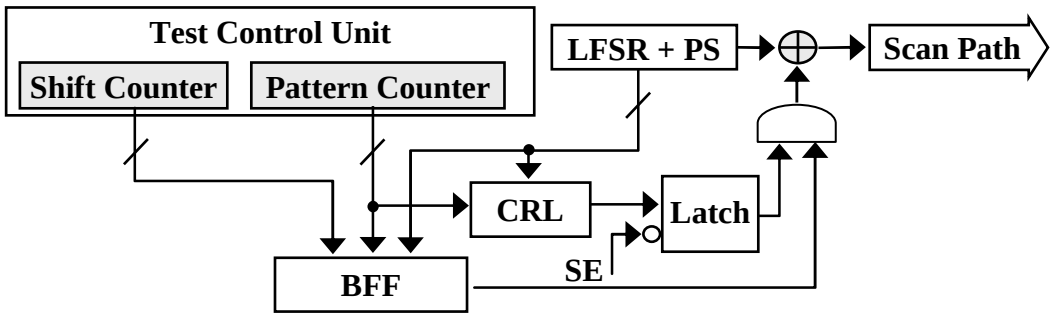


Figure 6.5: Bit-flipping function (BFF) and correction logic (CRL).

7. In the end, the embedded test sequence generated by the LFSR, the PS (if any), the bit-flipping and the CRL is simulated to determine the final transition fault coverage.

The use of the CRL can be benefical even in the case of stuck-at fault testing. This is illustrated in Table 6.1 (Appendix 1).

6.3 Experimental Results

Experiments have been performed to evaluate the bit-flipping DLBIST approach with respect to transition fault testing. The experimental setup and results are described in Appendix 1 (Table 6.2 – 6.4).

It can be observed that, indeed, the pseudo-random transition fault efficiency is much lower than the pseudo-random stuck-at fault efficiency. In general, increasing the test sequence length has a stronger impact on the transition fault detection than on the stuck-at fault detection.

Comparing the results obtained using the bit-flipping DLBIST approach for the stuck-at and transition fault testing, one can observe that, with only one exception, the deterministic test cubes embedded for transition fault testing have larger ratios of specified bits. This is due to the lower transition fault testability. In all the cases, the final stuck-at fault efficiency is much larger than the final transition fault efficiency. Moreover, this has been achieved along with a lower cell area overhead. The reason for this difference is again the lower random testability of the transition faults with the consequence that more patterns have to be embedded and more bits have to be flipped or preserved in the pseudo-random sequence. For a given test length, the DLBIST hardware overhead depends on the random testability of the CUT and on the amount with which the fault efficiency has to be increased.

The hardware overhead of the designs for which the number of embedded patterns has not been limited is significantly reduced by the increase of the test sequence length. Extending the test length from 10K to 64K reduces the overhead by more than 10% of the CUT size. In one case, increasing the test length by two orders of magnitude has reduced the overhead to half of the level from the previous entry that corresponds to a test sequence containing 64K patterns, at the price of a large increase in the run-time and memory requirements.

As long as the same number of deterministic test cubes is embedded, it is difficult to predict the dependence of the hardware overhead on the length of the test sequence. In this case, the overhead primarily depends on the average number of specified bits per embedded test cube, which is determined by the number and the difficulty of the target faults. Longer pseudo-random test sequences leave undetected faults which are more difficult to test. This tends to increase the number of specified bits necessary to detect the remaining fault. On the other hand, this may also decrease the number of newly detected faults per embedded test cube. That is why it is difficult to predict the evolution of the average number of specified bits per embedded test cube when the length of the test sequence is augmented. Increasing the length of the test sequence also improves the pattern embedding opportunities.

In the case of one design for which the number of embedded deterministic test cubes has been limited, increasing the length of the test sequence does not significantly change the hardware overhead, but it improves the final fault efficiency by more than 11%. In the other two such cases, extending the test sequence has a twofold beneficial impact. Incresing the number of the test patterns from 10K to the maximum that pass in a test application time of one second at the frequency of 100 MHz reduces the overhead by 11% and 7%. In parallel, the final fault efficiency is improved by more than 8% and 3%. It should be mentioned that the increase of the test sequence length improves the coverage of the non-modeled defects as well (Section 5.6).

In the case of the three largest designs, increasing the length of the test sequence has no significant impact on the run-time and memory requirements.

Investigating the trade-offs between the hardware overhead and the fault efficiency, which were obtained using test sequences that contain the maximum number of patterns which can fit in one second of test time at the frequency of 100 MHz, the following observations can be made. In the case of two large benchmark designs, 10 deterministic test patterns are already enough to obtain a larger fault efficiency than in the case when 800 deterministic test patterns are embedded into a 10K long test sequence. In this way, the hardware overhead can be reduced to 1% from 43% and 62%, respectively. In the case of the other large benchmark design used during the experiments, a similar fault efficiency can be achieved by embedding 100 deterministic patterns, at the cost of 5.5%, instead of 22%, hardware overhead.

6.4 Conclusion

In this chapter, an extension of the bit-flipping DLBIST approach for transition fault testing has been presented. This is the first time when a DLBIST scheme is used to test delay faults in circuits with standard scan design. The investigated delay testing approach is based on *functional justification*, but the scheme can also be applied with a minimum modification to an approach based on *scan shifting*. A special combinational module, the *correction logic* (CRL), has been introduced to further improve the test pattern embedding. Due to the rather low random-pattern testability of the transition faults, the saturation of their random fault coverage requires significantly longer test sequences, which in turn is beneficial for both limiting the hardware overhead and improving the coverage of the target and non-target defects.

Chapter 7

Scalable Synthesis of Irregular Combinational Functions with Large Don’t Care Sets

This chapter presents an innovative BDD-based logic synthesis method which is especially suitable for the logic implementations of *irregular* functions that have large *don’t care* sets. Here, a Boolean function is called irregular if its input assignments mapped to ‘1’ are randomly spread over the definition space. Examples of such functions are the BFF, the function implemented by the CRL, the BFX [Tou96] and the XMF [Tan04].

This is the first technique that exploits the DC-set together with the compactness of FBDDs (Definition 4.12) to improve the efficiency of the BDD-based logic synthesis. The presented experimental results show that for all the considered functions, implementations are found with a significant reduction of the gate count compared to SIS [Sen92] or the methods offered by a state-of-the-art BDD-package [Cudd]. This performance is due to both a reduction of the node counts in the resulting FBDDs and to a reduced number of gates needed to implement the FBDD nodes. The proposed method scales better and succeeds to get a better advantage of the DC-set.

Two examples of irregular Boolean functions with large DC-sets are analysed in Section 7.1. Section 7.2 presents a new heuristic method to improve the cover synthesis for such functions. In Section 7.3, experimental results are used to compare the proposed approach with SIS [Sen92] and methods available in the CUDD-package (e.g. *restrict* [Cou90]). Furthermore, the outcome of the new method is evaluated as input to Synopsys Design Compiler. The chapter is summarized in Section 7.4.

7.1 Examples of Irregular Incompletely Specified Boolean Functions

The bit-flipping function (BFF) and the bit-fixing function (BFX) [Tou96] are examples of irregular and incompletely specified functions with large DC-sets. Besides these functions, another such example will be described in this section. Like the BFF function, also this example comes from the field of coding and testing.

In most embedded test approaches [Ghe04][Koe01][Raj02][Tou96], the test responses are compressed by a multi-input shift register (MISR) (Figure 7.1), which delivers a signature containing the information about the correctness of the CUT. The test responses may contain unknown bits (Xs), which can appear due to the existence of multiple clock domains, floating buses or uninitialized memory elements. In order to obtain an uncorrupted signature at the end of the test, these Xs have to be *masked* to either logic '0' or logic '1' before they propagate into the MISR. This may be performed by combinational logic implementing a so-called *X-masking function* (XMF) [Tan04]. The XMF can be kept quite small by carefully selecting those bits of the test responses carrying the information about the CUT correctness which have to remain unmasked.

The inputs of the BFF and the XMF are the state bits of the pattern counter, the shift counter and the test pattern generator (TPG) which can be an LFSR and, eventually, a phase shifter. Both functions are incompletely specified functions. Consequently, they can be described by an ON-set and an OFF-set, containing the input assignments for which these functions must take the values '1' and '0', respectively. The remaining input assignments build the DC-set.

According to Chapter 5, the ON-set and the OFF-set of the BFF are the sets of states that correspond to the clock cycles in which the TPG output must or must not be flipped, respectively. The DC-set is the set of states that correspond to the clock cycles in which the LFSR output may be arbitrarily flipped.

In the case of the XMF, the ON-set is the set of states that correspond to the clock cycles in which an unknown test response bit must be masked before it is scanned into

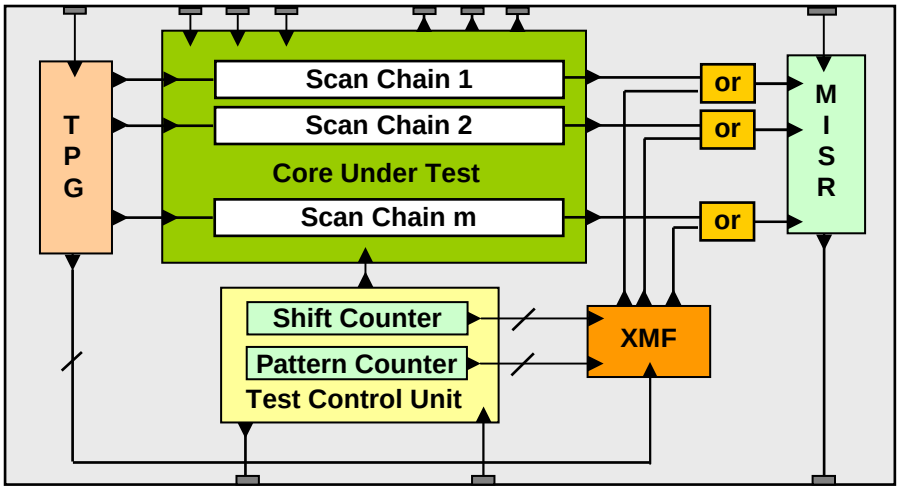


Figure 7.1: Embedded test architecture with MISR and X-masking function (XMF).

the MISR. Similarly, the OFF-set is the set of states that correspond to the clock cycles in which a test response bit carrying the information about the CUT correctness must not be masked. The DC-set contains the states that correspond to the clock cycles in which the test response bits may be arbitrarily masked before they are propagated into the MISR.

The DC-sets cover more than 99.99% of the definition space of both functions while the ON-sets and OFF-sets are randomly distributed over the rest of the definition space. One can identify the following sources of the high cardinality of the DC-sets.

- Not all the possible states and state combinations of the shift counter, pattern counter, LFSR and phase shifter (Figure 7.1) are necessarily appearing during the testing process.
- In the case of the BFF, the deterministic test cubes that have to be mapped to the pseudo-random test sequence contain many *don't care* bits and the number of embedded deterministic test cubes is a small fraction of the total number of pseudo-random test patterns.
- In the case of the XMF, usually a very small fraction of the bits in test responses are Xs or relevant to the fault coverage.

The large DC-sets offer a good base to optimize the logic implementation of these functions despite their irregularity, which is not the case with random functions with no or small DC-sets.

7.2 Proposed FBDD-based Logic Synthesis

This section proposes a new synthesis approach that transforms the ROBDD-based representation of an incompletely specified Boolean function into a FBDD-like cover whose circuit description requires a reduced number of gates. The following considerations are based on the notations introduced in Chapter 4 and on the definition below.

Definition 7.1 The *cardinality* of a function $f:\{0,1\}^n \rightarrow \{0,1\}$, denoted by $||f||$, indicates the number of fully specified input assignments mapped to '1', i.e. the number of minterms (Definition 4.7).

The goal of the synthesis procedure described below is to generate FBDD-like covers with a reduced gate count in the resulting circuit descriptions. This is achieved by first reducing the number of paths from the root node to a leaf node and second by looking for node sharing among different paths and even different FBDDs.

On one hand, each path in a BDD corresponds to a sub-space which is mapped either to '1' or to '0'. Similarly, the cover of an incompletely specified function $F(f_{on}, f_{off})$ can be chosen equal to '0' on the subspaces mapped by f_{on} to '0' and equal to '1' on the subspaces mapped by f_{off} to '0'. Consequently, the path reduction of the FBDD-based implementation can be achieved by finding a minimal partition of the definition space of the considered function into appropriate sub-spaces on which either f_{on} or f_{off} is equal to '0'. Given the incompletely specified function $F(f_{on}, f_{off})$ and the set of its input variables V, the synthesis method introduced here looks for a *good* partition of

the definition space into such special sub-spaces using the recursive depth-first process sketched below.

```
CreateCover (f_on, f_off, V){
    // V contains the indices of all the relevant input variables
    if size(f_on) > size(f_off) then  // size = ROBDD node count
        return ¬CreateCover (f_off, f_on, V);
    l = ∅;
    Cov = CreateLiteralCover (f_on, f_off, V, l);
    if (Cov ≠ ∅) then return Cov;
    Cov = FindCover (f_on, f_off); // optional: DC-based node reduction
    if (Cov ≠ ∅) then return Cov;
    for all i ∈ V and for l_i ∈ {x_i, ¬x_i}
        if f_on|li = 0 and f_off|li = 0 or
        if f_on|li = f_on|¬li and f_off|li = f_off|¬li then V = V – {i};
    if l ≠ ∅ then
        Cov = CreateCover (f_on|¬l, f_off|¬l, V);
        if f_off · Cov = 0 then return Cov;
        else return (¬l) · Cov; // new FBDD-node required
    return SplitOperator (f_on|¬l, f_off|¬l, V);
}
```

First, it is decided whether $F(f_{on}, f_{off})$ or $\neg F(f_{off}, f_{on})$ is implemented, depending on the compactness of the ROBDD-based representation of f_{on} and f_{off}. The ROBDD sizes are determined by their node count.

Subsequently, a variable x is determined (procedures *CreateLiteralCover* or *SplitOperator*) with respect to which the current definition subspace is decomposed into two new subspaces where x is either '1' or '0'. For each of the two subspaces a further recursive call of *CreateCover* may be required. The size of the resulting cover may be reduced by determining a minimal number of such successive recursive calls. Procedures *CreateLiteralCover* and *SplitOperator* implement heuristics to obtain near-optimal solutions.

```
CreateLiteralCover (f_on, f_off, V, l){
    Min = ∞;
    for all i ∈ V and for l_i ∈ {x_i, ¬x_i}
        if f_on|li = 0 and || f_off|¬li || < Min then
            Min = || f_off|¬li ||; l = l_i;
    if Min ≠ ∞ then
        if f_off|¬l = 0 then return ¬l;
    return ∅;
}
```

The procedure *CreateLiteralCover* provides the recursive process with the first stop condition. The recursion is stopped if a literal l is found for which $f_{off}|_{\neg l}$ and $f_{on}|_{l}$ are

equal to '0'. In this case $\neg l$ is chosen as cover for F. If this condition cannot be fulfilled and there are literals l_i, for which $f_{on}|_{li}$ is equal to '0', then that literal l_i which minimizes the cardinality of $f_{off}|_{\neg li}$ will be assigned to the generic argument l.

The procedure *FindCover* which provides the algorithm with the second stop condition is optional and will be discussed later.

Subsequently, the set of input variables V is pruned from those variables on which f_{on} and f_{off} depend in a *trivial* way (*for* loop of *CreateCover*). Depending on whether the literal l returned by *CreateLiteralCover* is different from the empty set $\emptyset$, either *CreateCover* or *SplitOperator* is called.

SplitOperator (f_{on}, f_{off}, V){
// first heuristic (| *val* | used for the absolute value of *val*)
$Max = 0$;
for all $i \in V$
C*heck* = | $||f_{on}|_{xi}||$ - $||f_{off}|_{xi}||$ | + | $||f_{off}|_{\neg xi}||$ - $||f_{on}|_{\neg xi}||$ |;
if C*heck* > *Max* **then** *Max* = *Check*; $m = i$;
// second heuristic
if *Max* = | $||f_{off}||$ - $||f_{on}||$ | **then**
MinOn = ∞; *MinOff* = ∞;
for all $i \in V$ **and for** $l_i \in \{x_i, \neg x_i\}$
if $||f_{on}|_{li}|| < MinOn$ **or**
if $||f_{on}|_{li}|| = MinOn$ **and** $||f_{off}|_{\neg li}|| < MinOff$ **then**
$MinOn = ||f_{on}|_{li}||$; $MinOff = ||f_{off}|_{\neg li}||$; $m = i$;
// choose the literal for the first recursion
$V = V - \{m\}$;
choose $l \in \{x_m, \neg x_m\}$ **such that** $||f_{off}|_{l}|| \geq ||f_{off}|_{\neg l}||$;
Cov_1 = **CreateCover** ($f_{on}|_l$, $f_{off}|_l$, V);
if $Cov_1 \cdot f_{off} \neq 0$ **then**
Cov_2 = **CreateCover** ($f_{on}|_{\neg l}$, $f_{off}|_{\neg l}$, V);
else if $f_{off}|_{\neg l} \neq 0$ **then**
Cov_2 = **CreateCover** (($\neg Cov_1$)·$f_{on}|_{\neg l}$, $f_{off}|_{\neg l}$, V);
else $Cov_2 = \neg 1$;
// assemble the cover: new FBDD-node required
if $Cov_1 \cdot f_{off} \neq 0$ **then** $Cov_1 = l \cdot Cov_1$;
if $Cov_2 \cdot f_{off} \neq 0$ **then** $Cov_2 = \neg l \cdot Cov_2$;
return $Cov_1 + Cov_2$;
}

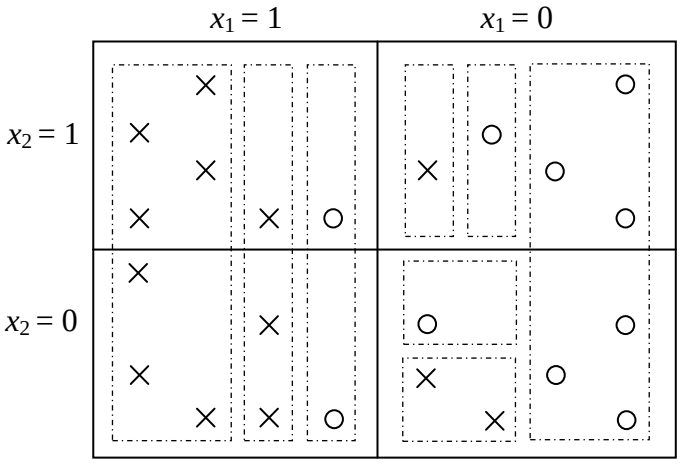
Procedure *SplitOperator* uses two heuristics. The first one looks for a literal l such that the cardinalities $||f_{on}|_l||$ and $||f_{off}|_{\neg l}||$ are higher than the cardinalities $||f_{off}|_l||$ and $||f_{on}|_{\neg l}||$, respectively. If such an *unbalancing* occurs, then the following inequality must hold:

$$| ||f_{on}|_x|| - ||f_{off}|_x|| | + | ||f_{off}|_{\neg x}|| - ||f_{on}|_{\neg x}|| | > | ||f_{off}|| - ||f_{on}|| | \quad (7.1)$$

The intuition behind the unbalancing is that we heuristically try to find the variable x that simultaneously minimizes both cardinalities $f_{on}|_l$ and $f_{off}|_{\neg l}$ ($l \in \{x, \neg x\}$). For example, consider the definition space presented in Figure 7.2, where the symbols 'x' and 'o' are used to represent the input assignments belonging to the ON-set and the OFF-set of the considered function, respectively. The dashed squares give a minimal partition of the definition space into sub-spaces containing only input assignments belonging either to the ON-set or to the OFF-set. Assume that one has to choose between the input variables x_1 and x_2 for the decomposition of the considered definition space. The enclosed table shows the number of input assignments belonging to the ON-set and the OFF-set in the sub-spaces defined by $x_1 = 1$, $x_1 = 0$, $x_2 = 1$ and $x_2 = 0$. The other input variables are not explicitly shown for simplicity reasons. In this case, the first heuristics of the procedure *SplitOperator* chooses the variable x_1 with respect to which the definition space is *unbalanced* and the inequality (7.1) is fulfilled. The left-hand side member of the inequality (7.1) is evaluated to 15/3 with respect to the variable x_1/x_2. In total, there are 13/10 input assignments belonging to the ON-set/OFF-set, so that the right-hand side member of the inequality (7.1) is evaluated to 3. It can also be observed that the cut line corresponding to the decomposition of the definition space with respect the input variables x_1 does not intersect any sub-space of the minimal partition. This does not happen in the case of the variable x_2.

If no *unbalancing* variable has been found, then the second heuristic is used. This heuristic chooses the variable x, which has an associated literal $l \in \{x, \neg x\}$ that minimizes the cardinality $||f_{on}|_l||$ as a primary optimization goal and minimizes the cardinality $||f_{off}|_{\neg l}||$ as a secondary optimization objective. The intuition behind this is similar to the one mentioned for the first heuristic of *SplitOperator*. For each literal $l \in \{x, \neg x\}$ a recursive call with the argument $(f_{on}|_l, f_{off}|_l)$ is performed iff $f_{off}|_l \neq 0$.

Both heuristics in *SplitOperator* are used to increase the chance of fulfilling the stop condition from *CreateLiteralCover* in the next recursive calls and thus to decrease in a greedy manner the number of subsequent recursive calls of the procedure *CreateCover*.



	ON-set	OFF-set
$x_1 = 1$	10	2
$x_1 = 0$	3	8
$x_2 = 1$	6	5
$x_2 = 0$	7	5

Figure 7.2: Example of the proposed decomposition of the definition space.

In order to limit the memory consumption of the whole process, the cofactor $f|_x$ is computed using the operator *BDD.Compose* instead of the operator *BDD.And*. In this way, the dependence of the cofactor $f|_x$ on the variable x is eliminated.

The heuristics used here to choose the new variable x depend only on the distribution of the ON-set and of the OFF-set over the definition space of the target function F. This makes the algorithm largely independent of the variable order used for the underlying ROBDD-based representation, which is not the case with the heuristic used in [Gue99], which has been proposed only for completely specified functions.

The structure of the resulting $Cov(F)$ can be efficiently modeled as a FBDD (Definition 4.12). A FBDD-based representation is preferred in this case, since an OBDD could require excessive memory usage. Consequently, in this process a FBDD-based representation is constructed node by node. Each non-terminal node of the FBDD is created during a distinct recursion step. A node created outside *SplitOperator* requires at most one 2-input logic operator, while a node created inside *SplitOperator* may require between one and three 2-input logic operators. *NAND* and *NOR* operators are preferred to *AND* and *OR* operators. In this way the logic is optimized not only by reducing the number of nodes in the FBDD, but also by reducing the operator count per node. Both goals are achieved by exploiting the DC-set.

So far, the node count has been minimized only by attempting to decrease the path count (e.g. looking for minimal partitions of the definition space, where either f_{on} or f_{off} is equal to '0'). The node count can be further reduced by allowing non-terminal nodes to become children of more than one parent node and by allowing parent nodes of the same child to belong to FBDDs corresponding to different outputs of the target function. This is nothing else than the well-known node reduction [Bry86] that usually makes the ROBDDs very compact, but which in the case of FBDDs is expected to have less impact on the node count.

Procedure *FindCover* is used to check whether the covers $Cov(SG)$ implemented by already synthesized sub-graphs SG are useful also in the case of the target function $F(f_{on}, f_{off})$. If such a sub-graph is found, one has only to point to its root node with a normal or a complemented edge (when $\neg Cov(SG)$ is required).

In order to reduce the node depth of the cover returned by *CreateCover*, it is important that *CreateLiteralCover* is called before *FindCover*. In order to increase the chances that a cover will be found by *FindCover*, this should be called before the *for* loop in *CreateCover*.

FindCover (f_{on}, f_{off}){
  for each element SG of a sub-set of all completed sub-graphs
    if $f_{on} \cdot Cov(SG) = f_{on}$ **and** $f_{off} \cdot Cov(SG) = 0$ **then**
      return $Cov(SG)$;
    if $f_{on} \cdot Cov(SG) = 0$ **and** $f_{off} \cdot Cov(SG) = f_{off}$ **then**
      return $\neg Cov(SG)$;
  return $\varnothing$;
}

The DC-based node reduction implemented by *FindCover* has the effect that the same node index (variable) may appear more than once on a path going from the root to a terminal node of the resulting FBDD (Definition 4.12). Nevertheless, such an effect has never been observed during the experiments, except for some increase of the circuit depth.

In order to be able to express all the possible forms that *Cov*(*F*) can take, a special node structure has been chosen. This allows the use of complemented edges to indicate the inversion of the function implemented by the sub-graph to which they point. As long as it is not required that the FBDD-based representations of the resulting covers are canonical, both *else* and *then* edges are allowed to be complemented. Special flags indicate whether the function implemented by each of the two child nodes has to be *multiplied* or not with the current node variable, taken with the right polarity. These flags have been introduced to support the optimization of the logic implementation by reducing the gate count per node. Each FBDD node also contains a pointer to the ROBDD-based representation of the function implemented by its sub-graph. In this way, the Boolean functions involved in the DC-based node reduction can be efficiently manipulated. The run-time and the memory consumption of the search associated with the DC-based node reduction can be reduced by limiting the number of investigated nodes (Appendix 1).

The worst case run-time complexity of the FBDD-based logic implementation of an incompletely specified function is proportional to the product of the number of input variables, the maximum size of the ROBDD-based representation of each output and the size of the resulting cover. When the DC-based node reduction is enabled, the square of the resulting cover size has to be taken. The node counts of the resulting covers are usually orders of magnitude smaller than the node counts of the original ROBDDs.

7.3 Experimental Results

The FBDD-based approaches published so far do not target the synthesis of incompletly specified functions. Consequently, the proposed FBDD-based method has been evaluated with respect to SIS [Sen92] and the OBDD-based methods available in the CUDD-package [Cudd] that are able to handle don't cares.

The experimental setup and results are described in Appendix 1 (Table 7.1 – 7.5).

First, the FBDD-based approach has been compared to the *restrict* operator (Definition 4.14). For the other OBDD-based optimization methods from [Cudd] that are able to handle don't cares like *constrain* (Definition 4.13) or *squeeze*, similar results have been obtained as with *restrict*.

The FBDD-based approach outperforms the *restrict*-based approach with respect to the node count and the number of logic operators in the resulting circuit descriptions at the cost of a run-time increase. The run-time requirement of the proposed approach can be significantly reduced by decreasing the searching space associated with the DC-based node reduction. On the other hand, enlarging this searching space will further improve the compaction of the resulting FBDD-like cover. The size of the searching space associated with the DC-based node reduction can be controlled with the help of several thresholds described in Appendix 2.

Subsequently, the FBDD-based method has been compared with several OBDD-based approaches that use combinations of the *restrict* operator and variable reordering. The variable reordering has been applied to all OBDDs corresponding to each output of the target function. As a result, all the covers obtained with the OBDD-based approach have the same variable ordering and, consequently, a maximized probability of node sharing among them. Variable reordering improves the operator count at the cost of a significant increase in the run-time.

The results of this comparison prove that the proposed FBDD-based method outperforms all the investigated OBDD-based approaches. Running the FBDD-based flow with the DC-based node reduction switched off results in operator counts that are between two and four times better than those obtained with the best investigated ROBDD-based approach. The FBDD-based approach with DC-based node reduction disabled also provides the implementations with the smallest depths. The operator count of the FBDD-based covers can be further improved by enabling the DC-based node reduction and increasing the associated searching space. In this way, one can obtain tradeoffs between the size of the resulting covers and the required run-time.

The resulting circuit descriptions have been synthesized with Synopsys Design Compiler and using a proprietary library. Compared to the best investigated ROBDD-based approach, the FBDD-based flow with the DC-based node reduction disabled reduces the area figures by a factor between two and three. This improvement has been achieved by using shorter run-times as compared to all ROBDD-based approaches. Moreover, the run-time of this simple configuration of the FBDD-based approach is by at least one order of magnitude shorter than the run-time of the ROBDD-based approach with the best logic area results. The area results of the FBDD-based approach can be further improved by enabling the DC-based node reduction.

In the end, the FBDD-based approach has been compared to SIS [Sen92] with respect to the implementation of single-output incompletely specified functions with large DC-sets. It is obvious that the FBDD-based method scales better and improves dramatically the number of gates and area (between 2 and 19 times). This suggests that the proposed FBDD-based approach enables a much better use of the *don’t cares* which in the descriptions of SIS and MIS are referred to as *external don’t cares* [Bra87][Sen92].

7.4 Conclusion

A new BDD-based logic synthesis procedure for irregular and incompletely specified functions with large DC-sets has been presented, which can help to find efficient multi-level implementations. The problem is reduced to the construction of a minimal FBDD by performing DC-based node reduction and mainly by partitioning the definition space of the target function into a reduced number of subspaces, which may be mapped either to ‘0’ or to ‘1’. Heuristics are used to find near-optimal partitions of the definition space into such subspaces and, consequently, to minimize the path and node count of the resulting FBDD-like covers. Furthermore, this approach is also able to use the DC-set to reduce the number of logic operators (i.e. gates) appearing in the circuit description of the non-terminal nodes.

Applying this approach to the synthesis of some benchmark bit-flipping functions [Ghe04] resulted in covers whose circuit descriptions contained about 70% less logic operators than the implementations obtained with the methods available in the CUDD-package (*restrict* operator and variable reordering) [Cudd]. The synthesis of the resulting circuit descriptions with Synopsys Design Compiler revealed that the FBDD-based approach improves the area figures by a factor between two and three, while the run-time consumption is significantly reduced. Moreover, the proposed method scales better and succeeds to get a better advantage of the DC-set than SIS.

A tool that implements a version of the approach presented here can be downloaded from [Fbdd].

Chapter 8

Conclusions

8.1 Summary

This work presents and details the development of the first scalable *deterministic logic built-in self-test* (DLBIST) approach. The implemented scheme is based on the STUMPS architecture (Figure 3.3) and it relies on a pattern generator that can achieve very high fault coverage. The particularity of this pattern generator is a combinational module that implements a so-called bit-flipping function (BFF). The BFF maps deterministic test cubes to a pseudo-random test sequence generated by an LFSR and, optionally, a phase shifter. Finding an efficient pattern (cube) mapping with low hardware overhead is a challenging task. The contribution of this work is a scalable solution for both the pattern mapping problem and the logic synthesis of the resulting BFF that describes this mapping.

This work starts with a short presentation of three of the basic fault models used to describe the defects which can appear during the manufacturing process of integrated circuits and with an introduction in the field of built-in self-test. An overview of the state-of-the-art methods that can be used for the logic synthesis of incompletely specified Boolean functions is also given.

A new pattern mapping algorithm has been proposed for bit-flipping and more generally for *test set embedding* DLBIST schemes. The new mapping method exploits the maneuverability and the compactness of the BDD-based function representation. Evaluations performed in the case of stuck-at fault testing have revealed that both run-time and memory requirements are improved by several orders of magnitude as compared to the original cube-based approach. Moreover, the proposed generation and implementation of the BFF does not require more run-time and memory resources than the ATPG or the fault simulation steps. This efficiency gain can be used to obtain even better solutions in terms of logic overhead and fault coverage.

For the first time, the effectiveness of the embedded test sequences obtained by mapping deterministic test cubes to pseudo-random test sequences has been evaluated with respect to the coverage of non-target defects. The resistive bridging fault model has been used to model non-target defects. The experimental results reveal that both deterministic test cubes and pseudo-random test sequences are useful for detecting non-target defects. Furthermore, it has been shown that increasing the length of the test sequences enhances their non-target defect coverage and significantly reduces the

logic overhead. This increases the appeal of the proposed DLBIST scheme and reduces the need for expensive ATEs.

An extension of the bit-flipping DLBIST approach for transition fault testing has been also presented. This is the first time when a DLBIST scheme is used to test delay faults in circuits with standard scan design. The investigated delay testing approach is based on *functional justification*, but the scheme can also be applied with a minimum modification to an approach based on *scan shifting*. A special combinational module, the *correction logic* (CRL), has been introduced to further improve the test pattern embedding. Due to the rather low random-pattern testability of the transition faults, the saturation of their random fault coverage requires significantly longer test sequences, which in turn is beneficial for both limiting the hardware overhead and improving the coverage of the target and non-target defects.

A new BDD-based logic synthesis procedure for irregular and incompletely specified functions with large DC-sets has been presented, which can help to find efficient multi-level implementations. The problem is reduced to the construction of a minimal FBDD by performing DC-based node reduction and mainly by partitioning the definition space of the target function into a reduced number of subspaces, which may be mapped either to ‘0’ or to ‘1’. Heuristics are used to find near-optimal partitions of the definition space into such subspaces and, consequently, to minimize the path and node count of the resulting FBDD-like covers. Furthermore, this approach is also able to use the DC-set to reduce the number of logic operators (i.e. gates) appearing in the circuit description of the non-terminal nodes.

Despite the fact that this new approach has been developed to optimize the implementations of the BFF and the CRL, the resulting algorithm can be applied for the synthesis of any incompletely specified function that is irregular and has a large DC-set. Among others, examples of such functions are the *bit-fixing function* (BFX) [Tou96] and the *X-making function* (XMF) [Tan04].

Applying this approach to the synthesis of some benchmark bit-flipping functions [Ghe04] resulted in implementations whose circuit descriptions contained about 70% less logic operators than the implementations obtained with the methods available in a state-of-the-art BDD package (*restrict* operator and variable reordering) [Cudd]. The synthesis of the resulting circuit descriptions with Synopsys Design Compiler revealed that the FBDD-based approach improves the area figures by a factor between two and three, while the run-time consumption is significantly reduced. Moreover, the proposed method scales better and succeeds to get a better advantage of the *don’t cares* which in the descriptions of SIS and MIS are referred to as *external don’t cares*.

A tool that implements a version of the approach presented here can be downloaded from [Fbdd].

8.2 Contributions Overview

The main contributions of the research presented in this work are as follows:

- **Scalable Pattern Mapping Approach:** An innovative approach has been introduced for mapping deterministic cubes to a pseudo-random test sequence. This approach relies on the ROBDD-based representation and manipulation of the involved Boolean functions and sets. The used algorithm assigns a pseudo-random pattern to each deterministic cube based on new and efficient mapping cost functions.
- **Evaluation of an Embedded Test Sequence with Respect to the Coverage of Non-modeled Defects:** An analysis has been presented of the coverage of non-modeled defects by pseudo-random sequences in which deterministic cubes have been embedded for the test of stuck-at faults. Resistive bridging faults have been used as a surrogate of non-modeled defects.
- **Evaluation of the Test Length Impact on Hardware Overhead and Defect Coverage:** The impact of the length of the embedded test sequences on the hardware overhead and the coverage of non-modeled defects has been investigated as well.
- **Extension of the bit-flipping DLBIST for Transition Fault Testing:** An extension of the *bit-flipping* DLBIST scheme for transition fault testing has been described. In order to improve pattern embedding, the bit-flipping scheme has been extended with a combinational logic module called *correction logic*. Possible tradeoffs between test length, hardware overhead and final transition fault coverage have been presented.
- **Innovative FBDD-Based Logic Synthesis Approach:** An important achievement of this work is a logic synthesis tool, which is used to improve the implementation of the BFF. In general, this tool is especially suited for the logic implementations of *irregular* functions that have large *don't care* sets. For such functions (e.g. BFF, BFX [Tou96] and XMF [Tan04]), FBDD-like covers are obtained and used as multi-level logic implementations.

The correspondence between these contributions and the chapters of the manuscript is given in Table 8.1.

Work Contributions	Manuscript Structure
Scalable Pattern Mapping Approach	Chapter 5
Evaluation of an Embedded Test Sequence with Respect to the Coverage of Non-modeled Defects	
Evaluation of the Test Length Impact on Hardware Overhead and Defect Coverage	
Extension of the bit-flipping DLBIST for Transition Fault Testing	Chapter 6
Innovative FBDD-Based Logic Synthesis Approach	Chapter 7

Table 8.1: Contributions of the work mapped to the structure of the manuscript.

8.3 Future Work

The proposed DLBIST scheme has been investigated only with respect to the transition fault testing based on *functional justification* (Chapter 6). Nevertheless, in some cases [Sav94] the *scan shifting* approach may ensure a better random testability of the transition faults and, consequently, a lower hardware overhead for the same final fault coverage.

Transition fault testing based on scan shifting can be done in parallel to stuck-at fault testing without affecting the diagnosis capability. In the case of transition fault testing based on functional justification, the diagnosis complexity is significantly increased if the investigated circuits are not guaranteed to pass the stuck-at fault test, at least for the initialization patterns used for transition fault testing. The problem here is that such a guarantee is expensive in the context of deterministic logic BIST.

Once an appropriate ATPG tool will be available, the proposed DLBIST scheme should be evaluated also for the test of path delay faults especially of the critical paths. A combination of critical path-delay tests and transition tests provides an adequate at-speed testing [Bus00].

The test sequence generated by the DLBIST scheme introduced here cannot be modified anymore, once the target CUT together with dedicated test hardware have been cast in silicon. Consequently, it would be interesting to combine this method with other approaches that retrieve the test information from on-chip memory or ATE. In this way, the scheme introduced here becomes more flexible and also the memory and bandwidth requirements of the *on-top* method may be significantly reduced.

Another extension of the work presented here is to develop a new data compression method for deterministic test cubes, in which, instead of encoding directly deterministic patterns, bit-flipping and reseeding [Hel92] information is stored and compressed. This method would work especially well when the *don't care* ratio in the embedded deterministic test cubes is sufficiently large, such that the encoded

information can be efficiently stored on-chip or on a cheap ATE. A first step in this direction is described in [Hak05].

The power consumption of any at-speed BIST-based approach can exceed the power rating of the chip, due to the high signal activity that random test patterns cause in some circuits. Both peak and average power for the presented DLBIST scheme should be analyzed and, if necessary, corrected.

References

[Abr90] M. Abramovici, M.A. Breuer, A.D. Friedman “Digital Systems Testing and Testable Design,” *New York: Computer Science Press* (W. H. Freeman and Co.), 1990.

[Agr81] V.K. Agrawal, E. Cerny “Store and Generate Built-In Testing Approach,” *IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1981, pp. 35-40.

[Akr78] S.B. Akers “Binary Decision Diagrams,” *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 509-516.

[Bal04] K.J. Balakrishnan, N.A. Touba “Improving Encoding Efficiency for Linear Decompressors Using Scan Inversion,” *IEEE International Test Conference (ITC)*, 2004, pp. 936-944.

[Bar82] H. Bardell, W.H. McAnney “Self-testing of Multi-chip Logic Modules,” *IEEE International Test Conference (ITC)*, 1982, pp. 200-204.

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

[Bar90] H. Bardell “Design Considerations for Parallel Pseudo-Random Pattern Generators,” *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 1, No. 1, 1990, pp. 73-87.

[Bas89] R.W. Bassett “Low Cost Testing of High Density Logic Components,” *IEEE International Test Conference (ITC)*, 1989, pp. 550-557.

[Bec92] B. Becker “Synthesis for Testability: Binary Decision Diagrams,” *Springer Verlag, STACS. LNCS*, Vol. 577, 1992, pp. 501-512.

[Bec95] B. Becker, R. Dreschler, R. Werchner “On the Relation Between BDDs and FDDs,” *Information and Computation*, Vol. 123, No. 2, December 1995, pp. 185-197.

[Bra87] R.K. Brayton, R. Rudell, A.L. Sangiovanni-Vincentelli, A. Wang “MIS: a Multiple-Level Logic Optimization System,” *IEEE Transactions on CAD*, November 1987, pp. 1062-1081.

[Bra97] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli “Logic Minimization Algorithms for VLSI Synthesis,” *Kluver Academic Publishers*, 1997.

[Brg84] F. Brglez “On Testability Analysis of Combinational Networks,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1984, pp. 221-225.

[Brg89] F. Brglez, C. Gloster, G. Kedem “Hardware-Based Weighted Random Pattern Generation for Boundary Scan,” *IEEE International Test Conference (ITC)*, 1989, pp. 264-274.

[Bry86] R.E. Bryant “Graph-Based Algorithms for Boolean Function Manipulation,” *IEEE Transactions on Computers*, Vol. C-35, No. 8, 1986, pp. 677-691.

[Bry91] R.E. Bryant “On the Complexity of VLSI Implementations and Graph Repre-

presentations of Boolean Functions with Application to Integer Multiplication," *IEEE Transactions on Computers*, Vol. 40, No. 2, 1991, pp. 205-213.

[Bus00] M.L. Bushnell, V.D. Agrawal "Essentials of Electronic Testing," *Kluwer Academic Publishers*, 2000.

[Cat96] K. Cattel, J.C. Muzio "Synthesis of One-Dimensional Linear Hybrid Cellular Automata," *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 15, No. 3, March 1996, pp. 325-335.

[Cha00] K. Chakrabarty, S. Swaminathan "Built-In Self Testing of High-Performance Circuits Using Twisted-Ring Counters," *IEEE International Symposium on Circuits and Systems*, 2000, pp. 72-76.

[Cha01] A. Chandra, K. Chakrabarty "Frequency-Directed Run Length (FDR) Codes with Application to System-On-A-Chip Test Data Compression," *VLSI Test Symposium (VTS)*, 2001, pp. 42-47.

[Cha94] S.-C. Chang, D.I. Cheng, M. Marek-Sadowska "Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions," *IEEE European Design and Test Conference (EDAC)*, 1994, pp. 620-624.

[Cha96] J.T.-Y. Chang, E.J. McCluskey "Detecting Delay Flaws by Very-Low-Voltage Testing," *IEEE International Test Conference (ITC)*, 1996, pp. 367-376.

[Che88] C. L. Chen "Exhaustive Test Pattern Generation Using Cyclic Codes," *IEEE Transactions on Computers*, Vol. 37, No. 2, February 1988, pp. 225 -228.

[Che96] C.A. Chen, S.K. Gupta "BIST Test Pattern Generators for Two-Pattern Testing-Theory and Design Algorithms," *IEEE Transactions on Computers*, Vol. 45, No.3, 1996, pp. 257-269.

[Coc98] B. F. Cockburn, A. L.-C. Kwong "Transition Maximation Techniques for Enhancing the Two-Pattern Fault Coverage of Pseudorandom Test Pattern Generators," *IEEE VLSI Test Symposium (VTS)*, Monterey, CA, 1998, pp. 430-439.

[Cou89] O. Coudert, C. Berthet, J. C. Madre "Verification of Sequential Machines Using Boolean Functional Vectors," *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 111-128.

[Cou90] O. Coudert, J.C. Madre "Verification of Synchronous Sequential Machines Based on Symbolic Execution," *Automatic Verification Methods for Finite State Systems, Springer-Verlag*, 1990, pp. 365-373.

[Cudd] http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[Dac90] W. Dachn, T.W. Williams, K.D. Wagner "Aliasing Errors in Linear Automata Used as Multiple-input Signature Analyzers," *IBM Journal of Research and Development*, Vol. 34, No. 2/3, 1990, pp 363-380.

[Dam89]M. Damiani, P. Olivo, M. Favalli, B. Ricco "An Analytical Model for the Aliasing Probability in Signature Analysis Testing," *IEEE Transactions on CAD*, Vol. 8, No. 11, 1989, pp. 1133-1144.

[Dre94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, M.A. Perkowski "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams," *31st ACM/IEEE Design Automation Conference*, 1994, pp. 415-419.

[Dre98] R. Drechsler, B. Becker "Binary Decision Diagrams Theory and Implementation," *Kluver Academic Publishers*, Dordrecht, 1998.

[Dr98] R. Drechsler, B. Becker "Ordered Kronecker Functional Decision Diagrams – A Data Structure for Representation and Manipulation of Boolean Functions," *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol.

17, No. 10, October 1998, pp. 965-973.

[Duf97] C. Dufaza, Y. Zorian “On the Generation of Pseudo-deterministic Two-patterns Test Sequence with LFSRs,” *IEEE European Design and Test Conference (EDAC)*, 1997, pp. 69-76.

[Eic83] B. Eichelberger, E. Lindbloom “Random Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test,” *IBM Journal of Research and Development*, Vol. 27, No. 3, May 1983, pp. 265-272.

[Eng03] P. Engelke, I. Polian, M. Renovell, B. Becker “Simulating Resistive Bridging and Stuck-at Faults,” *IEEE International Test Conference (ITC)*, 2003, pp. 1051-1059.

[Eng04] P. Engelke, I. Polian, M. Renovell, B. Becker “Automatic Test Pattern Generation for Resistive Bridging Faults,” *IEEE European Test Symposium (ETS)*, 2004, pp. 160-165.

[Eng05] P. Engelke, V. Gherman, I. Polian, Y. Tang, H.-J. Wunderlich, B. Becker “Sequence Length, Area Cost and Non-Target Defect Coverage Tradeoffs in Deterministic Logic BIST,” *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2005, pp. 11-18.

[Fbdd] www.ra.informatik.uni-stuttgart.de/~ghermanv/benchmarks/index.phtml

[Fel93] E. Felt, G. York, R. Brayton, A. Sangiovanni-Vincentelli “Dynamic Variable Reordering for BDD Minimization,” *IEEE European Design Automation Conference*, September 1993, pp. 130-135.

[Fur91] K. Furuya, E.J. McCluskey “Two-Pattern Test Capabilities of Autonomous TPG Circuits,” *IEEE International Test Conference (ITC)*, 1991, pp. 704-711.

[Geu00] M.J. Geuzebroek, J.Th. van der Linden, A.J. van de Goor “Test Point Insertion for Compact Test Sets,” *IEEE International Test Conference (ITC)*, 2000, pp. 506-514.

[Ger96] J. Gergov, C. Meinel “Mod-2-OBDDs. A Generalization of OBDDs and ExOR-Sum-Of-Products,” *Formal Methods in System Design*, *Kluwer*, Vol. 8, No. 3, 1996, pp. 273-282.

[Ghe04] V. Gherman, H.-J. Wunderlich, H. Vranken, F. Hapke, M. Wittke, M. Garbers “Efficient Pattern Mapping for Deterministic Logic BIST,” *IEEE International Test Conference (ITC)*, 2004, pp. 48-56.

[Ghe05] V. Gherman, H.-J. Wunderlich, M. Garbers, J. Schlöffel “DLBIST for Delay Testing,” *17th ITG/GI/GMM Workshop “Testmethoden und Zuverlässigkeit von Schaltungen und Systemen,”* Innsbruck, 2005, pp. 39-43.

[Gir97] P. Girard, C. Landrault, V. Moreda, S. Pravossoudovitch “An Optimized BIST Test Pattern Generator for Delay Testing,” *IEEE VLSI Test Symposium (VTS)*, 1997, pp. 94-100.

[Gol82] S.W. Golomb “Shift Register Sequences,” *Aegan Park Press*, Laguna Hills, 1982.

[Gon02] P. Gonciari, B. Al-Hashimi, N. Nicolici “Improving Compression Ratio, Area Overhead, and Test Application Time for System-On-A-Chip Test Data Compression/Decompression,” *IEEE Design, Automation and Test in Europe (DATE)*, 2002, pp 604-611.

[Gue99] W. Günther, R. Drechsler “Minimization of Free BDDs,” *ASP Design Automation Conference*, 1999, pp. 323-326.

[Gue00] W. Günther “Minimization of Free BDDs Using Evolutionary Techniques,” *International Workshop on Logic Synthesis*, Dana Point, CA, May, 2000.

[Gup96] S. Gupta, J. Rajski, J. Tyszer “Arithmetic Adaptive Generators of Pseudo-Exhaustive Test Patterns,” *IEEE Transactions on Computers*, Vol. 8, No. 45, 1996, pp. 939-949.

[Hak05] A. W. Hakmi, V. Gherman, H.-J. Wunderlich, M. Garbers, J. Schlöffel “Implementing a Scheme for External Deterministic Self-Test,” *IEEE VLSI Test Symposium (VTS)*, 2005, pp. 101-106.

[Hay74] J.P. Hayes, A.D. Friedman “Test Point Placement to Simplify Fault Detection,” *IEEE Transactions on Computers*, Vol. C-33, July 1974, pp. 727-735.

[Hel90] S. Hellebrand, H.-J. Wunderlich, O.F. Haberl “Generating Pseudo-Exhaustive Vectors for External Testing,” *IEEE International Test Conference (ITC)*, 1990, pp. 670-679.

[Hel92] S. Hellebrand, S. Tarnick, J. Rajski, B. Courtois “Generation of Vector Patterns through Reseeding of Multiple-Polynomial Linear Feedback Shift Registers,” *IEEE International Test Conference (ITC)*, 1992, pp. 120-129.

[Hel95] S. Hellebrand, B. Reeb, S. Tarnick, H.-J. Wunderlich “Pattern Generation for a Deterministic BIST Scheme,” *ACM/IEEE International Conference on CAD-95 (ICCAD95)*, 1995, pp. 88-94.

[Hel96] S. Hellebrand, H.-J. Wunderlich, A. Hertwig “Mixed-Mode BIST Using Embedded Processors,” *IEEE International Test Conference (ITC)*, 1996, pp. 195-204.

[Her96] K. Heragu, J.H. Patel, V.D. Agrawal “Segment Delay Faults: a New Fault Model,” *IEEE VLSI Test Symposium (VTS)*, 1996, pp. 32-39.

[Het99] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, J. Rajski “Logic BIST for Large Industrial Designs: Real Issues and Case Studies,” *IEEE International Test Conference (ITC)*, 1999, pp. 358-367.

[Hon97] Y. Hong, P. Beerel, J. Burch, K. McMillan “Safe BDD Minimization Using Don’t Cares,” *ACM/IEEE Design Automation Conference*, 1997, pp. 208-213.

[Hon00] Y. Hong, P. Beerel, J. Burch, K. McMillan “Sibling-Substitution-Based BDD Minimization Using Don’t Cares,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2000, pp. 44-55.

[Hua03] L. Huaguo “A New Technique for Deterministic Scan-Based Built-In Self-Test (BIST),” *Ph.D. Thesis, Shaker Verlag*, 2003.

[Hsu01] F.F. Hsu, K.M. Butler, J.H. Patel “A Case Study on the Implementation of the Illinois Scan Architecture,” *IEEE International Test Conference (ITC)*, 2001, pp. 538-547.

[Hug84] J.L.A. Hughes, E.J. McCluskey “An Analysis of the Multiple Fault Detection Capabilities of Single Stuck-At Fault Test Sets,” *IEEE International Test Conference (ITC)*, 1984, pp. 52-58.

[Iba75] O.H. Ibara, S. Sahmi “Polynomial Complete Fault Detection Problems,” *IEEE Transactions on Computers*, Vol. C-24, No. 3, 1975, pp. 242-249.

[Ish91] N. Ishiura, H. Sawada, S. Yajima “Minimization of Binary Decision Diagrams Based on Exchange of Variables,” *IEEE International Conference on CAD (ICCAD)*, November 1991, pp. 472-475.

[Iye90] V.S. Iyengar, B.K. Rosen, J.A. Waicukauski “On Computing the Sizes of Detected Delay Faults,” *IEEE Transactions on CAD*, Vol. 9, No. 3, 1990, pp. 299-312.

[Jou95] *Journal of Electronic Testing: Theory and Applications (JETTA): Special Issue on Partial Scan Methods*, Vol. 7, August/October, 1995.

[Kaj95] S. Kajihara, I. Pomeranz, K. Kinoshita, S.M. Reddy “Cost-Effective Generation of Minimal Test Sets for Stuck-At Faults in Combinatorial Logic Circuits,” *IEEE Transactions on Computers*, Vol. 14, 1995, pp. 1496-1504.

[Kar98] R. Karri, N. Mukherjee “Versatile BIST: an Integrated Approach to ON-Line /Off-Line BIST,” *IEEE International Test Conference (ITC)*, 1998, pp. 910-917.

[Keb92] U. Kebschull, E. Schubert, W. Rosenstiel “Multi-Level Logic Synthesis Based on Functional Decision Diagrams,” *European Conference on Design Automation (EDAC)*, 1992, pp. 43-47.

[Keb93] U. Kebschull, W. Rosenstiel “Efficient Graph-Based Computation of and Manipulation of Functional Decision Diagrams,” *European Conference on Design Automation (EDAC)*, 1993, pp. 278-282.

[Kei99] M. Keim, I. Polian, H. Hengster, B. Becker “A Scalable BIST Architecture for Delay Faults,” *IEEE European Test Workshop (ETW)*, 1999, pp. 98-103.

[Kha87] M. Khare, A. Albicki “Cellular Automata Used for Test Pattern Generation,” *IEEE International Conference on Computer Design (ICCD)*, 1987, pp. 56-59.

[Kie97] G. Kiefer, H.-J. Wunderlich “Using BIST Control for Pattern Generation,” *IEEE International Test Conference (ITC)*, 1997, pp. 347-355.

[Kie98] G. Kiefer, H.-J. Wunderlich “Deterministic BIST with Multiple Scan Chains,” *IEEE International Test Conference (ITC)*, 1998, pp. 1057-1064.

[Kie00] G. Kiefer, H. Vranken, E. J. Marinissen, H.-J. Wunderlich “Application of Deterministic Logic BIST on Industrial Circuits,” *IEEE International Test Conference (ITC)*, 2000, pp. 105-114.

[Koe79] B. Koenemann, J. Mucha, G. Zwiehoff “Built-In Logic Block Observation Techniques,” *IEEE International Test Conference (ITC)*, Cherry Hill, NJ, 1979, pp. 37-41.

[Koe91] B. Koenemann “LFSR-Coded Test Patterns for Scan Designs,” *IEEE European Test Conference (ETC)*, 1991, pp. 237-242.

[Koe01] B. Koenemann, C. Barnhart, B. Keller, T. Snethen, O. Farnsworth, D. Weather “A SmartBIST Variant with Guaranteed Encoding,” *IEEE Asian Test Symposium (ATS)*, 2001, pp. 325-300.

[Kra89] A. Krasniewski, S. Pilarski “Circular Self-Test Path: a Low Cost BIST Technique of VLSI Circuits,” *IEEE Transactions on CAD*, January 1989, pp. 46-55.

[Krs98] A. Krstic, K.T. Cheng “Delay Fault Testing for VLSI Circuits,” *Boston: Kluver Academic Publishers*, 1998.

[Lai04] L. Lai, J.H. Patel, T. Rinderknecht, W.T. Cheng “Logic BIST with Scan Chain Segmentation,” *IEEE International Test Conference (ITC)*, 2004, pp. 57-66.

[Lee59] C.Y. Lee “Representation of Switching Circuits by Binary-Decision Programs,” *Bell System Technical Journal*, Vol. 38, July 1959, pp. 985-999.

[Lee00] C. Lee, D.M. H.Walker “PROBE: a PPSFP Simulator for Resistive Bridging Faults,” *IEEE VLSI Test Symposium (VTS)*, 2000, pp. 105-110.

[Lev86] Y. Levendel, R.P. Menon “Transition Faults in Combinational Circuits: Input Transition Test Generation and Fault Simulation,” *IEEE International Fault-Tolerant Computing Symposium (FTCS)*, 1986, pp. 278-283.

[Li03] W. Li, C. Yu, S.M. Reddy, I. Pomeranz “A Scan BIST Generation Method Using a Markov Source and Partial Bit-fixing,” *IEEE Design Automation Conference (DAC)*, 2003, pp. 554-559.

[Lin87] C.J. Lin, S.M. Reddy “On Delay Fault Testing in Logic Circuits,” *IEEE Transanctions on CAD*, 1987, pp. 694-703.

[Lia02] H. Liang, S. Hellebrand, H.-J. Wunderlich “Two-Dimensional Test Data Compression for Scan-Based Deterministic BIST,” *Journal of Electronic Testing-Theory and Applications (JETTA)*, Vol. 18, No. 2, 2002, pp. 157-168.

[Mal92] Y.K. Malaiya, R. Rajsuman “Bridging Faults and IDDQ Testing,” *IEEE Computer Society Press*, Los Alamitos, California1992.

[Mcc65] E.J. McCluskey “Introduction to the Theory of Switching Circuits,” *McGraw-Hill*, 1965.

[Mcc81] E.J. McCluskey, S. Bozorgui-Nesbat “Design for Autonomous Test,” *IEEE Transactions on Computers*, Vol. 30, No. 11, 1981, pp. 866-875.

[Min97] S. Minato “Binary Decision Diagrams and Applications for VLSI CAD,” *Kluver Academic Publishers*, Dordrecht, 1997.

[Mit04] S. Mitra, K.S. Kim “X-Compact: an Efficient Response Compaction Technique for Test Cost Reduction,” *IEEE Transactions on CAD*, Vol. 23, No. 3, March 2004, pp. 421-432.

[Muk98] N. Mukherjee, T.J. Chakraborty, S. Bhawmik “A BIST Scheme for the Detection of Path-Delay Faults,” *IEEE International Test Conference (ITC)*, 1998, pp. 422-432.

[Oli98] A.L. Oliveira, L.P. Carloni, T. Villa, A.L. Sangiovanni-Vincentelli “Exact Minimization of Binary Decision Diagrams Using Implicit Techniques,” *IEEE Transactions on Computers*, Vol. 47, No. 11, 1998, pp. 1282-1296.

[Pan94] S. Panda, F. Somenzi, B.F. Plessier “Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams,” *IEEE International Conference on CAD (ICCAD)*, San Jose, CA, November 1994, pp. 628-631.

[Raj93] J. Rajski, J. Tyszer “Test Responses Compaction in Accumulators with Rotate Carry Adders,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 12, No. 4, April 1993, pp. 531-539.

[Raj98] J. Rajski, J. Tyszer “Design of Phase Shifters for BIST Applications,” *IEEE VLSI Test Symposium (VTS)*, 1998, pp. 218-224.

[Raj02] J. Rajski, J. Tyszer, M. Kassab, N. Mukherjee, R. Thompson, K.-H. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide, J. Qian “Embedded Deterministic Test for Low Cost Manufacturing Test,” *IEEE International Test Conference (ITC)*, 2002, pp. 301-310.

[Red92] L. N. Reddy, I. Pomeranz, S. M. Reddy “ROTCO: a Reverse Order Test Compaction Technique,” *IEEE EURO-ASIC Conference*, September 1992, pp. 189-194.

[Ren95] M. Renovell, P. Huc, Y. Bertrand “The Concept of Resistance Interval: a New Parametric Model for Resistive Bridging Fault,” *IEEE VLSI Test Symposium (VTS)*, 1995, pp. 184-189.

[Ren99] M. Renovell, F. Azais, Y. Bertrand “Detection of Defects Using Fault Model Oriented Test Sequences,” *Journal of Electronic Testing: Theory and Applications (JETTA)*, Vol. 14, No. 1-2, 1999, pp. 13-22.

[Rud93] R. Rudell “Dynamic Variable Ordering for Ordered Binary Decision Diagrams,” *IEEE International Conference on CAD (ICCAD)*, November 1993, pp. 42–47.

[Sau96] M. Sauerhoff, I. Wegener “On the Complexity of Minimizing the OBDD Size for Incompletely Specified Functions,” *IEEE Transactions on CAD*, Vol. 15, November 1996, pp. 1435-1437.

[Sav84] J. Savir, G.S. Ditlow, P.H. Bardell “Random Pattern Testability,” *IEEE Transactions on Computers*, Vol. C-33, No. 1, 1984, pp. 79-90.

[Sav92] J. Savir “Skewed-Load Transition Test: Part I, Calculus,” *IEEE International Test Conference (ITC)*, 1992, pp.705-713.

[Sav94] J. Savir, S. Patil “Broad-Side Delay Test,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 13, No. 8, 1994, pp. 1057-1064.

[Sch99] C. Scholl, D. Möller, P. Molitor, R. Drechsler “BDD Minimization Using Symmetries,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 2, 1999, pp. 81-100.

[Sei91] B.H. Seiß, P.M. Trouborst, M.H. Schulz “Test Point Insertion for Scan-Based BIST,” *IEEE European Test Conference (ETC)*, April 1991, pp. 253-262.

[Sen92] E. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, A. Sangiovanni-Vincentelli “Sequential Circuit Design Using Synthesis and Optimization,” *IEEE International Conference on Computer Design (ICCD)*, October 1992, pp. 328-333.

[Sha00] M. Sharma, J.H. Patel “Enhanced Delay Defect Coverage with Path-Segments,” *IEEE International Test Conference (ITC), 2000*, pp. 385-392.

[Sha03] M. Sharma, J.H. Patel, J. Rearick “Test Data Compression and Test Time Reduction of Longest-Path-Per-Gate Tests Based on Illinois Scan Architecture,” *IEEE VLSI Test Symposium (VTS)*, 2003, pp. 15-21.

[Shi94] T. Shiple, R. Hojati, A. Sangiovanni-Vicentelli, R. Brayton “Heuristic Minimization of BDDs Using Don’t Cares,” *ACM/IEEE Design Automation Conference*, 1994, pp. 225–231.

[Sie93] D. Sieling, I. Wegener “Reduction of OBDDs in Linear Time,” *Information Processing Letters*, Vol. 48, No. 3, 1993, pp.139-144.

[Sie95] D. Sieling, I. Wegener “Graph Driven BDDs – a New Data Structure for Boolean Functions,” *IEEE Theoretical Computer Science*, Vol. 141, No.1-2, 1995, pp. 283-310.

[Sie99] D. Sieling “The Complexity of Minimizing FBDDs,” *Mathematical Foundations of Computer Science (MFCS)*, 1999, pp. 251-261.

[Smi85] G.L. Smith “Model for Delay Faults Based Upon Paths,” *IEEE International Test Conference (ITC)*, 1985, pp. 342-349.

[Str90] A.P. Stroele, H.-J. Wunderlich “Error Masking in Self-Testable Circuits,” *IEEE International Test Conference (ITC)*, 1990, pp. 544-552.

[Str94] A.P. Stroele, H.-J. Wunderlich “Configuring Flip-Flops to BIST Registers,” *IEEE International Test Conference (ITC)*, 1994, pp. 939-948.

[Tan04] Y. Tang, H.-J. Wunderlich, H. Vranken, F. Hapke, M. Wittke, P. Engelke, I. Polian, B. Becker “X-Masking During Logic BIST and Its Impact on Defect Coverage,” *IEEE International Test Conference (ITC)*, 2004, pp. 442-451.

[Tim83] C. Timoc, M. Buehler, T. Griswold, C. Pina, F. Scott, L. Hess “Logical Models of Physical Failures,” *IEEE International Test Conference (ITC)*, 1983, pp. 546-553.

[Tou96] N.A. Touba, E.J. McCluskey “Altering A Pseudo-Random Bit Sequence for Scan-Based BIST,” *IEEE International Test Conference (ITC)*, 1996, pp. 167-175.

[Tou04] K.J. Balakrishan, N.A. Touba “Relating Entropy Theory to Test Data Compression,” *IEEE European Test Symposium (ETS)*, 2004, pp. 187- 192.

[Tri80] E. Trischler “Incomplete Scan Path with Automatic Test Generation Methodology,” *IEEE International Test Conference (ITC)*, 1980, pp. 153-162.

[Tro91] G. Tromp “Minimal Test Sets for Combinatorial Circuits,” *IEEE International Test Conference (ITC)*, 1991, pp. 204-209.

[Tsa00] K.-H. Tsai, J. Rajski, M. Marek-Sadowska “Star Test: the Theory and Its Applications,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 19, No. 9, September 2000, pp. 1052-1064.

[Tsa97] K.-H. Tsai, S. Hellebrand, J. Rajski, M. Marek-Sadowska “STARBIST: Scan Autocorrelated Random-Pattern Generation,” *ACM/IEEE Design Automation Conference (DAC)*, 1997, pp. 472-478.

[Vra02] H. Vranken, F. Meister, H.-J. Wunderlich “Combining Deterministic Logic BIST with Test Point Insertion,” *IEEE European Test Workshop (ETW)*, May 2002, pp. 389-394.

[Vra04] H. Vranken, H.-J. Wunderlich, F.S. Sapei “Impact of Test Point Insertion on Silicon Area and Timing During Layout,” *IEEE Design, Automation and Test in Europe (DATE)*, February 2004, pp. 810-815.

[Wai87] J.A. Waicukauski, E. Lindbloom, B.K. Rosen, V.S. Iyengar “Transition Fault Simulation,” *IEEE Design and Test of Computers*, Vol. 4, 1987, pp. 32-38.

[Wan86] L.T. Wang, E.J. McCluskey “Concurrent Built-In Logic Block Observer (CBILBO),” *IEEE International Symposium on Circuits and Systems (ISCAS)*, 1986, pp. 1054-1057.

[Wue04] A. Wuertenberger, C.S. Tautermann, S. Hellebrand “Data Compression for Multiple Scan Chain Using Dictionaries with Corrections,” *IEEE International Test Conference (ITC)*, 2004, pp. 926-935.

[Wun85] H.-J. Wunderlich “PROTEST: a Tool for Probabilistic Testability Analysis,” *ACM/IEEE Design Automation Conference (DAC)*, 1985, pp. 204-211.

[Wun88] H.-J. Wunderlich “Multiple Distributions for Biased Random Test Patterns,” *IEEE International Test Conference (ITC)*, 1988, pp. 236-244.

[Wun90] H.-J. Wunderlich “Multiple Distributions for Biased Random Test Patterns,” *IEEE Transactions on CAD*, Vol. 9, No. 6, June 1990, pp. 594-602.

[Wun96] H.-J. Wunderlich, G. Kiefer “Bit-Flipping BIST,” *IEEE International Conference on CAD (ICCAD)*, 1996, pp. 337-343.

[Wu98] H.-J. Wunderlich, R. Dorsch “Accumulator Based Deterministic BIST,” *IEEE International Test Conference (ITC)*, Washington D.C., 1998, pp. 412-421.

[Wun98] H.-J. Wunderlich “BIST for Systems-On-A-Chip,” *INTEGRATION, the VLSI Journal*, 1998, pp. 55-78.

[Wun02] H.-J. Wunderlich “Design and Test of System-on-a-Chip,” *Lecture Notes*, University of Stuttgart, 2002.

[Wur95] B. Wurth, K. Fuchs “A BIST Approach to Delay Fault Testing with Reduced Test Length,” *IEEE European Design and Test Conference (EDAC)*, 1995, pp. 418-423.

[Xia03] D. Xiang, S. Gu, J.G. Sun, Y. Wu “A Cost-Effective Scan Architecture for Scan Testing with Non-Scan Test Power and Test Application Cost,” *IEEE Design Automation Conference (DAC)*, June 2003, pp. 744-747.

[Zor90] Y. Zorian, V.K. Agarval “Optimizing Error Masking in BIST by Output Data Modification,” *Journal of Electronic Testing Theory and Applications (JETTA)*, Vol. 1, No. 1, February 1990, pp. 59-72.

Index

Appendix 1 – Tables with Experimental Results

The first experiments considered here refer to the evaluation of the new BDD-based pattern mapping approach presented in Chapter 5 with respect to the original cube-based approach [Wun96]. The experimental results (Table 5.5 – 5.11) have been obtained using GNU Linux machines equipped with 1 GB of memory and an AMD Athlon-XP processor running at 1.5 GHz. The BDD-based computations have been implemented using the CUDD-package [Cudd].

Table 5.5 presents the characteristics of the industrial designs that have been used as benchmark circuits. The first column reports the circuit name encoded as *pN*, where *N* denotes the number of nets in the design. The second column gives the number of scan flip-flops contained in each circuit. The last two columns report the stuck-at fault coverage and efficiency (Definition 2.1 – 2.2) achieved after applying 10,000 pseudo-random test patterns generated by a 32-stages long LFSR with a primitive polynomial.

Deterministic test cubes generated with an industrial ATPG tool (AMSAL[10]) have been embedded into the pseudo-random test sequences using the original cube-based and the new BDD-based mapping approaches.

During the generation of the BDD-based representations of the resulting BFFs, no static or dynamic variable reordering has been performed. The variables have been a priori and optimally arranged in groups corresponding to the state bits of the LFSR, the pattern counter and the shift counter. No phase shifter has been used. The experiments have been performed with the same variable order for all the designs.

In Table 5.6, the BDD-based and the cube-based mapping approaches have been compared with respect to the run-time requirements of the pattern mapping, ATPG and fault simulation tasks. The BDD-based approach reduces the pattern mapping time from several days down to a few minutes. The run-times of the two other tasks, ATPG and fault simulation, are considerably improved as well.

Design	# Flip-flops	Random stuck-at fault coverage [%]	Random stuck-at fault efficiency [%]
p19k	1,407	63.11	69.03
p59k	4,730	87.30	97.00
p127k	5,116	82.14	83.96
p278k	9,967	79.92	81.29
p333k	20,756	93.64	95.57
p951k	104,624	92.91	92.56
p2074k	58,835	64.11	92.54

Table 5.5: Benchmark designs characteristics with respect to stuck-at fault testing.

[10] Automatic Multi restartable Scan test pattern generation And Localization of faults.

Design	Cube-based approach			BDD-based approach		
	Mapping time [h:m]	ATPG time [h:m]	Fault simulation time [h:m]	Mapping time [h:m]	ATPG time [h:m]	Fault simulation time [h:m]
p19k	02:57	00:00	00:33	00:02	00:00	00:01
p59k	02:20	00:05	00:30	00:02	00:01	00:03
p127k	76:54	02:22	18:25	00:14	03:10	00:12
p278k	193:10	05:20	37:23	00:09	02:29	00:22
p333k	116:15	00:48	47:45	00:14	00:37	00:17
p951k	-	-	-	03:12	01:14	00:57
p2074k	-	-	-	03:59	02:55	00:35

Table 5.6: Run-time for different tasks of the cube-based and BDD-based algorithms. For the design *p2074k* a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz has been used.

The overall run-time (*Time*) and the memory (*Memory*) consumption (including also the run-time and the memory required for the BDD-based logic optimization) are quoted in Table 5.7. The BDD-based approach is able to reduce the total run-time from more than a week down to several hours, while also the memory requirements scale quite well with the circuit size.

The fault efficiencies and the cell area overhead obtained with both mapping approaches are reported in Table 5.8. In order to have comparable experimental results, the fault efficiency of the BDD-based approach has been limited to the maximum reachable with the cube-based approach. By spending more resources, even higher fault efficiency could be achieved. The only limitation is represented by the resources given to the ATPG tool. The last column (C*ell area*) shows the cell area overhead of the BFF implementation relative to the cell area of the CUT, obtained using Synopsys Design Compiler and a proprietary library. Only the logic overhead of the BFF implementation is given. The overhead of the other parts of the DLBIST hardware is relatively small and it may be neglected.

Design	Cube-based approach		BDD-based approach	
	Time [h:m]	Memory [MB]	Time [h:m]	Memory [MB]
p19k	03:30	58	00:27	58
p59k	02:55	138	00:11	66
p127k	97:41	368	11:13	211
p278k	235:53	584	15:21	318
p333k	164:48	660	09:07	290
p951k	-	-	14:22	1106
p2074k	-	-	18:37	1865

Table 5.7: Run-time and memory consumption of the cube-based and BDD-based algorithms. For the design *p2074k* a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz has been used.

Design	Cube-based approach		BDD-based approach	
	Stuck-at fault efficiency [%]	Cell area [%]	Stuck-at fault efficiency [%]	Cell area [%]
p19k	96.57	89.67	97.46	21.71
p59k	98.95	7.64	99.05	3.59
p127k	94.56	27.86	95.47	9.81
p278k	90.67	25.77	91.47	9.66
p333k	97.41	12.07	97.47	3.56
p951k	-	-	99.65	1.49
p2074k	-	-	98.97	2.64

Table 5.8: Fault efficiency and logic overhead of the cube-based and BDD-based algorithms.

Due to excessive run-time and memory requirements, no experimental results are available for the cube-based approach in the case of the 2 largest designs in Table 5.6, 5.7 and 5.8.

Table 5.9 illustrates how the new pattern mapping approach scales when the target fault efficiency is increased to the highest levels allowed by the ATPG tool. Most of the additional run-time is consumed during the deterministic pattern generation and the BDD-based logic synthesis of the BFF, while the time spent for fault simulation remains constant. These final fault efficiencies are practically not reachable by the cube-based approach in the case of the largest five designs. The presented approach does not only scale very well in terms of run-time and memory consumption, but also in terms of fault efficiency and area overhead. Additionally, it is shown that the logic overhead decreases considerably in the case of the largest designs.

Below, it is shown how the new mapping approach performs on smaller, but still difficult to test designs (Table 5.10 – 5.11). For this purpose, the ISCAS-85 and the combinational part of the ISCAS-89 benchmarks [Brg89][Wun96] have been used. The two benchmark suites are identified with the symbols “c” and “cs”, respectively.

Design	# Embedded patterns	Fault efficiency [%]	Time [h:m]	Memory [MB]	Cell area [%]
p19k	181	99.19	00:32	91	25.36
p59k	137	99.10	00:11	68	3.75
p127k	582	99.26	18:20	295	21.81
p278k	1,549	98.87	55:37	536	34.58
p333k	1,298	99.30	23:00	359	7.00
p951k	259	99.65	14:22	1,106	1.49
p2074k	302	98.97	18:37	1,865	2.64

Table 5.9: Results obtained with the BDD-based approach targeting the fault efficiency allowed by the ATPG tool. For the designs *p278k* and *p2074k* a machine equipped with 2 GB of memory and an Intel Pentium 4 CPU running at 2.4 GHz has been used.

Table 5.10 presents the number of scan flip-flops contained in each circuit and the stuck-at fault efficiency (Definition 2.2) obtained after applying 10,000 pseudo-random patterns generated by a 13-stages long LFSR with a primitive polynomial. Only those ISCAS benchmarks which still have undetected non-redundant stuck-at faults after applying 10,000 pseudo-random patterns are analyzed.

In Table 5.11, a comparison is presented between the BDD-based and the cube-based approaches with respect to the mentioned ISCAS designs. In most of these experiments, it has not been possible to achieve 100% final fault efficiency, due to the fact that the available ATPG tool was especially adapted for large industrial designs, where it is not relevant whether a few faults *expensive* to detect remain undetected.

For all the designs it has been possible to reach higher final fault efficiencies with the BDD-based approach. This is due to a loss of pseudo-randomly testable faults after some iterations of the cube-based approach, which could not be recovered by the available ATPG tool. With the exception of 2 small ISCAS designs (*cs641, cs713*), which have been completed in a few seconds, the total run-time of the BDD-based approach is much shorter, sometimes by even one order of magnitude. For the larger ISCAS designs the difference between the two approaches is more obvious. This proves the better scalability of the BDD-based algorithm, just like the experimental results for the large industrial designs. Furthermore, with the exception of a few ISCAS designs, the memory consumption (*cs641, cs713, cs838*) and the logic area (*cs641, cs5378*) are lower for the BDD-based algorithm.

Tables 5.12 – 5.13 present the experimental results of an investigation of the non-target defect coverage of the embedded test sequences obtained with the bit-flipping DLBIST scheme. Resistive bridging faults are used as a surrogate of non-target defects [Eng05]. The same types of machines have been utilized as for the experiments considered before.

Design	**Size [FFs]**	**Random fault efficiency[%]**
c2670	221	91.77
c7552	313	97.11
cs641	78	98.01
cs713	77	98.16
cs838	67	69.19
cs5378	263	97.44
cs9234	286	87.75
cs13207	852	91.69
cs15850	761	94.48
cs38417	1,770	92.22
cs38584	1,768	98.05

Table 5.10: Characteristics of the ISCAS (85 and 89) benchmark designs.

Design	Cube-based approach				BDD-based approach			
	Final fault efficiency [%]	Time [m:s]	Memory [MB]	Cell area [μm^2]	Final fault efficiency [%]	Time [m:s]	Memory [MB]	Cell area [μm^2]
c2670	99.00	07:30	10	2,213	99.67	06:55	5	852
c7552	99.73	03:59	14	5,694	99.87	01:16	6	3,272
cs641	99.65	00:03	4	73	99.65	00:04	4	101
cs713	99.78	00:03	4	87	99:78	00:03	4	79
cs838	97:13	00:13	4	1,631	98.35	00:06	4	1,362
cs5378	99.71	01:21	14	1,043	99.94	00:12	5	1,065
cs9234	98.89	69:18	17	8,575	99.40	31:59	6	5,578
cs13207	99.25	29:06	46	4,740	99.74	01:25	8	3,226
cs15850	99.88	11:23	27	11,235	100.00	00:35	9	6,259
cs38417	99.87	557:48	113	56,338	99.99	46:24	15	25,534
cs38584	99.95	94:37	88	8,696	99.99	02:29	14	4,769

Table 5.11: Comparison of the two approaches on some ISCAS (85 and 89) designs.

Sequences of 1K, 5K and 10K test patterns have been considered. The stuck-at fault coverage (Definition 2.1) achieved by the pseudo-random test sequences (before deterministic cube embedding) is reported in Table 5.12. The pseudo-random test sequences have been generated by a 13-stages long LFSR with a primitive polynomial. The results are reported for those ISCAS-85 and combinational cores of the ISCAS-89 circuits for which the 10K long pseudo-random test sequence did not detect all the non-redundant faults. For the other ISCAS circuits, no pattern embedding is required for 10K long test sequences.

The pseudo-random test sequences have been simulated for resistive bridging faults before and after deterministic test cubes have been embedded. The fault set consists of 10K randomly selected non-feedback resistive bridging faults. A density function ρ (Section 2.2) derived from the one used in [Lee00] is employed for all experiments. All measurements are performed using the simulator from [Eng03]. The SAT-based ATPG procedure from [Eng04] is used for computing the exact value of ADI_G (Definition 2.5). Due to the fact that the embedded deterministic cubes consider only the stuck-at faults, resistive bridging faults are a valid surrogate of non-target defects.

Design	1K	5K	10K
c7552	92.38	93.51	94.68
cs09234	72.31	80.79	83.60
cs13207	76.56	86.76	91.47
cs15850	84.58	89.98	91.14
cs38417	86.23	90.57	92.61
cs38584	90.47	93.44	94.31

Table 5.12: Stuck-at coverage of pseudo-random sequences before deterministic cube embedding.

Due to the fact that AMSAL and the available briging fault simulator [Eng03] do not use compatible circuit formats, the Mentor Grapics tool, FlexTest, has been utilized for stuck-at fault simulation and deterministic test pattern generation. The deterministic test patterns generated by FlexTest have been transformed into deterministic test cubes by inserting *don't cares* based on fault simulation.

Table 5.13 reports the resistive bridging fault coverage FC_G (Definition 2.6) of the pseudo-random test sequence (*Random* FC_G) and of the test sequence obtained after the deterministic cubes had been embedded (*Embedded* FC_G), for the circuits mentioned in Table 5.12 and the test sequence lengths mentioned before. The size of the bit-flipping logic (*LSIZE*) is measured as the number of 2-input logic operators in the resulting circuit descriptions.

It can be seen that the resistive bridging fault coverage of the pseudo-random sequences is consistently higher than their stuck-at fault coverage. Interestingly, random pattern resistant faults seem to be distributed differently in the case of stuck-at and resistive bridging faults. Two circuits (*cs09234*, *cs38584*) have more random pattern resistant resistive bridging faults than the other circuits. While *cs09234* has the lowest stuck-at fault coverage, *cs38584* has the second highest stuck-at coverage. Hence, the validity of stuck-at fault coverage in identifying circuits with many random pattern resistant resistive bridging faults appears to be limited.

The resistive bridging fault coverage increases considerably due to embedding. However, the pseudo-random test patterns also contribute to the detection of non-target defects. This is implied by the fact that applying more pseudo-random test patterns results in significantly higher resistive bridging fault coverage. This can be seen best in the case of the two circuits with a large number of random pattern resistant resistive bridging faults, *cs09234* and *cs38584*, for which the coverage gain from 1K to 5K is 5% and 2%, respectively. Note that the circuits for which the sequence yielded good resistive bridging fault coverage before embedding also have the highest resistive bridging fault coverage after embedding.

Finally, it can be observed that the increase of the test sequence length reduces the overhead of the bit-flipping logic up to a factor of 2.4 (*cs13207*).

Design	1K			5K			10K		
	Random FC_G	Embedded FC_G	LSIZE	Random FC_G	Embedded FC_G	LSIZE	Random FC_G	Embedded FC_G	LSIZE
c7552	99.28	99.83	583	99.44	99.87	546	99.61	99.87	433
cs09234	90.68	98.55	1,097	95.30	99.26	824	96.55	99.39	683
cs13207	95.58	99.31	889	97.62	99.66	541	98.53	99.70	367
cs15850	96.29	99.36	1,107	98.34	99.67	783	98.81	99.70	686
cs38417	97.50	99.46	4,135	98.57	99.54	3,170	98.93	99.65	2,697
cs38584	93.01	98.74	894	95.10	99.43	878	96.47	99.67	590

Table 5.13: Resistive bridging fault coverage (FC_G) of the pseudo-random and embedded test sequences and DLBIST overhead (LSIZE).

In Table 6.1, the effect of using the *correction logic* (CRL) on the logic overhead of the bit-flipping DLBIST architecture is shown in the case of stuck-at fault testing. The final stuck-at fault efficiency and the 2-input logic gates in the circuit description of the bit-flipping logic and the correction logic are reported for both architectures (with and without CRL). The last column (*Overhead Improvement*) shows the ratio of the overhead with and without CRL.

The next experiments considered here refer to the evaluation of the bit-flipping DLBIST scheme, as proposed in Chapter 6, with respect to transition fault testing. The reported experimental results (Table 6.2 – 6.5) have been obtained using GNU Linux machines equipped with 2 GB of memory and an Intel Pentium 4 processor running at 2.4 GHz.

Table 6.2 presents the industrial designs that have been used as benchmark circuits. It is assumed that these circuits contain only single-cycle paths. The same circuit denomination is utilized as in the case of Table 5.5. The second and the third columns give the number of scan flip-flops *(# Flip-flops*) and scan chains *(# Scan Chains*) contained by each design. The following column (*Test length*) shows the length of the test sequence. The last two columns report the pseudo-random stuck-at and transition fault efficiencies (Definition 2.2), respectively. For each design, the last entry line corresponds to a test sequence whose application would require one second at the frequency of 100 MHz. The pseudo-random test patterns have been generated by a 32-stages long LFSR with a primitive polynomial and a phase shifter.

In Table 6.3, one can compare the results obtained using the bit-flipping DLBIST approach for the stuck-at and transition fault testing of the benchmarks in Table 6.2. Table 6.3 reports the number of embedded deterministic test cubes, the percentage of specified bits in each set of embedded test cubes, the achieved final fault efficiency and the cell area overhead of the BFF and CRL (Figure 6.3) implementations for both fault models. The overhead of the other parts of the DLBIST hardware is relatively small and it may be neglected.

In order to limit the hardware overhead in the case of the three largest designs, the number of deterministic test cubes embedded for transition fault testing has been limited to 800.

Design	Without CRL		With CRL		Overhead Improvement [%]
	BFF [# gates]	Final fault efficiency [%]	BFF+CRL [# gates]	Final fault efficiency [%]	
p19k	8,636	99.98	8,520	99.97	98.7
p59k	3,357	99.15	3,015	99.15	89.8
p127k	71,795	99.87	68,049	99.87	94.8
p278k	97,270	99.49	93,443	99.42	96.1
p333k	33,406	99.43	31,136	99.44	93.2

Table 6.1: CRL impact on the overhead of the bit-flipping DLBIST architecture.

Design	# Flip-flops	# Scan chains	Test length	Random stuck-at fault efficiency [%]	Random transition fault efficiency [%]
p19K	1,407	29	10K	80.80	73.66
			32K	85.54	82.64
			64K	90.38	86.97
			1923K	95.87	90.74
p59K	4,730	20	10K	97.12	81.87
			32K	97.94	85.63
			64K	98.11	87.31
			192K	98.35	89.67
p127K	5,116	11	10K	84.42	55.83
			32K	89.39	64.80
			64K	91.65	68.53
			187K	93.75	73.82
p278K	9,967		10K	84.29	63.86
			32K	88.66	71.02
			64K	90.62	75.00
			318K	93.38	82.81
p333K	20,756	30	10K	95.62	66.66
			32K	96.73	73.39
			64K	97.14	76.39
			140K	97.51	78.26

Table 6.2: Benchmark characteristics with respect to transition fault testing.

With the exception of the design *p19K*, the deterministic cubes embedded for transition fault testing have larger ratios of specified bits. This is due to the lower transition fault testability. In the case of the design *p19K*, this lower testability has the effect that the used ATPG tool (AMSAL) delivers less deterministic test cubes with less detected faults and also less specified bits per cube than in the case of stuck-at fault testing.

Design	Stuck-at fault testing				Transition fault testing			
	# Embedded patterns	Ratio of specified bits [%]	Fault efficiency [%]	Cell area [%]	# Embedded patterns	Ratio of specified bits [%]	Fault efficiency [%]	Cell area [%]
p19K	181	26.48	99.19	25	145	10.64	94.40	17
p59K	137	2.77	99.10	4	1,077	03.00	96.43	26
p127K	582	12.04	99.26	22	800	15.24	76.35	43
p278K	1,549	6.10	98.87	35	800	14.48	86.66	62
p333K	1,298	0.75	99.30	7	800	2.94	84.95	22

Table 6.3: DLBIST applied to stuck-at and transition fault testing (10K test patterns).

In all cases, the final stuck-at fault efficiency is much larger than the final transition fault efficiency. Moreover, this has been achieved along with a lower cell area overhead with the exception of the design *p19K*. The reason for this difference is again the lower random testability of the transition faults with the consequence that more patterns have to be embedded and more bits have to be flipped or preserved in the pseudo-random sequence. This seems not to be the case of the *p19K* design. Nevertheless, as mentioned before, here it was just the ATPG that provided fewer deterministic test cubes to be embedded for transition fault testing.

In Table 6.4, one can observe the impact of increasing the test length on the final fault efficiency and cell area overhead (BFF and CRL) of the considered DLBIST scheme used for transition fault testing. The run-time and memory requirements are reported as well.

As expected, the hardware overhead of the first two designs, for which the number of embedded patterns has not been limited, is significantly reduced by the increase of the test length. Extending the test length from 10K to 64K reduces the overhead by more than 10% of the CUT size. In the case of the last entry corresponding to the design *p19K*, increasing the test length by 2 orders of magnitude has reduced the overhead to half of the level from the previous entry that corresponds to a test sequence containing 64K patterns, at the price of a large increase in the run-time and memory requirements.

Design	Test length	# Embedded patterns	Run-time [h:m]	Memory [MB]	Final fault efficiency [%]	Fault efficiency improvement [%]	Cell area overhead [%]
p19K	10K	145	00:16	58	94.40	20.74	17
	32K	125	00:24	61	94.40	11.76	11
	64K	105	00:23	67	94.40	7.43	7
	1,923K	54	04:01	577	94.41	3.67	4
p59K	10K	1077	07:22	252	96.43	14.56	26
	32K	942	06:19	240	96.55	10.92	20
	64K	865	05:45	230	96.64	9.33	18
	192K	738	05:53	286	96.69	7.02	15
p127K	10K	800	32:55	716	76.35	20.52	43
	32K	800	32:09	738	82.20	17.40	44
	64K	800	31:47	755	84.98	16.45	44
	187K	800	30:17	786	87.75	13.93	42
p278K	10K	800	29:01	1,408	86.66	22.80	62
	32K	800	30:34	1,415	90.24	19.22	57
	64K	800	32:16	1,431	91.84	16.84	54
	318K	800	48:37	1,508	94.93	12.12	51
p333K	10K	800	33:40	758	84.95	18.29	22
	32K	800	35:58	760	86.77	13.38	19
	64K	800	35:29	742	87.61	11.22	17
	140K	800	35:39	801	88.25	9.99	15

Table 6.4: Test sequence length impact on DLBIST used for transition fault testing.

As in the case of the previous experiments, the number of embedded deterministic test cubes for the three largest designs has been limited to 800, in order to limit the hardware overhead. As long as the same number of deterministic test cubes is embedded, it is difficult to predict the dependence of the hardware overhead on the length of the test sequence. In this case, the overhead primarily depends on the average number of specified bits per embedded test cube, which is determined by the number and the difficulty of the target faults.

Longer pseudo-random test sequences leave undetected faults which are more difficult to test. This tends to increase the number of specified bits necessary to detect the remaining fault. On the other hand, this may also decrease the number of new detected faults per embedded test cube. That is why it is difficult to predict the evolution of the average number of specified bits per embedded test cube when the length of the test sequence is augmented. Increasing the length of the test sequence also improves the pattern embedding opportunities.

In the case of the design *p127K*, increasing the length of the test sequence does not significantly change the hardware overhead, but it improves the final fault efficiency by more than 11%. In the case of the designs *p278K* and *p333K*, increasing the length of the test sequence has a twofold beneficial impact. Choosing a test sequence length of 318K and 140K instead of 10K reduces the overhead by 11% and 7%, respectively. In parallel, the final fault efficiency is improved by more than 8% and 3%, respectively. It should be mentioned that the increase of the test sequence length improves the coverage of the non-modeled defects as well (Section 5.6).

In the case of the three largest designs, increasing the length of the test sequence has no significant impact on the run-time and memory requirements.

Table 6.5 reports possible trade-offs between the fault efficiency and the hardware overhead in the case of the largest three benchmark designs. The considered test sequences contain the maximum number of test patterns which can fit in one second of test time at the frequency of 100 MHz. In the case of the designs *p127K* and *p278K*, 10 deterministic patterns are already enough to obtain a larger fault efficiency than in the case when 800 deterministic test patterns are embedded into a 10K long test sequence. In this way, the hardware overhead can be reduced to 1% from 43% and 62% (Table 6.4), respectively. In the case of the design *p333K*, a similar fault efficiency can be achieved by embedding 100 deterministic test patterns, at the cost of 5.5%, instead of 22%, hardware overhead.

The experiments described in the following have been conducted to evaluate the FBDD-based logic synthesis approach proposed in Chapter 7. For this purpose, SIS [Sen92] and the OBDD-based methods available in the CUDD-package [Cudd] that are able to handle *don't cares* have been used as reference. The experiments have been performed on GNU Linux machines equipped with 2 GB of memory and an Intel Pentium 4 processor running at 2.4 GHz.

Table 7.1 presents three multi-output bit-flipping functions which will be used as benchmark functions. These functions can be downloaded from [Fbdd]. The second and the third column report the number of inputs and outputs of the target functions. The fourth column (||ON-set|| + ||OFF-set||) gives the sum of the cardinalities of the ON-set and the OFF-set corresponding to each function. The last two columns show the non-terminal node count of the OBDD-based representation of each function.

Design	Test length	# Embedded patterns	Run-time [h:m]	Memory [MB]	Final fault efficiency [%]	Fault efficiency improvement [%]	Cell area overhead [%]
p127K	187K	10	5:16	477	76.83	3.01	1
		50	5:38	492	79.48	5.66	3
		100	6:41	510	80.56	6.74	5
		400	16:22	731	85.28	11.46	24
		800	30:17	786	87.75	13.93	42
p278K	318K	10	23:44	1,150	87.96	5.15	1
		50	24:00	1,169	88.98	6.17	4
		100	26:44	1,190	90.71	7.90	6.5
		400	34:53	1,306	93.26	10.45	27.5
		800	48:37	1,508	94.93	12.12	51
p333K	140K	10	5:33	528	81.29	3.03	1.5
		50	7:36	630	83.32	5.06	4
		100	9:44	661	84.47	6.21	5.5
		400	31:05	786	87.28	9.02	11.5
		800	35:39	801	88.25	9.99	15

Table 6.5: Possible trade-offs between the fault efficiency and the hardware corresponding to the maximum test length which can fit in one second of test time at the frequency of 100 MHz.

First, the FBDD-based approach has been compared to the *restrict* operator (Definition 4.14). This evaluation has been done with respect to the synthesis of each single output of the functions in Table 7.1. The results are reported in Table 7.2. For the other OBDD-based optimization methods from [Cudd] that are able to handle don't cares, like *constrain* (Definition 4.13) or *squeeze*, similar results have been obtained as with *restrict*.

The node sharing among the ROBDDs in the manager of the CUDD-package, which correspond to different outputs of the target functions, has not been taken into account and, consequently, no node or logic sharing has been allowed among the FBDD-based covers of the different outputs.

Multi-output function	#inputs	#outputs	||ON-set|| + ||OFF-set||	ON-BDDs size [# nodes]	OFF-BDDs size [# nodes]
p19K	82	24	85,215	615,379	654,387
p59K	77	19	9,918	158,160	315,314
p127K	67	10	663,750	6,876,383	8,067,136

Table 7.1: Multi-output incompletely specified benchmark functions.

Single-output function		*Restrict*-based			FBDD-based			FBDD/*Restrict*		
f_{on} size [#nodes]	f_{off} size [#nodes]	#nodes	#gates	Run-time	#nodes	#gates	Run- time	#nodes	#gates	Run-time
142	142	2	3	0ms	1	0	0ms	0.50	0	-
207	77	2	1	0ms	1	0	0ms	0.50	0	-
162	160	3	3	0ms	1	0	0ms	0.33	0	-
206	308	6	5	0ms	1	0	0ms	0.17	0	-
142	450	5	5	0ms	1	0	0ms	0.20	0	-
321	801	9	10	0ms	1	0	0ms	0.11	0	-
519	950	12	15	0ms	6	6	10ms	0.50	0.40	-
605	1,554	21	28	0ms	7	6	20ms	0.33	0.21	-
1,011	2,447	29	39	0ms	10	11	20ms	0.34	0.28	-
1,096	2,853	31	46	0ms	5	4	10ms	0.16	0.09	-
1,696	1,936	30	49	0ms	11	14	50ms	0.37	0.29	-
1,637	3,902	54	82	10ms	14	18	60ms	0.26	0.22	6
2,132	3,416	52	76	0ms	3	2	10ms	0.06	0.03	-
2,877	4,483	70	117	0ms	24	31	80ms	0.34	0.26	-
2,983	5,421	61	105	0ms	22	30	420ms	0.36	0.29	-
4,533	5,369	82	146	0ms	15	17	60ms	0.18	0.12	-
4,997	8,100	105	208	0ms	41	52	370ms	0.39	0.25	-
6,516	8,592	109	190	0ms	16	20	80ms	0.15	0.11	-
6,672	10,402	130	243	0ms	30	37	150ms	0.23	0.15	-
10,620	15,783	199	365	10ms	68	102	290ms	0.34	0.28	29
9,644	17,004	202	390	10ms	49	61	250ms	0.24	0.16	25
14,152	22,127	259	493	10ms	38	50	180ms	0.15	0.10	18
21,934	30,621	369	736	10ms	27	33	310ms	0.07	0.04	31
24,298	32,539	368	774	10ms	22	23	270ms	0.06	0.03	27
30,745	63,286	651	1,337	20ms	235	373	4s:370ms	0.36	0.28	218
60,075	87,712	897	1,973	20ms	230	342	3s:310ms	0.26	0.17	165
46,628	116,086	1,133	2,261	20ms	287	464	5s:370ms	0.25	0.21	268
64,744	128,072	1,260	2,763	30ms	285	429	7s:530ms	0.23	0.16	251
101,594	133,596	1,330	3,136	40ms	488	778	13s:410ms	0.37	0.25	335
102,056	134,892	1,371	3,177	40ms	69	82	3s:290ms	0.05	0.03	82
122,817	168,964	1,646	3,830	50ms	347	541	11s:310ms	0.21	0.14	226
121,533	179,303	1,682	3,911	50ms	128	170	4s:800ms	0.08	0.04	96
128,631	175,122	1,673	3,945	50ms	84	103	3s:250ms	0.05	0.03	65
125,024	181,348	1,700	3,977	50ms	65	80	3s:280ms	0.04	0.02	65
135,639	171,062	1,754	4,092	50ms	143	194	5s:560ms	0.08	0.05	111
140,137	179,155	1,758	4,140	50ms	97	123	4s:230ms	0.06	0.03	84
168,650	202,362	2,075	4,908	60ms	104	136	5s:270ms	0.05	0.03	87
162,915	215,385	2,129	4,980	60ms	103	124	5s:840ms	0.05	0.02	97
165,145	216,792	2,129	4,975	60ms	546	876	20s:510ms	0.26	0.18	341
172,147	242,628	2,221	5,319	70ms	858	1,485	38s:500ms	0.39	0.28	550
170,524	249,451	2,253	5,326	70ms	877	1,483	43s:070ms	0.39	0.28	615
188,465	249,986	2,364	5,651	80ms	601	967	31s:330ms	0.25	0.17	391
195,170	245,448	3,683	8,699	090ms	1,068	1,758	25s:840ms	0.29	0.20	287
397,885	514,091	7,117	17,405	210ms	2,635	4,796	6m:19s	0.37	0.28	1,804
657,647	816,669	11,268	28,345	340ms	2,245	3,836	5m:21s	0.20	0.14	944
1,025,346	1,261,458	16,832	43,348	530ms	4,295	7,463	14m:59s	0.26	0.17	1,696
1,001,823	1,305,252	17,343	44,309	490ms	6,927	13,001	22m:42s	0.40	0.29	2,779
1,296,617	1,609,523	21,175	54,782	630ms	7,190	12,960	32m:17s	0.34	0.24	3,074
1,330,764	1,706,077	22,397	57,813	610ms	7,771	14,362	28m:46s	0.35	0.25	2,829
1,408,613	1,749,112	22,837	59,246	700ms	7,672	14,030	46m:57s	0.34	0.24	4,024
1,429,387	1,760,305	22,979	59,886	730ms	8,227	14,972	38m:19s	0.36	0.25	3,149
1,732,319	2,137,125	28,219	73,670	870ms	8,357	15,041	1h:05m	0.30	0.20	4,482

Table 7.2: Comparison between the FBDD-based optimization approach and the approach based on the *restrict* operator.

The first two columns in Table 7.2 report the number of non-terminal nodes (*size*) of the ROBDD-based representation of each output of the functions in Table 7.1. The synthesis results obtained with the *restrict* operator and the FBDD-based method are shown in the following six columns. The two methods are evaluated with respect to the required run-times, the non-terminal node counts (*#nodes*) and the logic operator counts (*#gates*) in the resulting circuit descriptions.

The number of logic operators in the circuit description of a non-terminal FBDD node is obtained by counting the 2-input logic operators in the expression of the corresponding cover *Cov*(*F*). In the case of the OBDD-based implementation, the circuit description of each non-terminal node may require 3, 1 or 0 2-input logic operators, depending on whether the node has 0, 1 or respectively 2 children, that are terminal nodes [Bec92]. The terminal nodes require no hardware implementation and, as a consequence, their gate count is zero. The circuit description of a BDD with only 1 non-terminal node requires no logic operator (gate) for its implementation. This is the case of the first examples in Table 7.2. 1-input logic operators (e.g. INV-operator) are not counted. The last three columns in Table 7.2 report the ratios between the node counts *(#nodes*), logic operator counts (*#gates*) and the run-times (*run-time*) required by *restrict* and the FBDD-based method.

The FBDD-based approach outperforms the *restrict*-based approach with respect to the node count and the number of logic operators in the resulting circuit descriptions at the cost of a run-time increase. The run-time requirement of the proposed approach can be significantly reduced by decreasing the searching space associated with the DC-based node reduction. On the other hand, enlarging this searching space will further improve the compaction of the resulting FBDD-like cover. The size of the searching space associated with the DC-based node reduction can be controlled with the help of several thresholds described in Appendix 2.

Table 7.3 provides a comparison between the FBDD-based method and OBDD-based approaches that use combinations of variable reordering and the *restrict* operator. This evaluation has been done with respect to the synthesis of the functions in Table 7.1. The number of 2-input logic operators *(#gates*), the node depth (*Node depth*) and the 2-input gate depth (*Gate depth*) of the resulting covers as well as the run-time required to generate these covers (*Optimization time*) are reported for both approaches.

Each function has been synthesized three times with each approach. In the case of the FBDD-based approach, the reported experiments show tradeoffs between the run-time and the number of 2-input logic operators in the circuit description of the resulting covers. These tradeoffs have been obtained by changing the thresholds that control the size of the searching space associated with the DC-based node reduction. The first run corresponding to each function has been done with the DC-based node reduction disabled.

In the OBDD-based approaches used for the evaluation of the FBDD-based method, the variable reordering has been applied before *restrict* and to all ON- and OFF-ROBDDs corresponding to each output of the target function. As a result, all the covers obtained with the *restrict* operator have the same variable ordering and, consequently, a maximized probability of node sharing among them.

An unexpected observation is that the variable reordering performed on the covers found with *restrict* does not bring any node reduction of a multi-output implementation. Due to the fact that variable reordering is a time consuming procedure, the reported run-time consumption of the ROBDD-based approach with variable reordering takes into account only the application of the *restrict* operator and of the variable reordering done before it.

In the first OBDD-based run, no variable reordering has been performed. In the next two runs, the variables have been reordered based on the heuristics: CUDD_ REORDER_SYMM_SIFT and CUDD_REORDER_SYMM_SIFT_CONV [Cudd], respectively. The first heuristic is an implementation of symmetric sifting [Pan94], while the second heuristic is a converging variant of the first one. Variable reordering improves the operator count at the cost of a significant increase in the run-time. The converging heuristic for reordering the variables of the function *p127K* was still incomplete after days of execution.

The proposed method outperforms all the investigated OBDD-based approaches. Running the FBDD-based flow with the DC-based node reduction switched off results in operator counts (#gates) that are between two and four times better than those obtained with the best investigated ROBDD-based approach. The operator count of the FBDD-based covers can be further improved by enabling the DC-based node reduction and increasing the associated searching space.

The FBDD-based approach with DC-based node reduction disabled also provides the implementations with the smallest depths. In the case where the DC-based node reduction is enabled, the maximum node depth is always less than the number of input variables. A variable index appearing more than once on a path from root to a leave node has never been observed.

The circuit descriptions presented in Table 7.3 have been synthesized with Synopsys Design Compiler and using a proprietary library. Table 7.4 reports the resulting area (*Cell area*) measured in an arbitrary unit, the synthesis run-time (*Synthesis time*) and the total run-time required to generate the covers (*Optimization time,* in Table 7.3) and to synthesize them (*Synthesis time,* in Table 7.4).

Multi-output function	Restrict + Variable Reordering				FBDD			
	#gates	Node depth	Gate depth	Optimization time	#gates	Node depth	Gate depth	Optimization time
p19K	54,672	17	30	0m:20s	8,269	15	24	1m:31s
	39,231	17	30	4m:52s	7,200	23	37	17m:28s
	33,443	17	29	40m:22s	7,161	27	42	22m:07s
p59K	7,084	20	33	2s	1,543	16	25	11s
	4,669	19	31	2m:16s	1,428	23	34	27s
	4,601	19	30	18m:27s	1,423	23	34	1m:10s
p127K	390,057	23	42	24m:21s	120,122	21	36	35m:18s
	256,883	24	42	11h:16m	94,113	68	97	15h:00m
	-	-	-	-	93,837	61	96	16h:34m

Table 7.3: Optimization potential of the FBDD-based and the OBDD-based (*restrict* + variable reordering) approaches.

Multi-output function	Restrict + Variable Reordering			FBDD		
	Cell area	Synthesis time	Optimization + Synthesis time	Cell area	Synthesis time	Optimization + Synthesis time
p19K	147,074	46m:32s	46m:52s	34,464	1m:56s	3m:27s
	101,332	25m:30s	30m:22s	33,286	1m:29s	18m:57s
	89,681	17m:12s	57m:34s	32,917	1m:30s	23m:37s
p59K	23,075	1m:54s	1m:56s	7,014	30s	41s
	15,198	1m:02s	3m:18s	7,046	37s	1m:04s
	15,292	1m:07s	19m:34s	6,869	29s	1m:39s
p127K	1,349,051	15h:02m	15h:26m	521,814	4h:40m	5h:15m
	1,036,493	7h:06m	18h:22m	507,949	2h:51m	17h:51m
	-	-	-	508,840	3h:07m	19h:41m

Table 7.4: Synthesis results obtained using the FBDD-based and the OBDD-based (*restrict* + variable reordering) approaches.

Compared to the best investigated OBDD-based approach, the FBDD-based flow with the DC-based node reduction disabled reduces the area figures by a factor between two and three. This improvement has been achieved by using shorter run-times as compared to all OBDD-based approaches, if one considers the sum of *Optimization time* and *Synthesis time*. Moreover, the run-time of this simple configuration of the FBDD-based approach is by at least one order of magnitude shorter than the run-time of the OBDD-based approach with the best logic area results.

In the case of the FBDD-based approach, the area results can be further improved by enabling the DC-based node reduction. Nevertheless, the logic area is not always reduced by enabling the DC-based node reduction. A reason for this surprising phenomenon is the fact that the DC-based node reduction approach has not been tuned towards improving the area performance of Design Compiler, considered here as a black-box. This also indicates that Design Compiler can perform an efficient node reduction, equivalent to the node reduction based on graph isomorphism. The DC-based node reduction of the FBDD-based approach is especially useful in the case where the available logic synthesis tool cannot perform efficient logic optimizations.

Table 7.5 presents a comparison between SIS [Sen92] and the FBDD-based approach with respect to the implementation of incompletely specified functions with large DC-sets. Due to the scaling problems of SIS, only some of the smallest but untrivial functions that correspond to single outputs of the functions presented in Table 7.1 could be implemented. The second column reports the number of inputs of each single-output function. The third column (||ON-set|| + ||OFF-set||) gives the sum of the cardinalities of the ON-set and the OFF-set corresponding to each function. The fourth and the fifth columns show the non-terminal node count of the OBDD-based representation of each function. The next three columns (SIS) report the resulting number of gates, area and the required run-time when the target functions have been implemented directly with SIS. In the last three columns (FBDD+SIS), the same parameters are reported for the case where FBDD-like covers have been generated and later synthesized using SIS. In all the cases, SIS has been run with the *rugged script*. The statement *full_simplify -m nocomp* has been inserted at the beginning of the script. The library *nand-nor.genlib* has been used.

Single-output function	#inputs	||ON-set|| + ||OFF-set||	ON-BDD size [#nodes]	OFF-BDD size [#nodes]	SIS			FBDD + SIS		
					#gates	Cell area	Run-time [s]	#gates	Cell area	Run-time [s]
p1	82	229	6,516	8,592	354	760	28.60	21	39	0.11
p2	82	843	21,934	30,621	180	395	3.83	31	64	0.31
p3	77	1,708	30,745	63,286	674	1,534	1,046.74	366	754	26.85
p4	77	3,652	64,744	128,072	1,145	2,586	3,997.47	366	820	7.58

Table 7.5: Comparison between SIS and the FBDD-based approach combined with SIS.

It is obvious that the FBDD-based approach scales better and improves dramatically the number of gates and area (between 2 and 19 times). This suggests that the proposed FBDD-based approach enables a much better use of the *don't cares* which in the descriptions of SIS and MIS are referred to as *external don't cares* [Bra87] [Sen92].

Appendix 2 – Implementation of the Proposed Methods

This appendix presents some information related to the C/C++ code that implements the DFT flow sketched in Figure 6.4 and the logic synthesis algorithm shown in Figure 7.2. The DFT flow has been integrated into an industrial tool of Philips (AMSAL[11]), and its algorithm is a generalization of the algorithm in Figures 5.5.

The algorithm is implemented by the function *do_bddFlow*. Relative to the storage system of the *Institut für Technische Informatik* (ITI) at the University of Stuttgart, the function *do_bddFlow* is included in the file:

/home/ghermanv/vob_39_sa/amsal/src/atpg/src/bitfliping/bitfliping/bddFlow.cxx

in the case of stuck-at fault testing, and in the file:

/home/ghermanv/vob_gd/amsal/src/atpg/src/bitfliping/bitfliping/bddFlow.cxx

in the case of transition fault testing. The paths above correpond to the AMSAL release 3.9.

Relative to the flow presented in Figure 6.4, its tasks are executed by the functions listed in Table 9.1.

Function	Task
do_simulateLfsrPattern	Performs the fault simulation
do_atpg	Performs the ATPG
do_mapping	Implements the pattern mapping algorithm
LogicSyntheis_BFF	Implements the BDD-based optimization and logic synthesis of the BFF
LogicSyntheis_CRL	Implements the BDD-based optimization and logic synthesis of the CRL
LogicSyntheis_WEIGHT	Synthesizes a combinational module that can be used to weight, with a single set of complementary weights, the pseudo-random test sequence where the deterministic test cubes are embeded

Table 9.1: The functions that implement the flow presented in Figure 6.4.

[11] Automatic Multi restartable Scan test pattern generation And Localization of faults.

In the same folder with the file *bddFlow.cxx* are the files *bflBdd.** in which the class *CbflBdd* is defined. The most important methods of this class are explained in Table 9.2. In the same folder, the files FBDD.* and bdd.* can be found, where the structure *Node* and the class *CBdd* are defined. The structure *Node* contains all the parameters of a FBDD node (Chapter 7). The class *CBdd* is an encapsulation of the *BDD* class defined in the CUDD-package [Cudd].

In order to enable different configurations of the considered DFT flow described in Figure 6.4, different thresholds and flags are defined at the beginning of the files *bflBdd.** and *bddFlow*.cxx. Some of the most important of these parameters are described in Table 9.3.

The DFT flow described in Figure 5.2 is implemented by the procedure *processBitFlip*, which is included in the file:

/home/ghermanv/vob_3.4.0/amsal/src/atpg/src/bitfliping/bitfliping/bitflip.cxx

relative to the storage system of the *Institut für Technische Informatik* (ITI) at the University of Stuttgart.

Function	Task
AssignTestPattern	Implements the mapping cost function used by the pattern mapping algorithm It is called by *do_mapping*
LogicSyntheis_BFF and *LogicSyntheis_CRL*	See Table 9.1
OBdd2FBdd	Implements the heuristics described in Figure 7.2 It is called by *LogicSyntheis_BFF* and by *LogicSyntheis_CRL*
search	Performs the DC-based node reduction described in Chapter 7 It is called by *ROBdd2FBdd*
WriteVhdlBfl	It is called by the function *do_writeLogic* from the already mentioned top function *do_bddFlow*
traverse	It is used to dump the BFF and CRL in VHDL format It is called by *WriteVhdlBfl*
GetFlippedPattern	It checks whether the bits of a pattern generated by the LFSR and, eventually, by a phase shifter (PS) have to be flipped by the BFF and the CRL

Table 9.2: The most important methods of the class *CbflBdd*.

Parameter Name	File	Used to
ComplWeight	*bflBdd.h*	Enable the weighting with a single set of complementary weights of the pseudo-random test sequence where deterministic patterns are embedded
SequencePartitioning	*bflBdd.h*	Set the fraction of the test sequence where deterministic cubes are embedded
useCRL	*bflBdd.h*	Enable the use and the implementation of the correction logic
THRESHOLD_OVERLAP	*bflBdd.cxx*	Control the number of FBDD nodes that point to the ROBBD-based representation of the function implemented by their sub-graph
THRESHOLD_SEARCH	*bflBdd.cxx*	Control the size of the searching space used for the DC-based node reduction (Section 7.3)
PERMUTATION	*bflBdd.cxx*	Permute the groups of ROBDD variables corresponding to the state of the: LFSR, shift counter (SC), pattern counter (PC), phase shifter (PS) and scan chain number
INVERSION	*bflBdd.cxx*	Inverse the order of variables inside the groups mentioned above
considerLFSR / considersPS	*bflBdd.cxx*	Enable the inclusion of the LFSR/PS states in the definition space of the BFF
FLIP2FIX	*bflBdd.cxx*	Set OFF-set = ¬ON-set Transforms the BFF in a completely specified function
Store_bflBdd / Load_bflBdd	*bddFlow*.cxx	Stores/Loads the ROBDD-based representation of the BFF
Store_atpg / Load_atpg	*bddFlow*.cxx	Stores/Loads the deterministic test cubes to be embedded
Mapping	*bddFlow*.cxx	Choose one pattern mapping heuristic

Table 9.3: Thresholds and flags used to configure the DFT flow in Figure 6.4.

Appendix 3 – Related Papers

Conference Proceedings

1. V. Gherman, H.-J. Wunderlich, J. Schlöffel, M. Garbers "Deterministic Logic BIST for Transition Fault Testing," *IEEE European Test Symposium (ETS)*, 2006, pp. 123-128.
2. A.W. Hakmi, V. Gherman, H.-J. Wunderlich, M. Garbers, J. Schlöffel "Implementing a Scheme for External Deterministic Self-Test," *IEEE VLSI Test Symposium (VTS)*, 2005, pp. 101-106.
3. V. Gherman, H.-J. Wunderlich, H. Vranken, H. Vranken, F. Hapke, M. Wittke, M. Garbers "Efficient Pattern Mapping for Deterministic Logic BIST," *IEEE International Test Conference (ITC)*, 2004, pp. 48-56.

Workshop Contributions

1. P. Engelke, V. Gherman, I. Polian, Y. Tang, H.-J. Wunderlich, B. Becker "Sequence Length, Area Cost and Non-Target Defect Coverage Tradeoffs in Deterministic Logic BIST," *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2005, pp. 11-18.
2. P. Engelke, V. Gherman, I. Polian, Y. Tang, H.-J. Wunderlich, B. Becker "Sequence Length, Area Cost and Non-Target Defect Coverage Tradeoffs in Deterministic Logic BIST," *17th ITG/GI/GMM Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen,"* 2005, pp. 16-20.
3. V. Gherman, H.-J. Wunderlich, M. Garbers, J. Schlöffel "DLBIST for Delay Testing," *17th ITG/GI/GMM Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen,"* 2005, pp. 39-43.
4. A.W. Hakmi, H.-J. Wunderlich, V. Gherman, M. Garbers, J. Schlöffel "Implementing a Scheme for External Deterministic Self-Test," *17th ITG/GI/GMM Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen,"* 2005, pp. 27-31.
5. V. Gherman, H.-J. Wunderlich "Scalable Deterministic Logic Built-In Self-Test (DLBIST)," (poster), *Ekompass-Workshop*, Hannover, May, 2004.
6. V. Gherman, H.-J. Wunderlich "BDD-Based Implementation of the Bit-Flipping DLBIST," *Fachworkshop Testen*, Hannover, November, 2003.

Appendix 4 – Short Presentation of the Author

Valentin Gherman received the M.S. degree in technical physics from the "Politehnica" University of Bucharest, Romania, in 1997, and the Dipl.-Phys. degree in physics from the University of Siegen, Germany, in 2000. He also completed the curriculum *Maîtrise en Physique* at "Université de Bourgogne", France, in 1995, as participant in the program TEMPUS for student exchange in Europe.

From 2000 to 2001, he was research assistant at the University of Siegen, where he developed software models for the calculation of ground state observables (energy, number of spin flips) for two dimensional spin textures (skyrmions).

He has been with the University of Stuttgart since 2001, where he has worked as Teaching Assistant and has been involved in the project AZTEKE supported by the German Federal Ministry of Education and Research (BMBF) under the contract number 01M3063C.

Valentin Gherman is member of IEEE.

His research interests are in logic synthesis, BDDs, data compression and logic BIST.