INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Test Methodology for SRAM based FPGAs

By
Gurpreet S. Bhullar, B.Eng.

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

Master of Engineering

Department of Electronics

Carleton University
Ottawa, Ontario
May 22, 1997
©copyright
1997, Gurpreet S. Bhullar

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-22116-4

Test Methodology for SRAM based FPGAs

T. the scientist

V. Swark

Singht

Date: May 22, 1997

Abstract

The reconfiguration feature of SRAM FPGAs presents the possibility of implementing fault tolerant or reparable systems based on FPGAs. The implementation of these systems is critically dependent on the ability to detect and localize faults in SRAM FPGAs.

This theses addresses the development of a test methodology that allows fault detection and localization in logic and routing resources of coarse grained SRAM FPGAs. Furthermore, the test methodology is intended to address a broad range of architectures and device sizes. The testing of an SRAM FPGA involves the testing of logic and routing resources. A test approach for logic resources is presented that allows full functional fault coverage of Configurable Logic Blocks (CLBs) through use of specialized test configurations. These CLB test configurations and their test vectors were developed for the XC3000 and XC4000 devices. A methodology is presented whereby multiple CLBs in a device can be exercised in parallel with the above test configurations. Algorithms for dynamic generation of device configurations are presented that allow the test approach to be applicable to a broad range of device architectures and sizes. The algorithms were implemented and the resulting software was tested for the smallest and largest devices in both the XC3000 and the XC4000

architectures, thereby demonstrating the flexibility of the methodology.

An architecture independent methodology is presented that employs a functional level device model to achieve fault detection and location in routing resources of SRAM FPGAs. Test algorithms based on the device model are presented that allow dynamic generation of test configurations. These test configurations allowed full fault detection and location of single and multiple stuck-at and stuck-open faults in all local line resources. In addition, the test configurations allowed a high degree of fault coverage for programmable interconnect points, global lines and switch matrices. A functional level device model of the XC3020 device was developed for verification of test algorithms. The algorithms for developing the test configurations were implemented and the resulting software was verified against an injected fault set for the XC3020 device model using a fault simulator specific to that model.

A test environment is presented that allows the application of the logic and routing resource test methodology to FPGA devices. The test environment provides an interface between the the software developed in this research, the vendor CAD tools and a generic IC tester to allow automated testing of the DUT.

The information used in this thesis comes in part from the research programs of Prof. T. A. Kwasniewski, and Dr. V. Szwarc and their associates at Carleton University and CRC. The research results appearing in this thesis represent an integral part of the ongoing research program. All research results in this thesis including tables, graphs, figures and software but excluding narrative portions of the thesis are effectively incorporated into the research program and can be used by Prof. Kwasniewski, Dr. Szwarc and their associates for education and research purposes, including publication in the open literature with appropriate credits. Matters of intellectual property may be pursued collaboratively with Carleton University, CRC and or funding agencies where and as appropriate.

Acknowledgements

I wish to thank Profesor Tad Kwasniewski, and Dr. Valek Szwarc, my supervisors, for their support and patient guidance during my research at Carleton University and Communications Research Center.

The financial support of Carleton University, Canadian Institute for Telecommunications Research (CITR) and Communications Research Center (CRC) is gratefully acknowledged.

Finally, the author would like to dedicate this thesis to his parents without whose love this project would never have been undertaken.

Carleton University
Ottawa, Canada K1S 5B6

Text complete; leaf V// omitted in numbering.

Table of Contents

List of Figures	x
List of Tables	xii
List of Acronyms, Abbreviations and Terms	xiii
List of Symbols	xv
1 Introduction	1
1.1 Field Programmable Gate Arrays (FPGAs)	1
1.2 Current Approaches to Testing FPGAs	6
1.3 Thesis Statement	9
1.4 Testing for Reconfigurable Systems	9
2 Background	12
2.1 FPGA Programming Technologies	12
2.2 Architecture of a Representative SRAM FPGA	15
2.3 FPGA Test Requirements for Reconfigurable Systems	17
3 A Preliminary Test Approach for SRAM FPGAs	21
3.1 Testing of Logic Resources	22
3.1.1 A Test Method Using Regular Arrays	23
3.2 Testing of Routing Resources	25
3.3 Summary	34
4 A Methodology for Testing Logic Resources of SRAM FPGAs	35
4.1 CLB Test Configurations for XC3000 FPGAs	36
4.1.1 A Fault Model for LUT Based CLBs	36
4.1.2 Test Configurations and Test Vectors for CLB Primitives	39
4.1.3 XC3000 CLB Test Configurations	46
4.2 A Test Approach for Multiple CLBs for XC3000 FPGAs	49
4.2.1 Test Operation	52
4.3 An Architecture Independent Methodology for Testing Multiple CLBs	53

4.3.1	Implementation of Architecture Independent Approach	56
4.4	Summary	60
5	The Application of CLB Test Methodology to XC4000 FPGAs	62
5.1	CLB Test Configurations for XC4000 FPGAs	63
5.1.1	A Functional Fault Model for XC4000 CLBs	63
5.1.2	Test Configurations and Test Vectors for CLB Primitives	64
5.1.3	XC4000 CLB Test Configurations	71
5.2	Implementation of Architecture Independent Approach	73
5.2.1	Implementation Results	75
5.3	Limitations on Device Configuration Generation	77
5.4	Summary	81
6	A Methodology for Testing Routing Resources of SRAM FPGAs	82
6.1	A Device Model and Data Structures for XC3000 Devices	84
6.2	Test Approach for Routing Resources	86
6.3	Testing Local Line Resources	87
6.3.1	Algorithms for Generating Local Line Test Configurations	88
6.4	Testing PIPs	98
6.5	Fault Tracing Algorithm	102
6.6	Results	104
6.7	Summary	107
7	Integration of Logic and Routing Resource Testing	108
7.1	The Need for Integration	109
7.2	An Integrated Test Methodology	110
7.3	An Overview of the Test Environment	112
7.4	Integration of the Test Methodology Into a Reconfigurable System	115
7.5	Summary	117
8	Summary and Future Research	118
8.1	Areas for Enhancement and Future Research	120
A	XC3000 CLB Test Vectors	125

List of Figures

1.1	Gate array architectures	2
1.2	A typical coarse grained FPGA architecture	4
2.1	A PLICE antifuse	13
2.2	A floating gate device	14
2.3	SRAM programming technology	14
2.4	XC3020 logic and routing resources	16
3.1	XC3020 CLB architecture	23
3.2	A BSP multiplier	24
3.3	CLB resources covered by a BSP multiplier configuration	25
3.4	A typical grid generated during a test	30
4.1	XC3000 CLB with functionality defined in terms of configuration bits	38
4.2	Combinatorial logic modes FG, FGM and F implemented by a look-up table	41
4.3	Functional representation of look-up tables in terms of configuration bits	42
4.4	A schematic representation of a configuration for testing multiple CLBs	50
4.5	Implementation of architecture dependent approach	51
4.6	An example XC3020 device configuration	59
5.1	XC4000 CLB architecture	64
5.2	XC4000 dedicated carry logic	69
5.3	An example XC4002 device configuration	76
5.4	XC4000 carry logic test configuration	79
6.1	An example model representing local lines, CLB inputs and their interconnections	85
6.2	An example stage 2 local line test configuration	97
6.3	Net coverage by 4 PIP configurations	101
6.4	Local line testing simulation results for a fault set of 18 segments, all detected within 25 iterations	105

6.5	Simulation results with 16 I/Os prohibited for a fault set of 18 segments, all detected within 30 iterations	106
7.1	Test execution sequence	114

List of Tables

2.1	LUT implementing function $f = a \cdot b + c$	16
4.1	Function generator modes	41
4.2	Test vectors and expected outputs for multiplexers F and G	44
4.3	Test configurations for a function generator block	44
4.4	Transition table for D-registers	45
4.5	CLB test configurations defined in terms of CLB inputs and configuration bits E1-9	47
4.6	Results for dynamic generation of XC3020 device configurations with 64 and 48 available I/Os	58
4.7	Results for dynamic generation of XC3195 device configurations	60
5.1	Carry logic operational modes	68
5.2	The subset of carry logic operational modes for full test coverage	70
5.3	Results from dynamic generation of device configuration for XC4002 & XC4005 devices	77

List of Acronyms, Abbreviations and Terms

AMD	Analog Micro Devices
ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generator
BIST	Built In Self Test
BSP	Bit Serial word Parallel
CAD	Computer Aided Design
CIN	Carry Input
COUT	Carry Output
CL	Carry Logic
CLB	Configurable Logic Block
DUT	Device Under Test
EEPROM	Electrically Erasable Programmable ROM
EPROM	Erasable Programmable ROM
FF	Flip-Flop
FG	Function Generator
FPGA	Field Programmable Gate Array
FPIS	Field Programmable Interconnect System
IC	Integrated Circuit
IOB	Input Output Block
LUT	Look Up Table
MCM	Multi-Chip Module
NPSF	Neighborhood Pattern Sensitive Fault
ORA	Output Response Analyzer
ORCA	Optimized Reconfigurable Cell Array
PE	Processing Element
PFU	Programmable Function Unit
PIP	Programmable Interconnect Point
PLB	Programmable Logic Block

PLC	Programmable Logic Cell
PLD	Programmable Logic Device
PLICE	Programmable Low Impedance Circuit Element
PSF	Pattern Sensitive Fault
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SAF	Stuck At Fault
SA1	Stuck-at 1 fault
SA0	Stuck-at 0 fault
SO	Stuck-open fault
SPT	Shortest Path Tree
SRAM	Static Random Access Memory
TAP	Test Access Port
TBUF	Tri-statable Buffer
TF	Transition Fault
UV	Ultra Violet
WLB	Write Logic Block
Long Lines	Interconnects running across entire device
Short Lines	Interconnects running across the length of a CLB

List of Symbols

PIP with current flow from segment a to b

PIP with current flow from segment b to a

PIP connecting broken segment a-b, shaded PIP indicates programmed connection

AND gate

OR gate

XOR gate

Tri-statable buffer with enable input

Full adder

Registered delay

Multiplexer controlled by configuration bits

Multiplexer controlled by external signal

NMOS transistor

Signal controlled by configuration bit c

Chapter 1

Introduction

1.1 Field Programmable Gate Arrays (FPGAs)

The concept of programmable logic was first introduced in 1966 [1], when it was realized, that instead of relying on permanent fuses to customize a gate array, additional gates could be used to provide electrical programmability for customization. However, technology constraints limited the implementation of FPGAs until 1985 when the first commercial FPGAs were introduced. Since then, the use of FPGAs in system design has increased rapidly, prompting large investments in the field of programmable logic by the industry. The diverse demands of programmable logic, in turn, have resulted in the introduction of varied architectures targeted at specific consumer needs. To promote their architectures, device vendors supply CAD tools specific to their architectures. These tools provide a user friendly design

environment and are optimized to allow the end user to achieve short design cycles for those architectures. The introduction of FPGAs has opened new research areas in the field of CAD tool development, digital logic design and in the area of reconfigurable hardware. The term *soft hardware* is often associated with FPGAs and is indicative of the ease of implementation and high flexibility comparable to that of software. With the increasing use of FPGAs in commercial products and their potential for reconfigurable hardware, testability of FPGAs is gaining the attention of engineers and researchers.

Since their introduction, FPGAs have increased in speed and density from a few thousand gates to a few hundred thousand gates [1]. Earlier FPGAs with simple logic elements resembled Programmable Logic Devices (PLDs), and were implemented as either channelless (*sea of gates*) or channelled gate arrays as shown in Figure 1.1. The channelled archi-

Figure 1.1. Gate array architectures

ture accommodates routing resources in channels around the logic resources as shown in Figure 1.1(a), whereas the sea of gates architecture provides arrays of gates overlayed with the routing resources as shown in Figure 1.1(b). The routing resources in an FPGA are typically short and each connection may pass through several switches. In contrast, PLDs have a simple but inefficient routing architecture with every output connecting to every input through a single switch. As their densities have increased, the complexity of logic elements in the FPGAs have increased from a few gates to complex blocks containing Look Up Tables (LUTs), multiplexers and flip-flops, as opposed to PLDs which typically have a two level AND-OR logic. Devices with complex logic blocks are coarse in granularity and are implemented as channelled architectures, whereas devices with less complex blocks or fine granularity are implemented as channelless architectures.

Routing resources in a typical coarse grained device run horizontally and vertically across the device resulting in an array like structure as shown in Figure 1.2. Programmable connections are provided at the crossover points to allow interconnections between the horizontal and vertical routing resources. Programmable Logic Blocks (PLBs) are placed between the rows and columns of routing resources which allow access to the logic resources through programmable interconnect switches. The PLBs contain fundamental resources necessary to implement the desired user function; for coarse grained devices these may include LUTs, multiplexers and some sequential logic. The interconnect system formed from the routing resources can be used to implement inter-logic block, or logic block to I/O block connec-

Figure 1.2. A typical coarse grained FPGA architecture

tions. The I/O blocks are placed along the perimeter of the array of logic blocks, providing a buffered interface between external circuitry and internal logic. The I/O blocks are generally programmable and can be configured as tri-state, input, output or bidirectional pins. In addition, they provide some user accessible sequential logic resources.

Configuring a particular design or functionality onto an FPGA involves programming the logic blocks to execute the desired function, followed by programming the interconnects to route nets among the logic blocks. The programming of physical resources is achieved through setting the states of electrically controllable switches. These switches may take the form of a pass transistor or a fuse element, which in the on state serves to interconnect two physical resources, such as metal segments used for routing. The collective states of

these switches define the functionality of a device and are called the *device configuration*. In FPGAs based on Static RAM, the state of the switches are dependent on the contents of memory elements called the *configuration memory*. Thus, the device configuration is simply a stream of bits that is loaded into the configuration memory of the device at power-up. For other devices, however, special techniques may be required to program the switches. The CAD tools provided by the device vendors allow the end user to program the desired functionality onto the device without in-depth knowledge of the underlying architecture. The user generally specifies the design at a behavioral, Register Transfer Level (RTL), or schematic level. The CAD tools are then used to synthesize the design into logic modules that map onto physical resources on the device, followed by placement and routing to yield the final device configuration.

With increasing density and speeds, FPGAs are becoming viable alternatives to ASICs for low volume products. Compared to the standard ASIC design process, FPGAs have several advantages. In FPGAs design errors are comparably easy and inexpensive to rectify, especially so in the case of re-programmable FPGAs. Implementing changes incurred during debugging of a design simply involves reprogramming the device and can be done even at the system level, often without needing any changes to the system design. The ease of implementing a design helps to reduce the design cycle significantly. The short design cycle and fast debugging makes FPGAs ideal for prototyping large designs. Furthermore, the designer can iterate through design changes and their implementation many times in a

short period of time. This results in a design methodology which approaches a software development system in flexibility and ease of implementation.

1.2 Current Approaches to Testing FPGAs

New FPGA devices are highly flexible and provide a large combination of possible configurations to allow high utilization of logic and routing resources by user designs. Thus, FPGA devices, and SRAM FPGAs in particular present a high degree of testing complexity [2], which increases as the density of configurable resources increases. Traditionally, the issues of testability were of concern only to the device vendors, who guaranteed the operability of a device. However, with increasing use of SRAM FPGAs in commercial applications including reconfigurable systems, testability has become a major concern for the researchers. Very few testing techniques for FPGAs have been published to date, and these generally make use of scan test methods which have hardware overheads [3]. However, recently two methods that do not require additional circuitry cited below have been proposed for field testing of FPGAs. The first method due to C. Jordan et. al. [4] addresses the testing of incoming FPGAs to assure their integrity before employing them for prototyping designs. The proposed method takes advantage of the regular structure of FPGAs and extends the method for testing of regular arrays to testing the PLBs of SRAM FPGAs. The application of the method for the Xilinx XC3000 Configurable Logic Block (CLB) architecture has also been demonstrated. The second testing method developed by C. Stroud et. al. [5] employs a BIST

approach in the testing of SRAM FPGAs without incurring any hardware overhead. Further, C. Stroud et. al. [6] have analyzed LUT based FPGA architectures to yield a general expression for the number of test sessions required to completely test all logic blocks in an FPGA. The method is applied mainly to the testing of PLBs at the wafer, board and system level. However, the method has been shown to be applicable to Multi-Chip Modules (MCMs) as well [7].

In the approach proposed by C. Jordan et. al., the CLBs in an FPGA are configured as a single or a set of regular arrays of Processing Elements (PEs), with each PE of the array implemented using a single CLB. The array is then exercised with test vectors that test each PE in the array, thereby resulting in the testing of the CLB resources used by the PE. It follows then, that to achieve high fault coverage, the PE must use all CLB resources. An array with such a PE is an optimal array for testing purposes because it allows fault effect propagation from a faulty PE in an array to a primary output. In addition, the test vectors are propagated by each PE such that the output vectors of a PE form the input test vectors for the succeeding PE, thereby exercising each PE in the array. Moreover, the fault effect propagation characteristics, and the requirement that the PE occupy maximal resources of a CLB, involves finding an optimal regular array and deriving its test vectors for each new architecture.

The BIST approach employs specialized configurations for the CLB of an FPGA device that exercise all elements of that CLB when subjected to pseudo-exhaustive test patterns.

These test patterns are generated by a pseudo-random number generator configured using some CLBs within the same device. The outputs of the various CLBs under test are compared using Output Response Analyzers (ORAs), which are also constructed from CLBs within the same device. These ORAs simply generate a pass/fail signal at the end of a test. To test CLBs configured as pseudo-random number generators or ORAs, the above approach is applied in phases, with each phase covering previously untested CLBs. This approach, though good for fault detection, does not address fault localization.

The testing of routing resources has been addressed by Lombardi et. al. [8], which extends the diagnosis of Field Programmable Interconnect Systems (FPIS) to the interconnect system of FPGAs. The term FPIS refers to a system of nets that are interconnected through programmable switches arranged in grids. These systems can be stand-alone chips or a part of a chip such as the routing resources of an FPGA. The test strategy for detecting faults in switch matrices has been described and implemented for the switch matrix structure of the Xilinx XC4000 family of devices.

The common drawback in the approaches cited is that they only address fault detection for logic or routing resources and not fault localization. Furthermore, each of the methods address testing of a particular resource only. They do not suggest a method to integrate the testing of that resource with the testing of other resources in a device. Thus, the issue of developing an overall test strategy for all resources in an FPGA has not been addressed by any of the published approaches.

1.3 Thesis Statement

This research addresses the issue of developing an overall test strategy for SRAM FPGAs. The main contribution of this research is the development of a methodology and software necessary for accomplishing testability of SRAM FPGAs. Specifically, this thesis proposes a test methodology that meets the testability requirements of reconfigurable, fault tolerant or reparable systems based on FPGAs. The proposed methodology has been implemented and tested successfully for the Xilinx XC3000 and XC4000 devices. It is shown that the methodology is scalable to different device sizes within the same architecture and extensible to different device architectures.

1.4 Testing for Reconfigurable Systems

The reconfigurable, fault tolerant or reparable systems targetted by the proposed methodology maybe defined as those that can recover from a physical failure in a component FPGA through reconfiguration of the failed device so as to avoid faulty resources. Such systems require accurate information regarding the presence and location of faults within FPGA devices. Furthermore, they require full testing of all logic and routing resources in an FPGA regardless of the intended function of the device. A test methodology addressing the above requirements would form an important component in the implementation of these systems. Apart from the above, such a methodology can be applied to assure device integrity during

hardware development and system burn-in stages, thus providing a uniform test approach throughout the system development.

The test approach employed in this research relies on the reconfiguration feature of SRAM FPGAs to test the devices. The approach aims at applying a set of specialized device configurations to the DUT in order to test the underlying resources used by those configurations. The test configurations are optimized for the architecture of the targeted device in order to yield a high fault coverage. In addition, the test configurations are independent of the intended function of the device and include all resources for testing. The test configurations employed by the approach can be developed readily for any coarse grained SRAM FPGA architecture, hence the approach is flexible and easily extensible.

The remainder of this thesis is organized as follows: Chapter 2 describes the programming technologies of various FPGAs, and the issues of FPGA testing for reconfigurable systems. Chapter 3 describes a preliminary approach for exploring the test issues of SRAM FPGAs. Chapter 4 describes a methodology for testing the CLBs of a SRAM FPGA, and its implementation for the XC3020 device. Further, it introduces an architecture independent approach to testing CLBs, and discusses implementation of prototype software *CLB-Test*. Chapter 5 describes the application of *CLB-Test* to the XC4000 family of FPGAs and presents results from its application. Chapter 6 describes an architecture independent methodology for testing routing resources in SRAM FPGAs based on the functional level model of a device. A device model is implemented for the representative XC3020 FPGA.

The algorithms developed under the methodology are implemented, and the resulting prototype software *Routing-Test* is tested for the XC3020 device model. Chapter 7 describes the need and modifications to existing methodologies required to implement an integrated test methodology. A test environment that interfaces prototype softwares *CLB-Test* and *Routing-Test* developed in this research with the vendor tools and a generic IC tester to achieve an automated approach to testing of FPGAs is setup and described. Chapter 8 summarizes the thesis research and identifies the areas for possible future research

Chapter 2

Background

This chapter introduces the main types of programming technologies in commercial FPGAs. The architecture of an FPGA that uses SRAM based programming technology is described. It is shown that the unique feature of reconfiguration provided by SRAM FPGAs can be exploited to achieve testability and reparability. The issues in testing FPGAs considered for use in reconfigurable systems are discussed.

2.1 FPGA Programming Technologies

The programming technology of an FPGA refers to the technique by which interconnections between various physical resources are implemented. In FPGAs the interconnections are based on electrically programmable switches. Most commercial FPGAs can be classified according to their programming technologies [9] based on antifuse, floating gate or SRAM

controlled switches. The programming technology employed by an FPGA determines the characteristics of the device such as re-programmability, volatility, size and speed. The three main types of programming technologies are described below.

Antifuse devices are two terminal devices with a very high resistance in an unprogrammed state. The antifuse device can be programmed by applying a high voltage pulse (10-20 Volts) which creates a low resistance link between the two terminals. However, antifuses are generally one time programmable only. Antifuses such as those used in Actel FPGAs are built using Programmable Low Impedance Circuit Elements (PLICE) which are bidirectional when programmed. Figure 2.1 shows the cross-section of such a device. On applying a high voltage pulse across the PLICE dielectric, a low resistance path is created from the poly to the diffusion, thereby connecting two physical resources together.

Figure 2.1. A PLICE antifuse

Floating gate technology is used in EPROM/EEPROM based FPGAs such as those from Altera. Figure 2.2 shows a programmable element in which gate 2 is the floating gate. In normal operation, a high on word line turns on the transistor, pulling the bit line low and thus implementing an inverting connection. By depositing a negative charge on this gate,

Figure 2.2. A floating gate device

the threshold voltage for the transistor can be increased to a point where it never turns on. The deposited charge can be removed by exposing the gate to UV light, thus allowing re-programmability. EEPROM devices provide a mechanism to electrically remove the charge on the gate, and are thus easier to reprogram.

SRAM programming technology is the most popular of all due to its easy re-programmability and standard process technology. The switches in SRAM FPGAs are pass transistors controlled by a static memory element as shown in Figure 2.3(a). When the SRAM cell is

Figure 2.3. SRAM programming technology

programmed to a one, the pass transistor connects the two metal segments together. Similarly, multiple memory cells can be used for controlling logic elements such as multiplexers as shown in Figure 2.3(b). This kind of structure is often used for controlling data paths within the CLBs. Since programming doesn't require voltages different than the device supply voltage, SRAM FPGAs can be programmed in-circuit. This feature allows implementation of dynamically reconfigurable and reparable systems. Thus only SRAM FPGAs are considered in this thesis research. The architecture of a representative SRAM FPGA is described below.

2.2 Architecture of a Representative SRAM FPGA

Figure 2.4 shows the architecture of the Xilinx XC3020 device, which is a representative coarse grained SRAM FPGA. The logic resources in the device are arranged as an 8x8 array of Configurable Logic Blocks (CLBs). Each CLB has 9 inputs and 2 outputs, and employs a 5 input Look Up Table (LUT) to implement the user functions. The LUT which is essentially an SRAM used for implementing combinatorial logic functions. The truth table of the desired function is written into the memory, the input variables of the function are then used as the address of the memory. Thus, the output of the memory for each address emulates the output of a combinatorial logic implementation. For example, Table 2.1 shows the contents of a 3 bit LUT(8x1bit SRAM) implementing the function $f = a \cdot b + c$. The XC3020 CLB provides 2 D flip-flops to register the outputs of the LUT or to implement in-

Figure 2.4. XC3020 logic and routing resources

A	B	C	OUT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2.1. LUT implementing function $f = a \cdot b + c$

dependent sequential logic. The output multiplexers in the CLB can be used to route either LUT outputs or the flip-flop outputs to the 2 primary CLB outputs.

The routing resources in the XC3000 devices can be classified into local lines and global lines. Global lines run across the entire length and breadth of the device, whereas local lines run across the length and breadth of a single CLB. Each set of 5 horizontal or vertical local lines is terminated into a switch matrix at their intersection. Thus, each switch matrix has 20 ports corresponding to 4 sets of local lines terminating at the switch matrix. Each port can be connected to 5 other ports allowing the formation of longer nets consisting of local line segments. The connections between CLB I/O pins and routing resources are achieved through the use of Programmable Interconnect Points (PIPs). The I/O blocks, located along the periphery are user programmable and allow direct and registered data paths.

2.3 FPGA Test Requirements for Reconfigurable Systems

FPGAs have traditionally been employed for non critical applications such as glue logic on complex boards due to their low speeds and densities. Recently, FPGAs have been improving in logic density and speeds, which combined with their low cost and fast turn around times have led to their increasing use in commercial products. In addition, SRAM FPGAs provide the unique feature of reconfigurability that can be exploited to achieve reparability and fault tolerance [10], [11]. These factors have prompted research in the future use of FPGAs in applications that were previously considered too mission critical. Circuits in-

volving arrays of FPGAs in multi-chip modules are an example of such applications. In such implementations high fault tolerance and reparability are of great importance. Indeed, for applications that require reparability, the effective utilization of FPGAs is contingent on the ability to test the logic and routing integrity of the devices. To assure this integrity a means of detecting and localizing faults in the logic and routing resources of the device is required. Thus, testing must be extended beyond the verification of the functionality of a particular circuit to the total operability of the logic and routing resources of the device.

The functionality provided by an SRAM FPGA is controlled by its configuration memory. By changing the content of configuration memory it is possible to change the functionality provided by the FPGA. Conversely, the configuration memory content can be changed to achieve the same functionality through the use of different physical resources in the device. This presumes that the device has sufficient redundant physical resources to replace the failed resources. This concept forms the basis of providing fault tolerance and reparability through SRAM FPGAs. Hanchek [12] shows the applicability of this concept in both, achieving fault tolerance and reparability at the user level and achieving an increased yield during manufacturing. The exploitation of reconfigurability to achieve reparability is best illustrated through an example. Consider a system implemented using SRAM FPGAs. If one of the devices undergoes a field failure, the following steps are undertaken in a system employing reparability. First the system is brought off-line and subjected to diagnostic test patterns. This testing phase generates fault information pertaining to the physical resources

found to be faulty. Since the devices used in the system have redundant physical resources, it is possible to reconfigure the FPGA(s) to provide the same functionality as before through the use of redundant fault free resources. However, this is only possible if the testing phase generates accurate information pertaining to the location of faulty physical resources. Thus, the test methodology plays an important role in the implementation of a fault tolerant system.

FPGAs are normally vendor tested. However, the failures addressed by the proposed methodology are those that occur during integration, system burn-in or in the field. Field testing differs significantly from the manufacturer's test phase. Because the manufacturer has complete knowledge of the device architecture and physical implementation, it is possible to develop a set of configurations that when mapped onto the device, exercise all device resources. Field testing, however, is carried out by users and is based on the user model provided by the device manufacturer. The user model is an abstract representation of the device functionality, and thus the testing is limited to functional verification.

The extent of testing necessary to confirm the integrity of an FPGA device depends largely on the application the device is being used for. In an application where a fixed design is programmed on the FPGA, it is sufficient to test only those physical resources used by the design. However, for applications that require unconstrained FPGA reconfigurability, it is necessary to test all logic and routing resources. Furthermore, implementations requiring reparability, fault tolerance and reconfigurability are critically dependent on the ability to

detect and localize faults.

The ability of the diagnostic routines to provide accurate fault detection and fault localization is of significant importance. However, of equal importance is the ability of the reconfiguration system to take the above fault information into consideration while re-mapping the circuits onto the device without significant loss in performance. The loss in performance during reconfiguration results from the extra wiring delays introduced by non-optimal placement of components caused by the presence of faulty resources. Thus, performance degradation dictates that the faults be localized to a minimal possible set of physical resources so that the placement is not affected to a large extent [13], [14]. Such fine grained localization necessitates detection and localization of faults to logic as well as routing resources within a device.

With the objectives of fault detection and localization in both logic and routing resources, an architecture driven test methodology was developed for a representative SRAM FPGA, described in the next chapter.

Chapter 3

A Preliminary Test Approach for SRAM FPGAs

The problem of fault detection and location was first addressed for the Xilinx XC3000 architecture. An architecture specific test methodology was developed with the objective of identifying the issues associated with the testing of SRAM FPGAs. In addition, the setting up of a design environment for the implementation of this methodology was identified as a means of providing a framework for further enhancements to the methodology and their implementations. The XC3020 device was chosen to implement the test methodology as its small array size allowed rapid implementation of its model and algorithms based on the model, while the device architecture is sufficiently complex to highlight the testability issues of concern in coarse grained SRAM FPGAs.

Due to the large number of programmable routing resources in FPGAs, the testing of logic resources alone does not fully test the routing resources. Thus, the testing of logic and routing resources in coarse grained FPGAs are generally addressed in two separate phases. The logic resource test phase includes testing of CLBs, tri-state buffers, and I/O blocks, whereas the routing resource test phase includes testing of local lines, global lines, switch matrices and PIPs.

3.1 Testing of Logic Resources

Since the gate level descriptions of most commercial FPGAs are not available to the field test engineer, testing must be carried out at a functional level. To test the functionality of logic resources such as CLBs, they must first be programmed to execute a particular function. When programmed onto a CLB, this is called a CLB configuration. Testing of this function then results in determining the correct functionality of the utilized resources. Figure 3.1 shows the resources in a XC3020 CLB that need to be tested. Since the function used for testing may not utilize all of the resources in a CLB, it may be necessary to reprogram the same or a different function such that all resources are exercised. To reduce the number of CLB configurations, the function chosen for implementation must utilize all data paths possible in a single configuration. C. Jordan et. al. [4] proposed a method for functional testing of CLBs which employed certain regular arrays. The optimal function suggested in [4] for the XC3020 device was a “Bit Serial word Parallel” BSP multiplier. Figure 3.2(a)

Figure 3.1. XC3020 CLB architecture

shows the Processing Element (PE) of the BSP multiplier. The BSP multiplier receives two inputs: the input x arrives serially whereas the input y arrives in parallel. The serial input is broadcast to all the PEs simultaneously, and each PE receives a single bit from the parallel input. The BSP multiplier generates the output after $2n$ clock cycles, where n is the number of bits in the parallel input word.

3.1.1 A Test Method Using Regular Arrays

To test the CLBs of an FPGA, a regular array as shown in Figure 3.2(b) is mapped onto a row of CLBs, with each PE of the array implemented using a single CLB. A set of such arrays are used to cover all CLBs in the FPGA. The BSP multiplier mapped onto the FPGA is

Figure 3.2. A BSP multiplier

tested using constant propagating vectors. These test vectors regenerate themselves within the array, thus propagating a repeating sequence of test vectors through that array. A method for deriving an efficient set of such test vectors is also proposed in [4]. Since the concept of propagating test vectors is employed, it is necessary for the PE of the array to be “information lossless.” This property ensures that the test vectors and the detected fault effect is propagated from a faulty PE to the end of the array.

Fault Coverage

The CLB resources utilized by the PE of a BSP multiplier are shown in Figure 3.3, with the dark lines indicating the data paths used in the configuration. As observed from the figure, it is not possible to cover all possible modes of operation through a single configuration.

Figure 3.3. CLB resources covered by a BSP multiplier configuration

3.2 Testing of Routing Resources

During the testing of CLBs the routing resources used for interconnectivity of elements of the array are also tested. However, the detected faults cannot be localized to a specific routing resource, or to a resource within the CLB. In addition, the fault coverage of routing resources provided by the tests is too small. This section describes a method for testing the local line routing resources for the XC3020 device. The method is independent of logic resource testing and provides fault detection and location. Although the method is specific to local lines, certain switch matrix connections used to interconnect local lines are also tested. The faults in the switch matrices are detected as faults in adjoining local line segments.

The fault set covered by the method consists of stuck at logic 1 (sa1), stuck at logic 0

(sa0) and stuck open (so) faults. The detection of the above faults is carried out by configuring a path consisting of untested local line segments between two I/O pins and then performing a continuity test on that path. To test all local line segments it is necessary to configure and test such paths in an incremental manner. Before the paths can be exercised, the configured paths need to be converted to a configuration bit-stream and down loaded to the DUT. Furthermore, the results from each test need to be analyzed to determine the fault status of the used resources. The fault status determines the local line segments to be included in the next set of paths. This time consuming process was automated and is described below.

Local Lines Test Automation

The local line test automation problem consists of determining a set of untested local line segments that can be configured to form a path between two I/O pins. In addition, it involves localizing the fault observed at the output pin to one or more faulty segments in a path. In other words, the second part involves assigning each segment its appropriate fault status. The two parts, path generation and fault status updating, are dependent on one another. Thus, the process is solved in an incremental manner, where each stage involves generating one or more paths and updating the fault status of each segment in that path by examining the results. Furthermore, the procedure must accomplish complete testing in a minimal number of iterations to reduce test time.

The test procedure takes advantage of the grid like architecture of the local lines. As shown in Figure 2.4, the local lines are arranged in 9 rows and columns with switch matrices at the intersections, with each row and column segmented into 8 parts. For simplicity, only one of the group of 5 segments shown in Figure 2.4 are considered at a time. The above structure is modeled as a pair of 8x9 matrices each representing rows and columns of local lines, with the matrix row indices representing local line rows or columns and the matrix column indices representing the segments.

The test automation algorithm has 5 main components. Partitioning and search algorithms are used to solve the problem of generating appropriate paths. A routing generator is used for converting the paths to the final configuration for down loading to the DUT. The fault updating algorithm is used for updating the fault status of local line segments after testing. The algorithm operates on a pair of fault matrices which model the local lines. These components of the algorithm are described below.

The Fault Matrices model the local lines in the manner described above. Each element in

this pair of matrices stores the fault status of its corresponding local line segment.

The element can take four possible values representing the type of fault assigned to the segment, namely: sa1, sa0, so (stuck open), or fault free. The matrices are updated after each test to reflect the faults detected and localized to particular segments.

The Partitioning Algorithm makes use of the fault matrices to configure appropriate paths for detecting and localizing faults so as to minimize configurations.

The Search Algorithm finds a fault free path from input pin to the starting point of the paths configured for testing.

The Routing Generator creates a script for generating a design for the above configurations, which can be input to the vendor tools to give a final configuration bit stream that can be down loaded to the device.

The Fault Updating Algorithm uses the outputs of the previous test to detect and trace the faults to their origin and update the fault matrices to reflect the new status of local line segments.

To test the algorithm a fault simulator was written that could simulate the effect of injected faults on the outputs of the device under test (DUT). The fault simulator, used the “distance” of the injected faults from the output to predict which fault would dominate at any particular output. With the help of this simulator the algorithm was tested with sets of injected faults to determine its accuracy.

Partitioning Algorithm Partitioning is a crucial part of the test procedure, and it involves isolating a section of the local line grid within which untested segments are concentrated. Partitioning helps in the proper configuring of paths that can significantly reduce the number of device configurations required to achieve high fault coverage. The pseudo-code for the algorithm is presented below.

procedure name: partitioning

begin

*Search horizontal local line fault matrix for a faulty segment that
has not been included in a previous path*

Store the matrix indices for the above segment

*Search vertical local line fault matrix for a faulty segment that
has not been included in a previous path*

Store the matrix indices for the above segment

The matrix indices from above steps constitute the pivot point

Generate a grid of paths from pivot point to output pins

end

The matrix indices that form the pivot point indicate the row and column in the local line grid from which point onwards the local lines have been untested. This point is then considered a corner point of the new partition in which the grid of paths is configured for testing. A typical grid generated during a test process is shown in Figure 3.4.

Search Algorithm This algorithm reads the updated fault matrices and finds a fault free path from the input pin to the pivot point found by the partitioning algorithm. Often no fault

Figure 3.4. A typical grid generated during a test

free path exists from the desired input pin to the pivot point due to the existence of spurious faulty states generated while the test is in progress. In such a case the user can input a path manually or allow the algorithm to proceed with a default path. Finding a fault free path to the pivot point is fundamental in reducing the number of iterations in order to achieve high fault coverage. The pseudo-code for the algorithm is presented below:

procedure name: search_path

begin

while row number < pivot_row number and col number < pivot_col number **do**

col: Proceed along columns or vertical lines matrix

Store the vertical matrix indices

if segment corresponding to current indices is fault free **then**

add vertical local line segments to path until

a faulty segment is encountered or the pivot row is reached

else if current indices are already stored **then**

fall back to previous indices and goto row:

else store current indices and goto row:

row: Proceed along the rows or horizontal lines matrix

if segment corresponding to current indices is fault free **then**

add horizontal local line segments to path until

```
        a faulty segment is encountered or the pivot column is reached
    else if current indices are already stored then
        fall back to previous indices and goto col:
    else store current indices and goto col:
    Output fault free segments added to the path
end
```

Figure 3.4 shows a typical fault free path generated during the testing process.

Results

With the help of the fault simulator, the algorithm was tested for various sets of injected faults. It was found that though the algorithm did not miss any faults it reported some spurious faults, and the type of fault was not reliably determined. The following is an example set of fault matrices for horizontal(row) and vertical(col) local lines. The fault matrices on the left contain injected faults, whereas those on the right contain the faults detected by the algorithm.

[illegible]

- 3 fault free state
- 2 stuck open state
- 1 stuck at 1 state
- 0 stuck at 0 state

The spurious faults that were detected by this algorithm resulted from the fact that a segment immediately preceded by a faulty segment in a row and column matrix is not controllable and thus cannot be tested.

3.3 Summary

This chapter described the implementation of an approach using regular arrays for testing CLBs of XC3000 devices. In addition, a method for testing local line routing resources was proposed and implemented for the XC3020 device. The drawbacks in the test approaches for both resource types were identified. The regular array test method for CLBs involves finding a new array for each CLB architecture with the appropriate fault effect propagation characteristics. Furthermore, the method proved not to be modular as an entire array of CLBs needed to be configured in order to take advantage of this method. The routing resource test approach based on the local line matrices model was found to be inefficient, resulting in detection of spurious faults.

Chapter 4

A Methodology for Testing Logic

Resources of SRAM FPGAs

This chapter presents an alternate approach to testing of CLBs that employs specialized configurations instead of regular arrays to configure CLBs for testing. Since the specialized configurations would be prepared specifically for testing the CLB, the total number of configurations required to cover the entire CLB would be fewer as compared to the regular array test approach. The specialized CLB test configurations can be readily generated for any CLB architecture, and are not required to have special properties such as fault effect propagation. The test vectors for these configurations are simpler than those for regular arrays and are relatively easier to generate. In addition, it has been shown by Kwiat et. al. [15] that test vectors generated for unmapped circuits may not yield the expected fault coverage after

mapping the circuits, due to the effects of technology mapping. In this respect, the use of specialized configurations is advantageous as the fault coverage remains unchanged after mapping. Furthermore, the test scheme could take advantage of the regular structure of FPGAs, resulting in a modular approach towards testing multiple CLBs in parallel. With the aim of implementing such an approach for the XC3000 FPGAs, the specialized test configurations for the XC3000 CLB were generated as described below.

4.1 CLB Test Configurations for XC3000 FPGAs

The specialized CLB configurations are specific to each architecture and need to be developed before the test methodology can be applied to a device. The following sections describe the CLB configurations for XC3000 FPGAs. A fault model is described for LUT based CLBs that supports functional testing of CLBs. Based on the fault model, test configurations and test vectors for all CLB primitives are developed. Test configurations and test vectors for the entire CLB are described that apply the test configurations developed for each CLB primitive.

4.1.1 A Fault Model for LUT Based CLBs

Fault models and test generation methods for functional level testing [16] generally use a two level AND/OR representation of the functional modules. The stuck-at fault model can

be applied to this gate-level description and test vectors can be generated with Automatic Test Pattern Generator (ATPG) software. The problem with this approach is that it is non-deterministic and doesn't lead to 100% fault coverage. To overcome this a variant of the above model is used, where all possible changes in the truth table of the function are considered in order to accommodate all possible implementations. This approach is used for logic modules with a low number of inputs due to their exponential complexity.

These fault models or variations thereof work well for fine grained architectures with few inputs and relatively simple logic modules. For CLBs in devices with coarse grained architectures, such as those of the XC3000 and XC4000 families [17], such fault models are not suitable. These CLBs contain relatively complex modules that are both combinatorial and sequential, and the details of their physical implementation are not available. Thus, a two-level representation of these modules may give an inaccurate fault coverage and can be misleading. In such cases a higher fault coverage can be achieved by treating complex functional units such as flip-flops and multiplexers as primitives [18], and developing fault models for these primitives.

Functional Fault Model for XC3000 CLBs

The model depicted in Fig. 4.1 consists of only three primitives corresponding to multiplexers, D-registers, and an OR gate. Hayes [18] proposed a functional fault model suitable for circuits employing high level primitives such as the ones previously referred to. This func-

Figure 4.1. XC3000 CLB with functionality defined in terms of configuration bits

tional fault model is presented below.

If F is a function implemented by a combinatorial functional block B , then any physical fault asserts itself as a functional fault, thereby changing F to F^* . If B has S states, then the modified block B^* has S^* states and S^* should not be greater than S . This condition is true as long as a fault doesn't convert a combinatorial logic module to a sequential module. To cover all functional faults in the block it is necessary and sufficient to apply all 2^n input vectors, where n is the number of inputs.

If B is a sequential circuit, then the functional faults covered by the fault model include

all faults that do not cause an increase in the number of states in the sequential circuit. This restriction simplifies the derivation of an input-output sequence that distinguishes a fault free circuit from a faulty circuit [19]. Thus, the above fault model imposes the following restrictions: 1. for combinatorial circuits, faults should not change the circuit to a sequential one; 2. for sequential circuits, faults should not cause an increase in the number of states; 3. this model assumes that only one module in the circuit is faulty at any time. The next section describes the test vectors and configurations necessary to conduct the functional level testing of the primitives.

4.1.2 Test Configurations and Test Vectors for CLB Primitives

A CLB test configuration is defined as a circuit which, if mapped onto the CLB, will allow one or more primitives to be exercised with a set of test vectors. A CLB can be mapped with a test circuit by configuring the multiplexers that control the data path within the CLB. Figure 4.1 shows the CLB architecture, with programmable connections depicted with small triangles. These programmable connections are inputs to multiplexers and are thus mutually exclusive connections. To aid the description of test configurations developed herein, a set of signals E_{1-9} are introduced. These signals are control inputs to the multiplexers and represent bits in the configuration memory. Thus, any circuit that can be mapped onto the CLB can be described through the states of signals E_{1-9} .

The primitives that need to be tested in a CLB as depicted by Figure 4.1 are: a function

generator block, D-registers, multiplexers and an OR gate. To test these primitives, it is necessary to develop test vectors for each of them. Since the function generator block can be configured in three modes, some internal configurations need to be developed for the function generator primitive. The test vectors, and test configurations in the case of the function generator primitive, must be applied to each primitive to test it and verify its functionality. The test configurations and test vectors for each primitive are discussed next followed by a discussion of test configurations for the entire CLB.

Function Generator Block

Figure 4.2 shows the three modes that the function generator block can be configured in, namely F, FG, and FGM.. The functions of 4-5 variables indicated in Figure 4.2 are implemented with a pair of 16 x 1 bit look-up tables represented by letters 'F' and 'G'. The truth table of the desired function is stored in the LUT. The input variables are used to decode the address of the corresponding truth table entry in the LUT, thus, effectively implementing the desired function. In the XC3000 family of devices, the LUT memory can be written only during configuration of the device, hence, the LUT memory is considered as part of configuration memory. Its testing is conducted through the configuration memory read back facility provided in the device. Thus the LUTs can be replaced by 16:1 multiplexers, which represent the decoding function in the LUT. Figure 4.3 shows the resulting functional level description of the function generator block in terms of configuration bits. The configu-

Figure 4.2. Combinatorial logic modes FG, FGM and F implemented by a look-up table

ration bits c_{1-12} represent the control signals of the multiplexers shown in the figure. The states of these configuration bits fully describe any function generator block mode and are used as aids in describing this model's test configurations. Table 4.1 shows the relationship between the three modes of the function generator block and the configuration bits. This schematic does not have to accurately represent the physical implementation, as the faults

Mode	Configuration bits
FG	$c_1 = 0, c_2 = 0, c_{3-12}$ user settable
F	$c_1 = 1, c_2 = 1, c_{11,12} = 1, c_{3-10}$ user settable
FGM	$c_1 = 1, c_2 = 0, c_{11,12} = 1, c_{3-10}$ user settable

Table 4.1. Function generator modes

Figure 4.3. Functional representation of look-up tables in terms of configuration bits

covered are functional rather than gate level faults. However, the schematic does show all logic modules in the function generator block which need to be tested for functional faults.

From Figure 4.3 we see that the function generator block contains only combinatorial logic modules, namely, multiplexers. To test the multiplexers completely, each input of the multiplexer must be selected by setting appropriate control bits. This is not a problem for multiplexers other than F and G in Figure 4.3. All of these multiplexers have 2 or 3 possible control input combinations and their data inputs can be controlled through primary inputs with an appropriate configuration. However, multiplexers F and G have up to 16 possible control input combinations and their data inputs are actually configuration bits. To exercise

different data inputs the CLB has to be reconfigured. Thus, to test multiplexers F and G, with a minimal number of configurations, a checkerboard pattern is loaded into the LUT memory, such that, if the control inputs are exercised with a Gray code sequence, the checkerboard pattern is read out at the output. This ensures that for exactly one change in any control input there will be a corresponding change in the output state, thus isolating any faulty input combination. To completely test the two multiplexers the sequence has to be repeated with the inverted checkerboard pattern as well. The input test vectors and expected outputs are shown in Table 4.2. The functions thus implemented are given as:

$$F = \overline{A}\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}\overline{D} + A\overline{B}C\overline{D} + \overline{A}B\overline{C}D + A\overline{B}CD + \overline{A}BC\overline{D} + A\overline{B}C\overline{D}$$

$$\overline{F} = \overline{A}\overline{B}\overline{C}\overline{D} + A\overline{B}\overline{C}\overline{D} + \overline{A}B\overline{C}\overline{D} + A\overline{B}C\overline{D} + \overline{A}BC\overline{D} + A\overline{B}CD + \overline{A}BC\overline{D} + ABC\overline{D}$$

Before the LUTs can be exercised with the above test vectors, the data path to their control inputs through muxes 1 to 7 must be configured appropriately. Similarly, to observe the outputs of the LUTs, multiplexers 8 to 10 must be configured properly as well. To reduce the total number of configurations, these multiplexers are configured such that all three modes of the function generator block are implemented at least once. The test vectors for LUTs also test the multiplexers in the data path. Table 4.3 shows the 6 configurations required to completely test the function generator block shown in Figure 4.3. This number could have

A	B	C	D	F/G
0	0	0	0	1
0	0	0	1	0
0	0	1	1	1
0	0	1	0	0
0	1	1	0	1
0	1	0	0	0
0	1	0	1	1
0	1	1	1	0
1	1	1	1	1
1	1	1	0	0
1	1	0	0	1
1	1	0	1	0
1	0	0	1	1
1	0	1	1	0
1	0	1	0	1
1	0	0	0	0

Table 4.2. Test vectors and expected outputs for multiplexers F and G

Confgs	Inputs				c ₁	c ₂	c ₃	c ₄	c ₅	c ₆	c ₇	c ₈	c ₉	c ₁₀	c ₁₁	c ₁₂
FG	A	B	C	E	0	0	0	0	0	0	0	0	0	0	0	0
FGM(1)	A	Qx	C	D	1	0	0	1	0	0	0	1	0	0	1	1
FGM(2)	A	Qy	C	D	1	0	1	0	0	0	1	0	0	0	1	1
FGM(3)	A	B	Qx	D	1	0	0	0	0	1	0	0	0	1	1	1
FGM(4)	A	B	Qy	D	1	0	0	0	1	0	0	0	1	0	1	1
F	A	B	C	D	1	1	0	0	0	0	0	0	0	0	1	1

Table 4.3. Test configurations for a function generator block

Present State	Inputs (D,RD)		Next State	Output
0	0	0	0	0
0	0	1	0	0
0	1	1	0	0
0	1	0	1	1
1	0	0	0	0
1	0	1	0	0
1	1	1	0	0
1	1	0	1	1

Table 4.4. Transition table for D-registers

been reduced to 4 if inputs Q_x and Q_y were independently controllable. However, this is not possible without overly complicating the configurations and test vectors.

Multiplexers and D Flip-Flops

Apart from the function generator block the two primitives of interest are multiplexers and D-registers. The testing of multiplexers is conducted in the same manner as are the function generator block's multiplexer primitives. For testing the D-registers the clock circuitry is assumed to be fault free, an assumption commonly made for sequential circuit testing [18]. Any fault in clock circuitry will assert itself as a stuck-at type fault at the output of the register. However, a fault in the clock circuitry is not distinguishable from a fault in the register or at its input. The transition table for D-registers is given in Table 4.4. As the registers can be initialized to state 0 from any state, the inputs in the transition table constitute a checking sequence that will test the register completely, where, a checking sequence is defined as an

input-output vector pair that distinguishes a circuit from all derivatives of that circuit with a fault present in them. While this checking sequence is not guaranteed to be minimal, for a small sequential circuit the overhead is minimal.

4.1.3 XC3000 CLB Test Configurations

The configurations for the CLB are generated with the aim of applying the test vectors and configurations developed previously to each primitive while keeping the total number of CLB configurations to a minimum. The principal guidelines in implementing CLB test configurations are as follows:

1. The CLB test configuration C_n should be such that, for each primitive n with test vector T_n , there should exist a corresponding test set T_n^* that will allow T_n to be applied to that module's inputs by applying T_n^* to the primary inputs.
2. For each test set applied to a primitive, the test results at the outputs of the primitive should be propagated to the CLB outputs.
3. All primitives must be tested completely with a minimal number of configurations.

The first and second objectives imply that the inputs of the primitive under test should be controllable from primary inputs and the outputs should be observable at primary outputs.

As previously indicated the function generator block requires 6 configurations for complete testability. This imposes a lower limit on the total number of configurations. From

CLB Config	F.G. Config	F.G. I/Ps				Primary I/Ps				
C_1	FG(1)	A	B	C	E	-	-	EC	K	RD
C_2	FG(2)	A	B	C	E	-	-	EC	K	RD
C_3	FGM(1)	A	Q _x	C	D	E	DI	-	K	RD
C_4	FGM(2)	A	Q _y	C	D	E	DI	EC	K	RD
C_5	FGM(3)	A	B	Q _x	D	E	DI	EC	K	RD
C_6	FGM(4)	A	B	Q _y	D	E	DI	EC	K	RD
C_7	F	A	B	C	D	E	-	-	-	-
CLB Config	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	
C_1	0	0	0	0	0	0	0	0	0	
C_2	0	0	1	0	1	0	0	0	0	
C_3	1	1	0	1	0	1	1	0	1	
C_4	0	1	0	1	0	1	0	0	1	
C_5	1	0	0	1	0	1	0	0	1	
C_6	1	1	0	1	0	1	0	0	1	
C_7	1	1	X	X	X	X	1	1	1	

Table 4.5. CLB test configurations defined in terms of CLB inputs and configuration bits E1-9

Figure 4.1 it can be seen that multiplexers 11 and 12 require 3 configurations each and all other primitives can be tested in two configurations. Thus, the CLB configurations are developed such that the configurations of each primitive overlap one another. The resulting CLB configurations are shown in Table 4.5 in terms of the configuration bits E1 to E9. The test vectors employed during each configuration are included in Appendix A.

CLB configurations C_1 , and C_3 to C_7 shown in Table 4.5 exercise all 6 function generator block test configurations (F, FG, FGM(1-4)). Configuration C_2 is the only additional configuration. It is required to allow test vectors to be applied to inputs 'F' and 'G' of multiplexers 11 and 12. These inputs can be exercised only when the function generator block

is in the FG mode. Since these inputs of multiplexers 11 and 12 are exercised during two multiplexer configurations, an additional FG mode is required from the function generator block. From the columns for E3, E4 (Mux 11) and E5, E6 (Mux 12) in Table 4.5, it can be seen that all 3 modes of both multiplexers are exercised in CLB configurations C1, C2 and C3.

Configurations C4, C5, and C6 overlap the testing of the function generator block, with that of D-registers X and Y. During these configurations the test vectors are applied in two steps. During step 1 test vectors are applied to the function generator block and during step 2 test vectors are applied to one of the two D-registers. For example, since, in C4, FF Y is used indirectly to apply test vectors to input Qy of the function generator block during step 1, it cannot be tested during this configuration. However, FF X can be exercised in step 2. The two steps are required because inputs Qx and Qy to function generator block and D-register inputs are controllable through a single primary input (DI). The test configurations of other primitives are overlapped with those of the D-registers and function generator block during testing in configuration modes C3 to C7.

4.2 A Test Approach for Multiple CLBs for XC3000

FPGAs

The smallest device in the XC3000 family is the XC3020 device which has 64 CLBs; the largest is the XC3195 with 484 CLBs. Therefore, to minimize the overall test requirements, a method is required that will enable the simultaneous testing of a large number of CLBs.

The method described here is specific to the XC3000 FPGAs as it makes use of some of the architectural features in the XC3000 devices to allow all CLBs to be tested in parallel. In this method a design file, which incorporates a maximum number of CLBs, is created manually and stored for each test configuration of the CLB. The maximum number of CLBs is governed by the number of nets that can be accommodated by the routing architecture. The number of nets is in turn dependent on the number of signals needed per CLB. For the XC3020 device with each CLB test configuration using 7 inputs and 2 outputs, with 2 pairs of identical inputs, a routable design including all 64 CLBs was achieved. Since there are 7 test configurations for a CLB, 7 such design files need to be created and stored.

From Table 4.5 we see that for any configuration required to test the CLB, at most 8 CLB inputs are used. However, at least two of the F.G inputs are identical, thus requiring only 7 device I/O pins. These 7 input signals are routed to inputs of all CLBs. To observe the outputs for these test vectors the outputs of each CLB must be routed to a primary output, requiring a very large number of I/Os. To reduce the I/O overhead the tri-state buffers

and horizontal long lines of the device are formed into a multiplexer. Each CLB has two outputs, X and Y, with each output routable to a tri-state buffer. Two horizontal long lines connect the outputs of each tri-state buffer for each row of CLBs. Thus two wired-AND multiplexers can be formed for each output X and Y. This setup is shown in Figure 4.4 for one row of CLBs. The enable inputs of the buffers for each column of CLBs are controlled

Figure 4.4. A schematic representation of a configuration for testing multiple CLBs

by a single enable input, thus requiring an additional 8 inputs for the XC3020 device which has 8 rows x 8 columns. For the XC3020 device, which has 8 rows, there are 16 such multiplexers. The I/O pin requirement is thus reduced to sixteen. However, this requirement can be reduced further by routing the outputs of the multiplexers to a pair of shift registers, one for each output X and Y. The shift registers are formed out of flip-flops provided in the I/O

blocks. These I/O pins don't need to be connected externally as only the registers are used. The outputs of these shift registers are routed to primary outputs, reducing I/O pin requirement to only two pins. In principal the setup shown in Figure 4.4 emulates boundary scan. Figure 4.5 shows the implementation of this strategy for the XC3020 device. The shaded I/O blocks on the left hand side indicate the I/O pins used to implement the shift registers.

Figure 4.5. Implementation of architecture dependent approach

4.2.1 Test Operation

The testing is carried out by simultaneously applying test vectors to all CLB. The outputs of the CLBs are bused onto the horizontal long lines through tri-state buffers one column at a time, as shown in Figure 4.4. The data on the horizontal long lines is loaded in parallel into the X and Y registers whose contents are then shifted out serially, in 8 clock cycles, through the primary outputs for comparison. As the XC3020 consists of an 8x8 array of CLBs it takes 8 such cycles to observe the outputs of all 64 CLBs for every test vector applied to the device. In order to completely test a CLB, it must be exercised with all 7 CLB configurations developed previously. Thus to test the entire device, 7 device configurations are generated and the 8 test cycles are repeated for each of the device configurations.

In practice, the number of device configurations may be greater due to the difficulties encountered by the place and route CAD tools in routing the netlists of the given device configurations. The device configuration shown in Figure 4.5 utilizes 7 of the 9 input pins on each CLB and both output pins. However, 2 pairs of input pins share the same test vectors, thus reducing the I/O requirements for inputs to 5 pins. While device configurations requiring only 5 inputs can normally be routed, with 7 input pins committed, the NeoCAD FPGA place and route CAD tool utilized, failed to route the entire device configuration due to shortage of routing resources. To overcome the routing limitations, only alternate rows or columns of CLBs are included in device configurations with greater than 5 inputs. This modification doubles the number of design files that need to be created and stored to a total

of 14.

With this method, maximum usage of I/Os is achieved while keeping to a minimum the number of configurations or design files. Furthermore, the test vector length remains constant, irrespective of device size and the number of CLBs, and the same vectors can be applied to all CLBs in parallel. However, with an increase in device size, the number of cycles and the length of each cycle increase. The primary limitation of this method is that it is architecture dependent and inflexible to any kind of change. Since design files are created manually beforehand, this method cannot take advantage of scalability and regular structure of devices within a particular family. Where scalability refers to the applicability of the method to larger devices within the same architecture without requiring any modifications. For example, the design files created for XC3020 cannot be reused for larger devices instead, new ones have to be created and stored even though only the number of CLBs has increased, while there is no change in the architecture.

4.3 An Architecture Independent Methodology for Testing Multiple CLBs

Specialized CLB test configurations can be developed for any SRAM FPGA device for which the CLB architecture is known. However, to effectively test multiple CLBs for various FPGA devices, it is necessary to develop an architecture independent approach for con-

figuring the CLBs with the test configurations. This section describes an architecture independent methodology for testing multiple CLBs. The test methodology incorporates constraints on factors such as the number of available I/Os and of routable signals on the implementation of test configurations for entire device. These features make the test methodology suitable for in-circuit testing of FPGA devices. Furthermore, it allows the test strategy to be applicable at a number of levels in system design, including wafer, package and circuit board testing. The proposed methodology overcomes the limitations of the method described in the previous section by allowing on the fly generation of device configurations that exercise multiple CLBs. The algorithms employed by the methodology are independent of the device architecture and therefore support scalability within the same architecture, as well as extensibility to devices with different architectures. The applicability of the approach to newer architectures is demonstrated in Chapter 5 for the XC4000 devices.

The proposed methodology employs specialized test configurations optimized for a specific CLB architecture in an architecture independent approach for testing multiple CLBs. It incorporates algorithms that operate on the CLB configurations and generate netlists for entire device configurations that exercise multiple CLBs. The methodology achieves independence from the underlying architecture by separating the architecture independent algorithms that generate the device configuration netlists, from the architecture specific routines that process the netlist. These routines constitute a router that operates on the given netlist. The implementation of this methodology allows the use of the router provided by vendor

CAD tools or of a user developed router. The algorithm for generating the device configuration netlists forms the basis of this methodology and its operation is described below.

The input to the algorithm is a list of I/O pins available for use in the design and a configuration file. The configuration file contains all configurations required to test a single CLB. The CLB test configurations are dependent on the family of devices and are prepared manually as described in previous sections. The main objective of the algorithm is to configure as many CLBs as possible with each CLB test configuration, without running short of routing resources or I/O pins. To generate the device configurations for a given CLB test configuration, the algorithm decides on the number of inputs required for testing, which is equal to the number of inputs used in that particular CLB test configuration. It then assigns available I/O pins to these inputs. An iterative process is then initiated to decide how many CLBs can be included without running short of routing resources or I/O pins. The upper limit on the number of CLBs that can be included is placed by the number of available output pins. For example, in the XC3000 FPGAs each CLB has a pair of outputs, with the result that the number of CLBs that can be included is half the number of available output pins. The next step is to generate a netlist of source destination pairs for routing the primary inputs to CLB inputs and the CLB outputs to primary outputs. This is done by taking each primary input and assigning to it an input of each CLB in the device such that all test vectors can be applied to each CLB. Similarly, the output of each CLB is assigned an available output pin as a destination.

The netlist is provided to the router which routes the design. At this point it is possible to run out of routing resources, and part of the design may remain unrouted. In such a case the algorithm then decreases the number of CLBs and repeats the above process until a design is completely routed. The algorithm applies each CLB test configuration to the CLBs included in the final design. The final design is converted to a configuration bit-stream suitable for downloading to the device using vendor supplied CAD tools. The above steps are repeated to include remaining CLBs in the next design for the same CLB configuration. The complete cycle is repeated until each CLB test configuration has been applied to every CLB on the device.

The major drawback of this method is that it requires a larger number of configurations. This, then, is the tradeoff for achieving greater flexibility and reusability of algorithms. However, this increase in the number of configurations does not impact significantly on the overall storage requirements as configurations for only a single CLB need to be stored. This method is particularly suited for devices with boundary scan. In such devices the status of I/O pins can be read through the Test Access Port (TAP), thus simplifying testing.

4.3.1 Implementation of Architecture Independent Approach

The algorithms developed under the architecture independent approach were implemented as a prototype software *CLB-Test*. However, for generating the final device configurations,

a router for the targetted architecture is required. For the purposes of this implementation the router provided by the NeoCAD FPGA tools was used. These tools allow editing of a design through the use of scripts. This feature was used to interface *CLB-Test* and the router provided by the CAD tools. *CLB-Test* was tested on two devices of XC3000 architecture: the XC3020 device, which is the smallest device in the XC3000 series with 64 CLBs, and the XC3195, which is the largest device in the XC3000 series with 484 CLBs. The typical operation of generating the XC3020 device configurations using *CLB-Test* interfaced with the CAD tools is described below.

The CLB configuration C_1 defined in Table 4.5 was chosen for developing the device configuration. The above configuration uses 7 inputs: A, B, C, E, EC, K, RD, and two outputs: X, Y. Thus, 7 I/Os for inputs and 2 I/Os for outputs per configured CLB were required. Since the XC3020 device has a maximum of 64 available user I/Os, the maximum number of CLBs that could be configured with C_1 at a time was limited to 28. Thus, *CLB-Test* aimed at placing 28 CLBs and routing their inputs and outputs. The CLBs whose I/Os could not be routed were removed from the device configuration. Each succeeding configuration tried to place 28 CLBs, which included those that couldn't be routed in the previous configuration. For this implementation, the CLBs to be placed were chosen randomly for each configuration. Table 4.6 shows the data obtained from applying *CLB-Test* to the XC3020 device. Two conditions were imposed on *CLB-Test*: the availability of all device I/Os for test purposes and a limitation of available I/Os to 48. The table shows the number of CLBs actually exer-

cised by each device configuration generated by *CLB-Test*. Table 4.7 shows the results for

No. of CLBs	64	
Total I/Os	64	
I/Os available	64	48
CLB configuration I/Os	7/2	
Max. Configurable CLBs per device configuration	28	20
Min. Configurations possible	3	4
CLBs Placed per Configuration		
Configuration No. 1	22	20
Configuration No. 2	24	20
Configuration No. 3	18	18
Configuration No. 4	-	6

Table 4.6. Results for dynamic generation of XC3020 device configurations with 64 and 48 available I/Os

the XC3195 device with all I/Os available for test purposes. Figure 4.6 shows device configuration no. 1 for the XC3020 device using 22 CLBs. As observed from the tables, the number of CLBs that can actually be configured at a time are much less than the maximum possible. This results from the fact that the routing resources cannot accommodate the large number of nets required for the device configuration. However, the decrease in configured CLBs doesn't affect the total number of device configurations significantly. The XC3195 required only one additional device configuration, whereas, the total number for XC3020 remained unaffected. Since the XC3000 CLB requires 7 test configurations, the total number of device test configurations required for the XC3195 are 7. Thus, for the XC3020 device, with 16 I/Os prohibited (48 available), the total device configurations required to test all

Figure 4.6. An example XC3020 device configuration

No. of CLBs	484
Total I/Os	176
I/Os available	176
CLB configuration I/Os	7/2
Max. configurable CLBs per device configuration	84
Min. configurations possible	6
CLBs Placed per Configuration	
Configuration No. 1	69
Configuration No. 2	65
Configuration No. 3	70
Configuration No. 4	73
Configuration No. 5	67
Configuration No. 6	74
Configuration No. 7	66

Table 4.7. Results for dynamic generation of XC3195 device configurations

CLBs using this method are 28 as opposed to 7 using the architecture dependent method.

For the XC3195 the total number of device configurations required is 49.

4.4 Summary

This chapter introduced a methodology for testing CLBs of SRAM FPGAs using specialized test configurations. The development of the CLB test configurations was described for the XC3000 FPGAs. A method for exercising multiple CLBs was described. This method was specific to the XC3000 architecture and provided full fault coverage using the same number of device configurations as the number of CLB configurations. However, the method was inflexible and had an overhead of large storage requirements. An architecture indepen-

dent approach for testing multiple CLBs was introduced that overcame the limitations of the previous method. The new method achieved high flexibility through the dynamic generation of device configurations. The scalability of this approach was demonstrated through its application to the XC3020 and the XC3195 devices.

The following chapter demonstrates the applicability of the approach to different architectures by implementing the method for XC4000 FPGAs.

Chapter 5

The Application of CLB Test

Methodology to XC4000 FPGAs

The XC4000 CLB provides a number of enhancements over the XC3000 CLB that can cater to more complex applications. As compared to XC3000 device CLBs, it provides greater logic density with an additional LUT and flip-flops, and greater flexibility with additional multiplexers for configuring data paths within the CLB. The LUTs in the XC4000 CLB can be configured as RAM modules in addition to their normal functionality. In addition, the XC4000 CLB introduces a dedicated carry logic block targeted for applications requiring fast arithmetic operations. The large number of additional configuration modes in the XC4000 CLB increases the number of test configurations required to cover all possible functional faults. This increases the complexity of test configurations and the number of test vec-

tors significantly. This chapter describes the application of CLB test methodology described in Section 4.3 to the XC4000 FPGA CLBs. The implementation of the test methodology for a device requires the development of CLB test configurations for that device. A minimal set of test configurations that cover all functional faults in a XC4000 CLB is described in the following sections.

5.1 CLB Test Configurations for XC4000 FPGAs

5.1.1 A Functional Fault Model for XC4000 CLBs

The test configurations are derived using the functional fault model introduced in section 4.1.1. Since the primitives of the XC4000 CLB are the same as those of XC3000 CLBs, the fault model used for XC3000 CLBs can be extended to XC4000 CLBs. The restrictions imposed by the functional fault model are summarized here:

1. For combinational circuits, faults should not change the circuit to a sequential circuit.
2. For sequential circuits, faults should not cause an increase in the number of states.
3. This model assumes that only one module in the circuit is faulty at any time.

The above fault model allows the primitives to be tested at a functional level. Test configurations and vectors for exercising the primitives of the CLB are developed in the following sections.

5.1.2 Test Configurations and Test Vectors for CLB Primitives

Figure 5.1 shows the internal architecture of the XC4000 CLB. The CLB is composed of

Figure 5.1. XC4000 CLB architecture

4 main primitives, namely, LUTs, flip-flops, multiplexers and a Carry Logic (C.L.) block. Testing of the CLB involves two steps. During first step the CLB is configured to allow each of the above primitives to be controllable and observable. It is possible that multiple configurations will be required to cover all primitives. During the second step the configurations are exercised with appropriate test vectors. The multiplexer primitives control the data path within the CLB, and as a result vectors used for testing other primitives exercise the multiplexers included in their data paths. It is thus possible to fully test the functionality

of all multiplexers by developing the CLB configurations such that they exercise all possible data paths. The other primitives, however, require the generation of special test vectors to achieve complete test coverage. The following sections describe the test vectors for LUTs and Carry Logic primitives. The testing of D flip-flops is conducted in same manner as described in section 4.1.2.

LUTs

The XC4000 CLB has 3 LUTs: F, G and H. LUTs F and G each have 4 inputs and can be configured for 4 functions as follows: 1. 4 input logic function; 2. 16x1bit RAM module; 3. 16x1bit ROM module; 4. A special function to aid the carry logic in implementing an arithmetic function. LUT H has only one configuration mode in which it combines the outputs of the F and G LUTs with another input to give a function of up to 9 input variables.

Since the F and G blocks are essentially a 16x1bit memory, testing them in RAM configuration mode covers the other 3 functions as well. Furthermore, the XC4000 CLB allows the F and G modules to be configured as 32x1bit RAM modules with the aid of Write Logic Block (WLB) as shown in Figure 5.1. In this configuration mode input D1 to the WLB acts as the 5th address input, while D0 acts as the data input. By configuring both LUTs in the above mode, both F and G modules as well as the WLB can be tested simultaneously.

Since the testing of the LUTs is carried out in RAM configuration mode, the test patterns used for testing the LUTs constitute standard RAM test patterns. However, there are

significant differences in the implementation of LUTs and standard RAMs[20]. It is to be expected that the failure mechanism in RAMs are different from those of LUTs and in fact they form a superset of the failure mechanism of LUTs[21]. The following section describes the test patterns appropriate for testing a LUT configured as a RAM module.

Test Patterns for the RAM Module Since the implementation details of FPGA LUTs are unknown, the behavioral or functional model of the RAM modules is considered. For detecting all possible faults for the behavioral model, the Pattern Sensitive Fault (PSF) model is used. A PSF in its most general form is said to occur when the contents of a cell, or the ability to change its contents, is influenced by the contents of all other cells in the memory. Thus the PSF model with no restrictions can be considered a superset of all other fault models and testing for all PSF faults constitutes an *exhaustive* test which covers all faults in a memory [21]. However, for an n bit memory the number of test patterns required to sensitize all PSFs for each cell are $3 \cdot n \cdot 2^n$. As this results in a large number of test vectors for even a 32bit memory, a functional model is most appropriate. The functional model for the LUT configured as a RAM need not be detailed as we are not interested in localizing faults within the LUT itself [21]. Thus, we consider a reduced functional model, which consists of three blocks: address decoder, memory cell array, read/write logic. Under the reduced functional fault model, the complete functional fault set is reduced to following: Stuck-At Fault (SAF), Transition Fault (TF), Coupling Fault (CF), Neighborhood Pattern Sensitive Fault (NPSF). All combinations of reduced functional faults that need to be tested are pre-

sented in [21] along with the test for those combinations. In general, SAF, TF and CF are addressed by the family of *march* tests such as marching 1/0 pattern. A march test is defined as consisting of a finite sequence of march elements, where march elements are defined as a finite sequence of operations applied to a cell before proceeding to the next cell. For example, the marching 1/0 pattern consists of following march elements: writing a 0 ($w0$) to each cell in ascending address order; reading a cell with an expected value of 0 and writing a 1 ($r0, w1$) in ascending address order; reading a cell with an expected value of 1, writing a 0 ($w0$) and reading a cell with expected value of 0 ($r1, w0, r0$). The constituting march test is represented with the notation: $\{\uparrow(w0); \uparrow(r0, w1); \downarrow(r1, w0, r0)\}$. These tests are widely employed for reduced functional model, and as such, the standard patterns provided by [21] are directly used.

Carry Logic

XC4000 CLBs provide dedicated carry logic for implementing arithmetic functions. The carry logic may be configured for a single or double stage depending on whether the CLB is used to implement one or two full adders. The functions supported by the carry logic are add, subtract, add/subtract, increment, decrement, increment/decrement and 2's complement. The configurations of the carry logic for each of these functions are represented by tuples such as ADD-FG-CI. The first part describes the arithmetic operation, the second part describes the function block used and the third part describes the source of the carry

Operation	Block	Carry Source
ADD	F	CI
SUB	G	F1
ADDSUB		F3
INC		1
DEC		0
INCDEC		

Table 5.1. Carry logic operational modes

input. There is a large number of possible combinations that the carry logic can be configured in. Table 5.1 shows the possible values for each of the three fields. All combinations of these three fields are allowed. Apart from those described above, two additional operations are provided: force carry source to carry output and examine carry input. However, since testing each of these configurations can take too many test configurations, a minset of all possible configurations needs to be derived to cover all functional faults.

Figure 5.2 shows the internal architecture of the carry logic [22] in terms of configuration bits. The carry logic can be partitioned into two blocks, namely, G carry block and F carry block. Each block provides the following functions:

1. Generating carry output depending on the input operands.
2. Bypassing of each stage or introducing a fixed value into the carry chain.

The rest of the circuitry and configuration bits decide for which operations the carry is generated, as described in [22]. To completely test the carry logic, the functionality of each block and of all configuration modes must be exercised. A subset of the configuration modes that

Figure 5.2. XC4000 dedicated carry logic

C.L. Config.	Configuration Bits 1-9								
ADDSUB-FG-CI	1	1	1	x	1	1	1	x	1
DEC-G-0	1	1	0	0	0	0	0	0	0
INC-F-CI	0	1	1	x	1	1	0	1	0
FORCE-1	0	1	0	0	0	0	x	x	1
FORCE-F3	0	1	0	0	0	1	x	x	x

Table 5.2. The subset of carry logic operational modes for full test coverage

exercise the complete functionality of carry logic is given in Table 5.2, along with the coverage achieved in each mode. The functional coverage achieved through these configurations is explained below.

C.L. configuration ADDSUB-FG-CI tests three functions of the C.L. block. In the ADDSUB configuration, bit c7 and input F3 control the addition or subtraction operation, as shown in Figure 5.2. Thus the above configuration tests carry logic for addition, subtraction and the control function which switches between the two operations. The middle field indicates that both F and G C.L. blocks are tested for the above operations. The third field indicates that the configuration control for the carry source path for input CI is also tested.

The second configuration DEC-G-0 tests three additional functions: the G carry block for decrement operation, the bypass function of F carry block and the initiation of a '0' into the carry chain.

Configuration INC-F-CI tests the increment operation for F carry block and the bypass function of G carry block. It is not necessary to test both increment and decrement operations for each block as the operations are the same as adding or subtracting a '1'. This is

achieved by forcing the carry or the borrow to be active while keeping the second operand as zero. Thus, the INC/DEC operations only ensure that the second operand can be forced to a zero and the carry or borrow can be asserted.

The above three configurations test all of the allowed functions of F and G carry blocks. The functions remaining to be tested are the initiation of a '1' into the carry chain and testing of F3 as a carry source. The last two configurations test the remaining two functions.

It is to be noted that carry source F1 is not included in any of the above test configurations. This is because of the nature in which carry is generated. XC4000 C.L. block generates carry by comparing input operands to decide whether to multiplex the operand or carry input to the carry output. Since F1 is the input operand that is to be multiplexed, normal add subtract operations test the F1 path through the carry logic. Thus no extra configuration is required to test the F1 carry source.

5.1.3 XC4000 CLB Test Configurations

The primary objective of CLB test configurations is to allow the primitives to be exercised with appropriate test vectors. The requirements for achieving full functional fault coverage for the primitives LUT, Flip-Flops and the carry logic block have already been discussed. The data path multiplexers are exercised by the test vectors for other primitives and are tested in parallel with those primitives. Thus the CLB configurations are developed such that the maximum number of multiplexers are tested in each configuration. As observed

from Figure 5.1, the minimum number of test configurations is determined by muxes OM1, OM3 and CM1-4 which have 4 data inputs each.

The test configurations are partitioned into two groups. Those configurations that test the function generator blocks, flip-flops and the multiplexers, fall into the first group. The remaining configurations test the carry logic and associated circuitry and fall into the second group. The partitioning of the configurations results from the constraints placed by the routing resources on carry logic configurations. Specifically, the carry logic input CI and output COUT have dedicated routing resources and can not be accessed through local or global interconnects and PIPs like conventional CLB inputs. Thus, to examine the test results for the carry logic it is necessary to utilize the function generator and output pins of the adjoining CLB to access the local or global lines.

It has been found that a total of 5 configurations are necessary to test all logic resources other than carry logic. The coverage achieved through these 5 configurations is as follows. The testing for each of the above configurations is conducted in two phases. During phase one, the function generator block F, G and H are exercised, while during phase two, the flip-flops are tested. This two phase approach is employed to reduce the number of configurations, as the flip-flops and the LUTs share the Data Input (DI). The flip-flops in the XC4000 CLB allow a number of operational modes, including set or reset function, controllable clock enable, and positive or negative clock edge triggering. The first 4 configurations exercise the flip-flops in each of these functional modes to ensure full functional fault

coverage. While developing the above configurations it is ensured that each of the inputs for muxes CM1-4 and OM1,3 are exercised. In addition, the input muxes to F and G LUTs are also tested, except for mux IGM2 which is only partially tested.

The function generator blocks are configured as either 32x1 bit RAM or 2 16x1 bit RAMs for the first 4 configurations. The lower order 4 bits of the address are connected to the 4 inputs of both LUTs. The 5th address bit is connected to input D1 of the WLB. The input D0 of the WLB is the data input for the RAM and is controlled through DIN from CM2. The outputs of the LUTs are connected to the as yet unused inputs of the output mux's OM1-4 and thus to primary outputs X and Y. In the case that LUTs are configured as a 32x1 bit RAM the outputs of the LUT are connected through the H block prior to connecting it to the output muxes. In addition the DIN input is connected to both flip-flops FFX and FFY which are tested during the second phase. The LUTs are tested through the use of the checker-board pattern described in previous section, while the flip-flops are tested through the use of checking sequences described in previous chapters.

5.2 Implementation of Architecture Independent Approach

As with the XC3000 FPGAs, there is a necessity for devising a method by which multiple CLBs can be exercised with the test configurations in parallel. The method described in section 4.3 was proposed as a generic and easily extensible method for testing CLBs in parallel. The following paragraphs explain how the above method is applied to XC4000 devices. In

addition, the limitations in the application of the proposed method are described.

The method described in section 4.3 addresses the problem of exercising a large number of CLBs with the previously generated CLB test configurations. To accomplish this, it generates device configurations by mapping multiple CLBs with a CLB test configuration. The method exploits the regular structure of SRAM FPGAs to automate the task of generating device configurations. The device configuration is subject to constraints such as availability and flexibility of routing resource as well as I/O pins. These constraints are taken into account by the algorithm for generating the device configuration as described in section 4.3.

The inputs to the algorithm are the test configurations for a single CLB such as those described in previous sections. The algorithm essentially does iterative placing and routing of each configuration onto as many CLBs as possible, while meeting the constraints on the routing resources and IO pins. The result of the algorithm execution is a set of netlists in terms of the physical resources of the device. Each netlist is a device configuration exercising a number of CLBs with a given CLB test configuration. It is possible that more than one device configuration may be required to exercise all CLBs for a particular test configuration. The set of netlists thus generated is processed through vendor tools to obtain binary configurations that can be directly down loaded to the DUT.

5.2.1 Implementation Results

The method for automated generation of device configurations described above was implemented, and its application to two devices of Xilinx XC3000 was described in Section 4.3.1. The prototype software *CLB-Test* developed and tested on the XC3000 devices was applied to two devices of XC4000A architecture: XC4002A which has 64 CLBs, and XC4005A which has 196 CLBs. The application of *CLB-Test* to the devices of XC3000 and XC4000 architectures demonstrates the extensibility of the approach. Table 5.3 shows the number of CLBs covered per device configuration for XC4002A and XC4005A devices. For this implementation, a CLB test configuration, with 13 inputs (F1-4, G1-4, C1-4, K) and 3 outputs (YQ, Y, XQ) was chosen for both devices. As shown in Figure 5.1 F1-4 and G1-4 are inputs to function generator F and G, whereas C1-4 and K represent control and clock inputs respectively. Figure 5.3 shows device configuration no. 3 for the XC4002A device using 15 CLBs. The shaded blocks indicate the CLBs configured for testing. The shaded I/O blocks along the periphery of the device are used for test vector inputs or primary outputs for allowing the CLB outputs to be observed. As seen from the table, a minimum of 5 device configurations are required to exercise each CLB with the given CLB test configuration using this method. For the XC4000 CLBs a total of 5 CLB test configurations are required to test all logic resources other than the carry logic. Thus, with this method, 25 device configurations are required to exercise all CLBs with all 5 CLB test configurations. For the XC4005A device, a total of 35 device configurations are required to cover all CLBs. The

Figure 5.3. An example XC4002 device configuration

Device	XC4002A	XC4005A
No. of CLBs	64	196
Total I/Os	64	112
CLB configuration I/Os	13/3	13/3
Max. Configurable CLBs per device configuration	15	33
Min. Configurations possible	5	6
CLBs Placed per Configuration		
Configuration No. 1	14	29
Configuration No. 2	14	31
Configuration No. 3	15	28
Configuration No. 4	14	29
Configuration No. 5	7	29
Configuration No. 6	-	30
Configuration No. 7	-	20

Table 5.3. Results from dynamic generation of device configuration for XC4002 & XC4005 devices

number of I/Os used in the XC4000 configurations are not crucial, as these devices provide boundary scan which can be used to access all I/Os needed for test purposes. As the number of I/Os used in test configurations can be quite large, this method is more suitable for devices that incorporate boundary scan. However, most newer SRAM FPGAs do include boundary scan.

5.3 Limitations on Device Configuration Generation

Although the automated generation of device configurations described above is feasible in most cases, there are limitations to this approach. The limitations are primarily imposed by

the fact that certain resources are highly specific to a device. Thus, generic algorithms for generating device configurations cannot make use of such resources. An example in this case is the dedicated carry logic available in XC4000 CLBs. Some of the reasons that the methodology outlined in the previous section is not directly applicable for configurations employing carry logic are explained below.

The two main I/Os to the dedicated carry logic are CIN and COUT. Both of these signals are not directly accessible to general routing resources in the device. Instead, they have dedicated routing paths that link all CLBs to form a programmable chain. Since the carry logic is generally used to form adders or counters whose units are usually placed as a chain of CLBs, the delay between CLBs affects the critical path timing of the circuits. The dedicated paths provide the least delay and avoid the programmable connections in general routing resources that cause extra delay. Thus, in order to access the inputs and outputs of carry logic, additional CLBs are required. These adjoining CLBs are configured to route the nets between the general routing resources and the carry logic. The above constraint introduces the concept of relative placement of related logic blocks. Since the methodology assumes that all blocks are independent and of same functionality, the placement strategy fails. However, it is possible to incorporate relative placement in the placement routine. Indeed this is targeted as a potential area for improvement in future research.

Since the current placement strategy fails for carry logic configurations, these configurations are prepared manually. Figure 5.4 shows such a device configuration. The require-

Figure 5.4. XC4000 carry logic test configuration

ment of an additional CLB to access inputs and outputs limits the use of test configuration to every other CLB in a given column or chain path as shown in Figure 5.4. The initializing CLB (A/C) are configured with mode FORCE-F1. This mode allows the COUT of that CLB to be driven by input F1 which is accessible through normal routing resources. The COUT of the initializing CLB is connected to the CIN of the CLB under test, thus indirectly exercising the carry logic of CLB under test with the required test vectors. In addition, the initializing CLB must also route the COUT of the preceding CLB to an observable output. This can be achieved by routing the CIN of the initializing CLB through a LUT (F or G) and to output X or Y, where normal routing resources can be used to route signals to primary outputs. The configuration shown in Figure 5.4 forms a chain of CLBs going upwards. With this method each link in the chain can be exercised in parallel with independent vectors, with the outputs available through the next link. Inputs other than the CIN required by the CLB configuration can be assigned to primary inputs of the CLB as these are free to be used. This configuration also tests the programmable connections on the dedicated path in the upward direction.

Since each device configuration tests every alternative CLB, two configurations are required for each column in the device. The second configuration is similar to the first except that the chain starts from the top and proceeds downward, thus testing all CLBs and also the dedicated path downward connections. To test the entire device, such configuration as described above must be generated for all of the 5 CLB test configurations that exercise the

dedicated carry logic.

5.4 Summary

This chapter described the application of CLB test methodology for XC4000 devices. The implementation of the methodology required the development of CLB test configurations. The development of the test configurations using the same fault model as that for XC3000 devices was described. It was seen that the primitives in XC4000 CLBs were the same as those of XC3000 CLBs except for the dedicated carry logic. The test configurations and vectors for this primitive and the RAM mode of the LUTs were described. The limitations of the CLB test methodology in the context of its application to dedicated carry logic were identified. An alternate approach that required developing the test configurations manually was described that overcame those limitations.

Chapter 6

A Methodology for Testing Routing

Resources of SRAM FPGAs

In the case of fixed design, devices can be tested by means of test vectors which exercise the logic and routing resources used. For applications that employ reconfiguration, full testing of the routing resources is required. Thus routing resources other than those used in a particular design must be tested. Furthermore, tests other than those used for logic resources must be employed as the latter do not provide adequate information pertaining to the localization of any routing faults. For this reason, routing resource testing must be conducted independently of logic resource testing.

This chapter introduces an architecture independent methodology for detecting and localizing faults in the routing resources of SRAM FPGAs. The proposed methodology em-

employs a functional level model of the device to attain independence from its architecture. As a result, the methodology is generic and readily applicable to devices with different architectures. These characteristics are further enhanced through dynamic generation of device test configurations, supported by the methodology.

The implementation of the methodology involves the development of a functional level model for the device to be tested. With the current approach, the time taken to develop a device model is very large. As a result, the methodology is implemented for the XC3020 representative SRAM FPGA, which, due to its small size, allows rapid implementation of the device model and faster debugging and verification of the implemented software *Routing-Test*.

The methodology incorporates various algorithms for testing the routing resources. These test algorithms have been implemented and the resulting software (*Routing-Test*) has been verified for the XC3020 device model. The following sections describe a functional level modeling scheme for SRAM FPGAs and the data structures suitable for implementing the model. Further, the test approaches that address routing resources of SRAM FPGAs are discussed along with the algorithms based on the device model for implementing those approaches. Finally, results from the simulation of *Routing-Test* against an injected fault set are presented.

6.1 A Device Model and Data Structures for XC3000

Devices

A device from the XC3000 family can be broadly partitioned into two categories of resources, namely, logic and routing. The former consists of Configurable Logic Blocks (CLB), Tri-state Buffers (TBUF), and Input/Output Blocks (IOB), while the latter consists of local lines, long lines, switch matrices, and Programmable Interconnect Points (PIP) [17].

The device is modeled as a directed graph, where the vertices of the graph model the physical resources on the device, and the edges model the interconnections between these resources [23]. Bidirectional connections are modeled as a pair of directed edges. Thus, local line segments, global lines and pin-wires are modeled as vertices, whereas the connections provided by switch matrices and PIPs are the edges to those vertices. CLBs, TBUFs and IOBs are not considered as vertices themselves; rather the input and output pins of these resources are considered as vertices, with only in-coming or out-going edges. Figure 6.1 (a) shows a representative set of resources, and Fig. 6.1 (b) shows how these resources are modeled.

The directed graph for the above model is implemented by means of linked list data structures. The vertices of the directed graph are represented by members of a list named the vertex-list. Each vertex, in turn, maintains a list of all edges emanating from that vertex called the edge-list. A terminal vertex, that has only in-coming edges, has an empty

Figure 6.1. An example model representing local lines, CLB inputs and their interconnections

edge-list. The members of both of these lists store the test status information updated after each test for use by the test algorithms. Some of the information stored in the list members, and its significance is described as follows: 1. test status indicates whether the resource has been included in a previous test or not, and whether the fault indicated in “fault type” is confirmed or suspected; 2. fault type: sa1, sa0, so or fault free reflects the state of the vertex or edge after each test; 3. weight assigned to each vertex and edge, depends on the above two parameters and dictates the likelihood of the resource inclusion in the next test configuration.

6.2 Test Approach for Routing Resources

The test approach to routing resources is similar to that of logic resources in that it takes advantage of the reconfiguration feature of SRAM FPGAs to map certain test configurations onto the routing resources during testing. These configurations are then subjected to test vectors to detect faults in the underlying routing resources. Similarly, additional test configurations can be mapped onto the routing resources to localize a fault to a particular resource.

However, FPGAs provide a large number of routing resources and interconnections to accommodate a wide range of circuits. As a result, the combinations of possible test configurations to cover all routing resources can be very large. Generating these test configurations to achieve fault detection and localization for each device manually can be extremely inefficient in time and storage requirements. To avoid these overheads the proposed methodology incorporates certain algorithms based on the device model that allow dynamic generation of test configurations for both fault detection and localization in routing resources. For testing purposes the routing resources are split into those that can be tested independently of logic resources and those that cannot. PIPs fall into the latter category as they implement connections between routing and logic resources and are tested in parallel with logic resources. Thus, the test algorithms for these two resource types are addressed separately. The following sections describe the test approach for the routing resources that can be tested independent of logic resources.

6.3 Testing Local Line Resources

In the XC3000 family of devices the local line segments run vertically and horizontally along the height and width of individual CLBs. The local line segments are interconnected by switch matrices, which are present at points of intersection [17]. An effective way to simultaneously test a large number of local lines and switch matrix connections is to form long routes from a source to a destination by connecting many local line segments through switch matrices. By driving such created routes to logic high and low levels and observing the logic levels at the destination pins, the resources used in the routes can be tested for stuck-at-1, stuck-at-0 or stuck-open faults.

The presence of faults in one or more segments in a route would result in a fault at the primary output, however, it would not be possible to identify the faulty segment(s). To localize a fault to a particular segment it is necessary to observe intermediate nodes in the path under test, which will be referred to as the primary path. Intermediate nodes can be observed by branching the path at those nodes and extending a path from the branch point to another primary output. These branches will be referred to as secondary paths. Since secondary paths themselves may include many segments, branches originating in the secondary path may also be necessary to allow faults in its segments to be isolated. This would suggest the need for implementing tree like nets, with as many branches as permitted by the number of available primary outputs, in order to achieve a high degree of observability. The following section describes the algorithms that can be used to develop such test configurations.

6.3.1 Algorithms for Generating Local Line Test Configurations

Test configurations, such as those suggested in previous section, can be generated using an algorithm similar to a routing algorithm, where a routing algorithm is one that selects an optimal path in a network of paths based on some cost function. The algorithm proposed here is based on a cost function specified in terms of the fault status of resources. The input to this router algorithm is a netlist, consisting of a source which is the driver for the net, selected in a pseudo-random fashion. The rest of the available pins are configured as output pins and are used as destinations.

Given the netlist, the router generates a test configuration consisting of paths formed out of local line segments. The router decides which segments to include in the paths for a particular test configuration on the basis of the weights that are assigned to each resource in the device model. The weights are decided on the basis of the fault status of the resources as determined from previous tests, and are updated after every test to reflect the new fault status of each resource. Thus, the algorithm for generating test configurations for local line segments has two main components, a weight assignment rule and a router. Both of these components operate on the device model described in previous sections and are presented below.

Weight Assignment Rule

The weight assignment rule determines the likelihood of a resource being included in a test configuration by assigning a weight to the resource depending on its fault status. A routing resource may have the following fault status in descending order of priority: 1. untested; 2. stuck-at-1/0 or stuck-open, with status not-confirmed; 3. stuck-at-1/0 or stuck-open with status of the resource confirmed to be faulty; 4. fault-free. Resource status in category 3 occurs under very restricted conditions, and such resources are few in number. This type of resource has the lowest priority. The lowest priority resource is assigned the highest weight and vice versa. Furthermore, a fault-free status is always considered to be a confirmed status.

The above weight assignment rule is used in assigning each resource its proper weight prior to the invocation of the router. Thus it assures that the selection of resources by the router for each test configuration was based on the most recent fault status of each resource, resulting in a path that allows high fault coverage.

Router

The objective of the router is to route an optimal path from a source to the destinations in a given netlist, based on a cost function. The problem of finding an optimal path can be redefined as one of finding the shortest distance path from a source to a destination, where the cost function for determining the shortest distance is the sum of weights of the segments

in the path. The weights reflect the priority of inclusion of a resource in a path. Therefore, the cost function optimizes the distance such that the shortest distance path will include the most resources that have a high priority. The shortest distance path problem in a directed graph, with no negative weights and thus no negative cycles, can be solved by Dijkstra's Shortest Path Tree (SPT) algorithm [24] presented below.

Shortest Path Tree Algorithm The SPT algorithm forms the core of the router by generating an SPT rooted at the source. This algorithm generates iteratively a shortest path tree by a process of local minimization, where the distance to each segment in the tree is updated each time a shorter path is found. The SPT algorithm is presented below.

procedure name: sp_tree(source)

The algorithm uses a queue of nodes, which starts out empty.

Each node has following members

dist_s— the length of path in tree from source s to node v.

parent — the parent of node in tree (initially NULL).

inqueue()— true if and only if node is in the queue.

queue operations enqueue() and dequeue() insert or

delete a member from queue.

queue operation head() results in a node

drawn from the head of the queue.

```

begin

s.dists = 0

enqueue(s)

while queue is not empty do

    for each successor w of head() do

        if head().dists + length(head(),w) < w.dists then

            w.dists = head().dists + length(head(),w)

            w.parent = head()

            if !w.inqueue() then

                enqueue(w)

        dequeue()

end

```

Router Algorithm The algorithm for the router is constructed using the SPT routine described above. The input to the algorithm is a list of sources with the source for highest priority net placed first. Each source has a list of destinations. The router routes each net using the SPT routine. The router doesn't do any iterative routing to optimize each net, and

is thus a single stage router. Its primary use is in the local line test configuration generation, where there is only one net to be routed. For this reason, iterative routing is considered unnecessary. The router, however, does allow more than one net to be routed at a time for the PIP and CLB test configuration generation algorithms. In such cases it avoids conflicts for routing resources between different nets by locking each net as it is routed, thus forcing the other net to use different resources, even if the resulting path is non-optimal. It has been found that for the specific cases of PIP and CLB test algorithms, a single stage is sufficient to route all nets required by the algorithms at a given time. The routine for the router is as follows:

procedure name: router(netlist)

$P_{s,d}$ represents a list of resources in path P

from source s to destination d .

begin

while source list is not empty **do**

 sp_tree(source)

while destination list is not empty **do**

trace path from destination to source

store path $P_{s,d}$

lock resources in path $P_{s,d}$

output routed design

end

Local Line Test Configuration Algorithm

The algorithm incorporates the router and weight assignment rule in generating local line test configurations as follows:

procedure name: ll_test

begin

for $i = 1$ **to** *Max. iterations* **do**

choose source pin pseudo-randomly

setup destinations list from available pins

generate netlist from source and destination list

assign_weights()

router(netlist)

output routed test configuration for local lines

end

Algorithm Verification The algorithm was tested on the XC3020 device model, which has 839 local line segments with the maximum number of iterations set to 50. A fault set consisting of 18 faults was injected into the device model. During testing, it was found, that all segments had been exercised by at least one test configuration within 10 iterations. During these iterations the algorithm localized and marked potentially faulty resources. At this point the number of faulty resources was found to be nearly twice the size of the originally injected fault set of 18. Additional iterations did not improve the fault localization and did not significantly reduce the number of spurious faults. These results indicate that the algorithm operates as desired as long as the number of fault-free resources is much smaller than the number of untested or potentially faulty resources. Thus, as the number of fault free resources increases, the weight assignment rule is no longer adequate and must be further refined in order to improve the operation of the router.

A Two Stage Approach

From the above results it was observed that at a certain stage in the algorithm the lengths of the paths are dominated by weights of fault-free segments rather than faulty or potentially faulty segments. The effect of this imbalance is that the addition of a faulty resource to a path consisting of one or more fault-free resources causes the length of the path to increase greatly, thus increasing the probability that a shorter path will be identified and selected and the path with the faulty resource will be discarded. As a result, the faulty resources are not

included in test configurations, and further iterations are futile. Clearly, a new strategy is needed at this point. To overcome this limitation, the two stage approach described below has been adopted.

Stage 1 In this stage the algorithm is as previously described.

Stage 2 This stage of the algorithm starts when all resources have been included in at least 1 configuration.

During stage 1 the algorithm included faulty segments in a path in a non-deterministic manner based on the weights of the resources. In stage 2 the faulty segments are included in paths on the basis of fixed rules. The algorithm first searches the faulty segments in the model, then makes an attempt to route a fault-free path from the primary source to each of the faulty resources. Next, the algorithm proceeds to route a fault-free path from each faulty resource to a primary destination. In this manner the stage 2 algorithm tries to include exactly 1 faulty segment in an otherwise fault-free path. Thus, this stage, accomplishes two goals. Firstly it exercises potentially faulty segments, and secondly, it confirms all faulty segments and thus avoids spurious faults. The stage 2 algorithm needs a different weight assignment rule to assure fault-free paths to and from a faulty segment. The new weight assignment rule in descending order of priority is as follows: fault free; untested; so, unconfirmed; sa1/0, unconfirmed; and so, sa1/0, confirmed. Again low priority resources are assigned high weights and vice versa. The second stage of the algorithm using the same SPT and router routines is as follows:

procedure name: ll_test_stage2

begin

for $i = 1$ to Max. iterations **do**

choose primary source pin

search for all faulty segments

let faulty segments as secondary destinations

netlist = (source, sec. destination list)

assign_weights() / with new weight rule */*

router(netlist)

store routed design

while more un-confirmed faulty segments exist **do**

secondary source = faulty segment

destination list = available IO pins

netlist = (sec. source, destination list)

assign_weights()

router(netlist)

if unused destinations exist **then**

choose destination with shortest path from secondary source.

concatenate path from sec. source to prim. dest to prev. design

output concatenated design

end

Figure 6.2 shows an example test configuration generated by the `ll_test_stage2` routine. The test configuration has a single input (indicated in the figure) while all other I/Os are configured as outputs.

Figure 6.2. An example stage 2 local line test configuration

6.4 Testing PIPs

The programmable connections between the pin-wires and local lines or global lines are known as Programmable Interconnect Points (PIPs). In order to provide sufficient routing flexibility each pin-wire has an average of 5 PIPs connecting it to 5 different routing resources [17]. In the worst case, for each configuration only one of the connections can be exercised for each pin-wire. This implies that to test all programmable interconnect points the number of configurations required is equal to or greater than the maximum number of PIPs per pin-wire. For the XC3000 family, the maximum number of PIPs on a pin-wire is 7 [17], thus requiring at least 7 configurations. Since PIPs connect routing resources to pin-wires, the test vectors for PIPs must propagate through logic resources connected to the pin-wires. Test configurations for PIPs, therefore, include logic resources as well. To propagate the test vectors through the logic resources to primary outputs, the logic resources such as CLBs must be configured properly. To avoid generating extra CLB configurations for PIP testing and to reduce testing time, PIP testing is carried out alongside CLB testing. Test configurations for PIPs are formed out of routing resources required for CLB testing, i.e., test configurations for PIPs are generated while routing the nets connecting primary inputs to inputs of a CLB. The nets used in CLB configurations are routed using a modified router.

The objective of the modified router is to exercise untested PIPs with each new configuration. The modified router has a new weight assignment rule that exhibits a memory

mechanism. PIPs used in previous configurations are given a high weight which is indicative of low priority of inclusion in the next configuration. The weight assigned to a PIP is increased progressively with each inclusion in a configuration, to increase the likelihood of including previously untested PIPs in a test configuration. The test vectors used during CLB testing also exercise the PIPs. Since the test vectors for CLBs can isolate a faulty input, the net containing a faulty PIP is easily isolated by observing the outputs from CLB tests. A typical CLB configuration has 7 used inputs with each CLB requiring approximately 8 configurations for complete testing. This provides sufficient coverage to exercise a large percentage of PIPs surrounding a CLB.

Weight Assignment Rule for PIP Testing

The weight assignment rule assigns weights to a PIP based on its use in previous test configurations as follows:

procedure name: assign_weights_pips()

begin

while *more resources exist* **do**

if *resource is a PIP* **then**

read fault status

lookup weight in table

```
weight = weight + factor × configuration number  
assign weight  
next resource  
end
```

A PIP used in previous configurations is assigned a higher weight than unused PIPs, thus promoting incorporation of unused PIPs in further tests. The weights for used PIPs are incremented with every use in a configuration. This makes used PIPs less likely to be included with every passing configuration, which is necessary to ensure that PIPs which lie on longer routes to a CLB pin would also be included. The following is the allocated weight assignment for PIPs in descending order of priority: untested; so, unconfirmed; sa1/0, unconfirmed; fault free; with so,sa1/0, confirmed having the lowest priority.

The PIP test configuration algorithm is implemented with the modified weight assignment routine and the router and is shown below:

```
procedure name: pip_test()  
  
  begin  
  
    for i = 1 to Max. CLB configurations do  
  
      netlist is generated by CLB Test algorithm
```

```
configure CLBs  
  
assign_weights_pips()  
  
router(netlist)  
  
output routed design  
  
conduct CLB testing  
  
end
```

Figure 6.3 shows the net coverage achieved by representative configurations for PIP testing.

Figure 6.3. Net coverage by 4 PIP configurations

6.5 Fault Tracing Algorithm

The algorithms presented in previous sections require the fault status of each resource in device model to be updated after each test. However, it is not possible to determine the fault status of each resource directly, as the only available data is the list of faults observed at the outputs and the routed design. In fact, the problem of updating fault status is the same as that encountered in fault localization, which is addressed by assigning each resource its proper fault status. The fault tracing algorithm solves both of these problems. The algorithm starts by assigning all resources in a path the fault observed at the primary output of that path. Since the paths are in the form of a tree, some resources are common to two or more paths. It follows that conflicts will arise in determining the status of these resources, if branches of the path have different faults at the outputs. The fault tracing algorithm solves these conflicts using the following rules:

1. Fault-free status has highest priority and over-rides any other status previously assigned to the resource.
2. Fault-free status is the only confirmed status and doesn't change once assigned.
3. Stuck-at fault has the next highest priority and over-rides a stuck-open fault status.
4. Stuck-open status has lowest priority and can be assigned only to resources that have been as yet untested.

5. Stuck-at and stuck-open faults are confirmed if and only if the path from primary input to that resource and from the resource to a primary output has only fault-free resources.

During the initial phase of testing, it follows from rule 4 that a large number of resources will be assigned unconfirmed faults. However, as governed by the weight assignment rule, an unconfirmed faulty resource is more likely to be included in further tests, and, as a result, it is more likely to be included in a fault-free path and to attain its correct status. By the end of stage 1 testing only a few unconfirmed faulty resources remain which are addressed in stage 2. Since in stage 2 exactly one faulty resource is used in a path, a fault at the output confirms the fault of that resource. The inputs to the algorithm are the faults observed at primary outputs and the routed design. Thus, the fault tracing algorithm performs the dual task of updating fault status and localization of faults. The fault tracing algorithm is presented below and takes as input the faults observed at primary outputs and the routed design.

procedure name: `fault_backtrace()`

begin

for each primary output do

let fault = observed fault

while current_resource $\neq$ source do

if output is not fault-free then

```
    if resource.fault_status  $\neq$  fault-free then
        resource.fault_status = fault
    else
        resource.fault_status = fault-free
        current_resource = current_resource.parent
end
```

6.6 Results

A fault simulator specific to the device model has been implemented to verify the local line testing algorithms. The simulator is able to model single and multiple stuck-at and stuck-open type faults. In the case of multiple stuck-at faults, the simulator uses the concept of distance of a faulty resource from the primary output to determine which fault will be dominant at that output. *Routing-Test* routines for local line testing have been verified using the simulator for injected fault sets of up to 18 faults of both stuck-at and stuck-open types. The simulations were done on the XC3020 device model which has 839 local line segments. The results of the simulations are shown in Fig. 6.4.

In the case of no faults, complete local line testing is achieved in under 10 configurations. From Fig. 6.4 it can be seen that for an injected fault set of 18 faults, all faults were detected and localized within 25 configurations. There were no spurious errors generated

Figure 6.4. Local line testing simulation results for a fault set of 18 segments, all detected within 25 iterations

for the given fault set. Though the number of configurations required to test only local line segments may seem to be large, the simulations were done without any prior tests on other resources. It is expected that the number of configurations will be greatly reduced if local line testing is carried out after CLB testing, since the number of local line segments tested during CLB testing is greater than 70%. Local line testing, in that case, will only be needed to test the remaining 30%, thus requiring fewer configurations. To observe the effect of available I/O pins on the performance of the algorithms, simulations were performed with 16 I/O pins prohibited from use. Figure 6.5 shows the results of these simulations. From

Figure 6.5. Simulation results with 16 I/Os prohibited for a fault set of 18 segments, all detected within 30 iterations

these results it can be seen that there is an increase in the number of configurations required to localize all faults by 5. However, from the unvisited segment graph it can be seen that there is no increase in the number of configurations necessary to test all local line segments in the case of no faults. In this case too, there were no spurious faults. From the results it can be conclude that the algorithms perform better with a large number of available I/O pins. This requirement is easily satisfied in most devices with boundary scan, where all available I/O pins can be used for testing.

6.7 Summary

In this chapter a methodology for testing SRAM FPGA's routing resources has been presented. The methodology employs a functional level device model and generic algorithms based on the model for dynamic generation of device test configurations. A device model has been developed for the XC3020 device and has been used to verify the test algorithms for various resources implemented in the prototype software *Routing-Test*. Results from the tests prove the methodology to be successful in achieving accurate fault localization while maintaining a high degree of flexibility. Test results show that *Routing-Test* can test all local line segments within 10 iterations for a no faults case with little effect of number of available I/Os on performance. For a fault set of 18 faults including stuck-at and stuck-open type faults, *Routing-Test* successfully detected and localized all faults within 25 iterations, with an increase of only 5 iterations for reduced number of I/Os.

Chapter 7

Integration of Logic and Routing

Resource Testing

The previous chapters have introduced test methodologies for fault detection and localization of logic and routing resources in SRAM FPGAs. However, to enhance their fault detection and localization capabilities, it is necessary to integrate the two methodologies. This chapter describes the integration of logic and routing resource test methodologies through the use of a common device model. The modifications to existing test methodologies necessary for implementing an integrated test methodology are presented. An experimental test environment is described that allows the application of the test methodology to faulty devices. In addition, an approach for incorporating the test methodology into a reconfigurable system using the concept of a universal device model is proposed.

7.1 The Need for Integration

The logic resource test methodology introduced in chapter 4 allows both the detection of faults in CLBs and the localization of a faulty CLB. However, a fault that manifests itself at a primary output of the FPGA under test may, in fact, be caused by a faulty routing resource present in the nets connecting primary device I/Os to CLB I/Os. As a result, fault free CLBs may be diagnosed with spurious faults. Such spurious faults can be avoided if fault information from routing resource testing is incorporated during CLB testing. During testing of a particular resource, other resources included in a test configuration may also be tested. If the fault status of these resources is retained and used during the development of test configurations for other resources, the incidence of spurious faults can be reduced. In addition, the fault information from different test phases available through the integration of test methodologies can be cross-examined to improve fault localization. For example, a set of resources with suspected faults may be narrowed if certain resources within the set are found to be fault-free during test phases for resource types other than those in the given set.

From the above discussion, it can be noted that the principal factor towards achieving integration is the availability of fault information obtained from each test phase during subsequent generation of test configurations. Apart from the fault information, it is equally important to have information regarding the use of a particular resource in previous test configurations. The device model employed by the routing resource test methodology stores

the above information for the use of test algorithms. Thus, the device model presents an efficient means of integrating the two test methodologies. The following section describes the implementation of an integrated test methodology based on a common device model.

7.2 An Integrated Test Methodology

The use of a common device model is fundamental to the realization of an integrated test methodology for SRAM FPGAs, as it allows both test methodologies to share the fault information for all resource types. A functional level device model, the main component in the routing resource test methodology, upon which all test algorithms are based was introduced in chapter 6. The device model and data structures described in section 6 incorporate all elements essential for implementing the integrated test methodology, including a provision for storing the fault status of each resource in the device and the use or lack of each resource in previous test configurations. This model can, therefore, be used for the integrated test methodology without major modifications. However, the algorithms employed by the logic resource test methodology introduced in chapter 4 are not based on the device model, and therefore cannot access the fault information obtained during routing resource testing. The problem is, thus, one of incorporating the device model into the logic resource test methodology.

The logic resource test methodology employs architecture specific CLB test configurations in an architecture independent approach to exercise multiple CLBs with those configu-

rations. As discussed in section 4.3, the algorithms employed by the methodology generate a netlist for each test configuration, which is routed with the help of the vendor's CAD tools. The use of the router provided by these tools imposes certain limitations, implying that the fault information obtained through prior test phases cannot be used during the generation of subsequent test configurations. Moreover, it implies that the status of the resources used in each test configuration cannot be retained. Thus, its use violates the main criterion for integration.

The above problems can be eliminated through the use of a router based on the device model used by the routing resource test methodology. Such a router has been implemented for the routing resource test methodology and is described in section 6.3.1. It should be noted that the router presented there is of limited capability and cannot route as many nets as the one included in the vendor's CAD tools. In addition, it is currently applicable only to the XC3020 device, as the model necessary for its operation has only been implemented for the XC3020 device. However, the router does provide all the necessary elements for proper operation of the logic resource test methodology. Furthermore, it allows all test algorithms to be based on a common device model.

The use of a common device model allowed the implementation of a prototype test software *FPGA-Tester* incorporating both logic and routing resource test methodologies. The *FPGA-Tester* software has been interfaced with the vendor CAD tools and a generic IC tester for application of the test methodology to faulty devices. The resulting test environment is

described below.

7.3 An Overview of the Test Environment

The algorithms employed by *FPGA-Tester* rely on dynamically updated fault information in the device model for proper operation. The dynamic updating of fault information is possible only if there exists a means by which test configurations can be automatically downloaded to the DUT, exercised by appropriate test vectors, and the test results uploaded in real time. An experimental test environment has been set up that allows the above operation by interfacing *FPGA-Tester* to the vendor's CAD tools and a generic IC tester. An overview of the test environment is presented below.

The *FPGA-Tester* software running on an HP-Apollo 715 workstation generates a device test configuration during each test phase. Before the test configuration can be downloaded to the DUT, it must be converted to the appropriate configuration bit-stream format. However, the configuration bit-stream format is proprietary, necessitating the use of the vendor's CAD tools to convert the test configuration to the appropriate configuration bit-stream. Most FPGA CAD tools provide a chip editor with a scripting interface that allows the user to edit the design through the use of script commands. *FPGA-Tester* takes advantage of this interface and outputs the test configuration in the form of a script. When executed by the chip editor, this script generates the desired test configuration. Although this design is in the format internal to the vendor's CAD tools, it is identical to the test configuration generated

by *FPGA-Tester*.

The NeoCAD FPGA Foundry tools running on the HP-Apollo 715 workstation are used for this test environment. The chip editor (EPIC) provided by these tools is invoked by *FPGA-Tester* with the device configuration script as input. The script is processed by EPIC, resulting in a routed design in NeoCAD's internal format. Once the device configuration is converted into internal format of the CAD tools, it is processed by downstream tools to convert it into the appropriate configuration bit-stream.

The testing of the device is carried out with the help of an IC Tester (IMS XL100) controlled through a host machine (Mentor Graphics workstation, Domain Series 3000). The host machine is linked to the HP-Apollo workstation through a LAN connection. The IC Tester executes a batch process under the initiation of the current test routine in progress. In the batch process the IC tester reads the configuration bit-stream and down loads it into the device. It then reads the test vectors corresponding to the test configuration and applies them to the device. The outputs are sampled and compared with the expected outputs. The fault information thus generated is returned to the test routine for the fault tracing algorithm to process. Figure 7.1 illustrates the steps involved in an entire test cycle. The sequence of events in Figure 7.1 are listed below:

1. Test algorithms for various resources operate on the device model and generate a netlist, which is converted into a routed design using the router.
2. The routed design is converted into a script readable by NeoCAD's EPIC tool. EPIC

Figure 7.1. Test execution sequence

is used to convert the routed design into a NeoCAD proprietary format. The NeoCAD configurations bit-stream generator “Bitsh” is used to convert the design to the appropriate bit-stream format for down loading to the DUT.

3. The IMS tester under the control of the test routine down loads the configuration bit-stream into the DUT. The tester then applies test vectors for that particular configuration to the DUT and compares the outputs with the expected outputs. The fault information generated is given to the fault tracing algorithm.
4. The fault tracing algorithm receives a list of errors observed at the primary outputs and traces these errors to their originating resources.

The completion of the experimental setup was hindered by the lack of an adequate in-

terface for the IC tester. Specifically, the initiation of the batch process, under the control of *FPGA-Tester* was impeded, due to the lack of a network interface to the controller. It is expected that with the installation of the new version of the controller software, the above mentioned problems would be solved.

7.4 Integration of the Test Methodology Into a Reconfigurable System

This section introduces a novel concept of integrating the test methodology developed in this research into a reconfigurable system employing SRAM FPGAs through the use of a universal device model. Before the concept can be introduced, it is important to understand the principal components in a reconfigurable system and their function. Consider a reconfigurable system composed of the following parts: a circuit employing SRAM FPGAs with redundant resources, an error detection module, a fault detection and location module, a re-mapping module, and a control module. Such a system finds application where the hardware/software overhead of the additional modules is justified by the greater cost of replacing the faulty hardware. Satellite communications systems employing FPGA based Multi-Chip Modules (MCMs) represents an example of such systems where reparability by conventional techniques is impractical.

The operation of the system described above can be outlined as follows: the error de-

tection module continually monitors the output of the main circuit and indicates any errors that it detects to the control module. The control module then signals the fault detection and location module, which subjects the FPGA devices to the test methodology. The test methodology generates information pertaining to the presence and location of faults in the devices. This information is used by the re-mapping module to reconfigure the faulty device in the system using redundant resources, such that the faulty resources are avoided.

The re-mapping module generally consists of place and route CAD tools with the additional functionality of considering faulty resources while generating the new design to be mapped onto the FPGA. Although current CAD tools do not support the above functionality, recently, new CAD tools have been proposed that support the development of systems based on dynamic reconfiguration [25] [26]. Since these tools support reconfiguration they allow isolation of certain resources during re-mapping a design. The faulty resources can thus be avoided during reconfiguration by using this feature. However, the proper operation of these CAD tools is highly dependent on the fault status information of each physical resource.

The new approach allows the CAD tools to access the fault information through the incorporation of the device model into the CAD tools. The inclusion of the device model into the CAD tools will allow seamless integration of *FPGA-Tester* and the reconfiguration module consisting of the CAD tools. Present place and route CAD tools are based on a device model similar to the one used by *FPGA-Tester*. However, the model employed by the CAD

tools supports a different information set that doesn't include the fault information. The proposed universal device model supports a superset of the information required for proper operation of *FPGA-Tester* and the CAD tools. With all tools operating on the same device model, each software module has available the required fault information for all resources at each test stage. Thus, the universal device model provides a means of integrating the operation of the test methodology and the reconfiguration software in a manner that allows an efficient implementation of a reconfigurable system.

7.5 Summary

In this chapter a novel approach towards integration of test methodologies and CAD tools has been presented. The approach proposed a universal device model for logic and routing resource test methodologies that is equally applicable for implementing CAD tools such as routers. Based on the universal device model an integrated test methodology has been presented that incorporates logic and routing resource testing. An overview of a reconfigurable system employing the integrated test methodology has been presented. Further, the advantages of incorporating the universal device model in the reconfiguration module's CAD tools to avoid faulty resources has been described for such a system.

Chapter 8

Summary and Future Research

The objective of this research was to develop a test methodology for SRAM based FPGAs targeted at the fault detection and localization requirements of reconfigurable systems. With the above objective, the requirements of a test methodology for SRAM FPGAs employed in a reconfigurable system were identified. With the fault detection and localization in logic and routing resources as chief requirements, two test methodologies addressing logic and routing resources were successfully implemented and verified. Both methodologies employ architecture independent algorithms for dynamic generation of device test configurations to enhance their applicability to different FPGA architectures.

The algorithms for both methodologies have been implemented and the resulting prototype software have been tested. Results from the tests prove the test methodologies to be successful in accurately detecting and localizing faults. In addition, verification of the

prototype software for logic resource test methodologies against the smallest and largest devices in the two Xilinx FPGA architectures proves the methodology to be scalable within the same architecture and extensible to different architectures.

For implementing the logic resource test methodology, CLB test configurations for the XC3000 and XC4000 architectures were developed. It was found that 7 configurations were required for the XC3000 CLB, whereas the XC4000 CLB required a total of 10 configurations. The algorithms for developing the device configurations were implemented and tested for the XC3020A, XC3195A, XC4002A and the XC4005A devices. It was found that a total of 28, 49, 35 and 45 device configurations were required to exercise all CLBs with the given set of CLB configurations in these devices respectively. The XC4000 device configurations included 10 configurations to test the 5 carry logic configurations.

Although the routing resource test methodology has been verified for only the XC3020 device, the methodology is inherently extensible to different architectures due to the use of a functional level device model. All test algorithms are based on the functional level device model and are therefore independent of the device architecture. For the verification of routing resource test methodology the algorithms for testing routing resources were implemented and a model was developed for the XC3020 device. A fault simulator developed for testing the algorithms against an injected fault set was used to determine the fault detection and localization capability of the algorithms. It was found that for the XC3020 device a set of 18 faults were detected and localized within 25 iterations, thus requiring 25 device

configurations. However, in the case of no faults, only 10 iterations were required to completely test all local lines. These results indicate the routing resource test methodology to be successful in 100% detection and localization of single and multiple faults in local lines.

The integration of architecture independent logic and routing resource test methodologies was proposed to improve their fault detection and localization capabilities. It was shown that the existing device model used for the routing resource test methodology fulfilled the requirement of supporting the integrated test methodology. The modifications necessary for the logic resource test methodology to allow integration were discussed and implemented, resulting in a prototype software *FPGA-Tester* that incorporates both logic and routing resource test methodologies.

8.1 Areas for Enhancement and Future Research

The generation of a device model is essential for the application of the proposed test methodology. However, the current method for preparing the device model manually has been a limiting factor in the application of the methodology. A method that overcomes current limitations has been proposed. It allows automated generation of a device model using the schematic image of the device provided by the vendor's tools. It is possible to automatically extract the information necessary for generating the device model from this image. This method helps reduce the time and complexity involved in the development of the device model. In addition, the incidence of connectivity errors are reduced, thus reducing the

debugging time. Another area for enhancement is the completion of the experimental setup for interfacing *FPGA-Tester* with the IC tester and the vendor's CAD tools.

The algorithms developed for testing routing resources do not include testing of bridging faults. This has not been possible as testing for bridging faults requires exercising adjacent resources with test vectors. However, the functional model used by the test algorithms doesn't include "adjacency" information to avoid architectural details. Thus, further research is needed in this area to develop configurations for testing bridging faults. The routing resource test methodology requires additional research in the area of interconnect testing to improve the fault coverage of switch matrices. A method proposed in [8] is suitable for interconnect testing, however, further work is required to incorporate the method into the current test methodology.

Currently, the methodology developed does not address testing of I/O blocks completely. In addition, logic resources other than the CLBs are not tested under this methodology, as those resources are generally specific to a particular architecture. A method for integrating the testing of such resources into the generic test methodology is required, and is a possible area of future research.

In addition to the above enhancements, further research into diagnostic information necessary for reconfigurability of a particular design is necessary. The information required to achieve reconfiguration without significant loss in performance determines the extent of fault localization and thus affects the implementation of test algorithms.

References

- [1] J. V. Oldfield and R. C. Dorf, "Field Programmable Gate Arrays," Wiley-Interscience, 1995.
- [2] R. O. Duarte and M. Nicolaidis, "A Test Methodology Applied to Cellular Logic Programmable Gate Arrays," 4th International Workshop on Field-Programmable Logic and Applications Proceedings, 1994.
- [3] S. O. Frake, S. G. Lawson, and J. E. Mahoney, Xilinx, "A Scan-testable Mask Programmable Gate Array for Conversion of FPGA Designs," IEEE Custom Integrated Circuits Conference, pp.27.3.1-27.3.4, 1992.
- [4] C. Jordan and W. R. Marnane, "Incoming Inspection of FPGAs," European Test Conference, pp.371-376, 1993.
- [5] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Built-In Self-Test of Logic Blocks in FPGAs," VLSI Test Symposium, 1996.
- [6] C. Stroud, S. Konala, P. Chen, and M. Abramovici, "Evaluation of FPGA Resources for Built-In Self-Test of Programmable Logic Blocks," VLSI Test Symposium, 1996.
- [7] C. Stroud and A. D. Russ, "Non-Intrusive Built-In Self-Test for FPGA and MCM Applications," Proc. IEEE International ASIC Conf., pp.480-485, 1995.
- [8] F. Lombardi, D. Ashen, X. Chen, and W. K. Huang, "Diagnosing Programmable Interconnect Systems for FPGAs," ACM/SIGDA Fourth International Symposium on Field-Programmable Gate Arrays, 1996.
- [9] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays," Proceedings of the IEEE, Vol. 81, No. 7, pp.1013-1027, July 1993.
- [10] R. Cuddapah and M. Corba, "Reconfigurable Logic for Fault Tolerance," Proc. 5th International Workshop on Field Programmable Logic and Applications, pp.380-386, 1995.

- [11] N. Toon, "Reconfigurable Hardware from Programmable Logic Devices," Proc. 4th International Workshop on Field Programmable Logic and Applications, pp.281-287, 1994.
- [12] F. Hanchek and S. Dutt, "Node-Covering Based Defect and Fault Tolerance Methods for Increased Yield in FPGAs," 9th International Conference on VLSI Design, pp.225-229, January 1996.
- [13] H. Huong, V. Szwarc, T. A. Kwasniewski, "FPGA Reconfigurability in the Presence of Logic and I/O Faults," CCECE 1997.
- [14] K. Roy and S. Nag, "On Channel Architecture and Routability of FPGA's Under Faulty Conditions," Proc. 4th International Workshop on Field Programmable Logic and Applications, pp.361-366, 1994.
- [15] K. Kwiat, W. Debany, and S. Hariri, "Effects of Technology Mapping on Fault Coverage in Reprogrammable FPGAs," IEEE Proc.-Comput. Digit. Tech., Vol. 142, No. 6, pp.407-410, November 1995.
- [16] M. Herman and W. Hoffman, "Fault Modeling and Test Generation for FPGAs," 4th International Workshop on Field Programmable Logic and Applications Proceedings, pp.1-10, Sept 1994.
- [17] Xilinx, "Programmable Logic Data Book," 1994, San Jose, California.
- [18] J.P. Hayes and T. Sridhar, "A Functional Approach to Testing Bit-Sliced Microprocessors," IEEE Trans. on Computers, Vol. C-30, No.8, pp.563-571, August 1981.
- [19] A.D. Friedman and P.R. Menon, "Fault Detection in Digital Circuits," Englewood Cliffs, NJ: Prentice Hall, 1971.
- [20] M. Robert, L. Torres, F. Moraes and D. Auvergne, "Influence of Logic Block Layout Architecture on FPGA Performance," 4th International Workshop on Field Programmable Logic and Applications Proceedings, pp.1-10, Sept 1994.
- [21] A.J. van de Goor, "Testing Semiconductor Memories Theory and Practice," John Wiley and Sons, 1991.
- [22] B. New, "Using Dedicated Carry Logic in XC 4000 Family of FPGAs," Xilinx Application Note 013.001.
- [23] C. Chen, et. al., "TRACER-FPGA: A Router For RAM Based FPGAs," IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, Vol.14, No. 3, pp.371-374, 1995.

- [24] F. Buckley and F. Harary, "Distance in Graphs," Addison-Wesley, 1990.
- [25] P. Lysaght, H. Dick, G. McGregor, D. McConnell and J. Stockwood, "Prototyping Environment for Dynamically Reconfigurable Logic," pp.409, Proc. 5th International Workshop on Field Programmable Logic and Applications, FPL 1995.
- [26] P. Lysaght and J. Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays," IEEE Trans. on VLSI Systems, Vol. 4, No. 3, pp.381-390, September 1996.

Appendix A

XC3000 CLB Test Vectors

Configuration Primary Inputs Expected O/Ps

C1: LUT F implements function 'F' (shown on Page 43.)
LUT G implements function 'F*' (shown on Page 43.)

C1	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
I	0	0	x	x	0	x	0	x	0	1	0	0
	0	0	x	x	0	x	0	x	1	0	1	0
	0	0	x	x	0	x	1	x	1	0	0	1
	0	0	x	x	1	x	1	x	1	0	1	0
	0	0	x	x	1	x	0	x	1	0	0	1
	0	1	x	x	1	x	0	x	1	0	1	0
	0	1	x	x	0	x	0	x	1	0	0	1
	0	1	x	x	0	x	1	x	1	0	1	0
	0	1	x	x	1	x	1	x	1	0	0	1
	1	1	x	x	1	x	1	x	1	0	1	0
	1	1	x	x	1	x	0	x	1	0	0	1
	1	1	x	x	0	x	0	x	1	0	1	0
	1	1	x	x	0	x	1	x	1	0	0	1
	1	0	x	x	0	x	1	x	1	0	1	0
	1	0	x	x	1	x	1	x	1	0	0	1
	1	0	x	x	1	x	0	x	1	0	1	0
	1	0	x	x	0	x	0	x	1	0	0	1

Table A.1

C2: LUT F implements function 'F*'

LUT G implements function 'F'

C2	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
<hr/>												
I	0	0	x	x	0	x	0	x	0	1	0	0
	0	0	x	x	0	x	0	x	1	0	0	1
	0	0	x	x	0	x	1	x	1	0	1	0
	0	0	x	x	1	x	1	x	1	0	0	1
	0	0	x	x	1	x	0	x	1	0	1	0
	0	1	x	x	1	x	0	x	1	0	0	1
	0	1	x	x	0	x	0	x	1	0	1	0
	0	1	x	x	0	x	1	x	1	0	0	1
	0	1	x	x	1	x	1	x	1	0	1	0
	1	1	x	x	1	x	1	x	1	0	0	1
	1	1	x	x	1	x	0	x	1	0	1	0
	1	1	x	x	0	x	0	x	1	0	0	1
	1	1	x	x	0	x	1	x	1	0	1	0
	1	0	x	x	0	x	1	x	1	0	0	1
	1	0	x	x	1	x	1	x	1	0	1	0
	1	0	x	x	1	x	0	x	1	0	0	1
	1	0	x	x	0	x	0	x	1	0	1	0

Table A.2

C3: LUT F implements function 'F'

LUT G implements function 'F*'

C3	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
<hr/>												
	0	x	0	x	0	0	0	0	x	0	1	1
	0	x	0	x	0	1	0	0	x	0	0	0
	0	x	0	x	1	1	0	0	x	0	1	1
	0	x	0	x	1	0	0	0	x	0	0	0
	0	x	1	x	1	0	0	1	x	0	1	1
	0	x	1	x	0	0	0	1	x	0	0	0
	0	x	1	x	0	1	0	1	x	0	1	1
	0	x	1	x	1	1	0	1	x	0	0	0
	1	x	1	x	1	1	0	1	x	0	1	1

1	x	1	x	1	0	0	1	x	0	0	0
1	x	1	x	0	0	0	1	x	0	1	1
1	x	1	x	0	1	0	1	x	0	0	0
1	x	0	x	0	1	0	0	x	0	1	1
1	x	0	x	1	1	0	0	x	0	0	0
1	x	0	x	1	0	0	0	x	0	1	1
1	x	0	x	0	0	0	0	x	0	0	0
0	x	0	x	0	0	1	0	x	0	0	0
0	x	0	x	0	1	1	0	x	0	1	1
0	x	0	x	1	1	1	0	x	0	0	0
0	x	0	x	1	0	1	0	x	0	1	1
0	x	1	x	1	0	1	1	x	0	0	0
0	x	1	x	0	0	1	1	x	0	1	1
0	x	1	x	0	1	1	1	x	0	0	0
0	x	1	x	1	1	1	1	x	0	1	1
1	x	1	x	1	1	1	1	x	0	0	0
1	x	1	x	1	0	1	1	x	0	1	1
1	x	1	x	0	0	1	1	x	0	0	0
1	x	1	x	0	1	1	1	x	0	1	1
1	x	0	x	0	1	1	0	x	0	0	0
1	x	0	x	1	1	1	0	x	0	1	1
1	x	0	x	1	0	1	0	x	0	0	0
1	x	0	x	0	0	1	0	x	0	1	1

Table A.3

C4: LUT F implements function 'F'
 LUT G implements function 'F*'

C4	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
0	x	x	0	0	0	0	0	0	1	0	x	1
0	x	x	0	0	1	0	0	0	1	0	x	0
0	x	x	0	1	1	0	0	0	1	0	x	1
0	x	x	0	1	0	0	0	0	1	0	x	0
0	x	x	1	1	0	0	1	1	0	0	x	1
0	x	x	1	0	0	0	1	1	0	0	x	0
0	x	x	1	0	1	0	1	1	0	0	x	1
0	x	x	1	1	1	0	1	1	0	0	x	0
1	x	x	1	1	1	0	1	1	1	0	x	1

1	x	x	1	1	0	0	1	1	0	x	0
1	x	x	1	0	0	0	1	1	0	x	1
1	x	x	1	0	1	0	1	1	0	x	0
1	x	x	0	0	1	0	0	1	0	x	1
1	x	x	0	1	1	0	0	1	0	x	0
1	x	x	0	1	0	0	0	1	0	x	1
1	x	x	0	0	0	0	0	1	0	x	0
0	x	x	0	0	0	1	0	1	0	x	0
0	x	x	0	0	1	1	0	1	0	x	1
0	x	x	0	1	1	1	0	1	0	x	0
0	x	x	0	1	0	1	0	1	0	x	1
0	x	x	1	1	0	1	1	1	0	x	0
0	x	x	1	0	0	1	1	1	0	x	1
0	x	x	1	0	1	1	1	1	0	x	0
0	x	x	1	1	1	1	1	1	0	x	1
1	x	x	1	1	1	1	1	1	0	x	0
1	x	x	1	1	0	1	1	1	0	x	1
1	x	x	1	0	0	1	1	1	0	x	0
1	x	x	1	0	1	1	1	1	0	x	1
1	x	x	0	0	1	1	0	1	0	x	0
1	x	x	0	1	1	1	0	1	0	x	1
1	x	x	0	1	0	1	0	1	0	x	0
1	x	x	0	0	0	1	0	1	0	x	1

test vectors for FF X

A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
x	x	x	x	x	x	x	x	0	1	0	x
x	x	x	x	x	x	x	0	0	0	0	x
x	x	x	x	x	x	x	0	0	1	0	x
x	x	x	x	x	x	x	1	0	1	0	x
x	x	x	x	x	x	x	1	0	0	1	x
x	x	x	x	x	x	x	0	0	0	0	x
x	x	x	x	x	x	x	0	0	1	0	x
x	x	x	x	x	x	x	1	0	1	0	x
x	x	x	x	x	x	x	1	0	0	1	x

Table A.4

C5	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
	0	0	0	x	x	0	0	0	1	0	1	x
	0	0	0	x	x	1	0	0	1	0	0	x
	0	0	1	x	x	1	0	1	1	0	1	x
	0	0	1	x	x	0	0	1	1	0	0	x
	0	1	1	x	x	0	0	1	1	0	1	x
	0	1	0	x	x	0	0	0	1	0	0	x
	0	1	0	x	x	1	0	0	1	0	1	x
	0	1	1	x	x	1	0	1	1	0	0	x
	1	1	1	x	x	1	0	1	1	0	1	x
	1	1	1	x	x	0	0	1	1	0	0	x
	1	1	0	x	x	0	0	0	1	0	1	x
	1	1	0	x	x	1	0	0	1	0	0	x
	1	0	0	x	x	1	0	0	1	0	1	x
	1	0	1	x	x	1	0	1	1	0	0	x
	1	0	1	x	x	0	0	1	1	0	1	x
	1	0	0	x	x	0	0	0	1	0	0	x
	0	0	0	x	x	0	1	0	1	0	0	x
	0	0	0	x	x	1	1	0	1	0	1	x
	0	0	1	x	x	1	1	1	1	0	0	x
	0	0	1	x	x	0	1	1	1	0	1	x
	0	1	1	x	x	0	1	1	1	0	0	x
	0	1	0	x	x	0	1	0	1	0	1	x
	0	1	0	x	x	1	1	0	1	0	0	x
	0	1	1	x	x	1	1	1	1	0	1	x
	1	1	1	x	x	1	1	1	1	0	0	x
	1	1	1	x	x	0	1	1	1	0	1	x
	1	1	0	x	x	0	1	0	1	0	0	x
	1	1	0	x	x	1	1	0	1	0	1	x
	1	0	0	x	x	1	1	0	1	0	0	x
	1	0	1	x	x	1	1	1	1	0	1	x
	1	0	1	x	x	0	1	1	1	0	0	x
	1	0	0	x	x	0	1	0	1	0	1	x

A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
x	x	x	x	x	x	x	x	0	1	x	0
x	x	x	x	x	x	x	0	0	0	x	0
x	x	x	x	x	x	x	0	0	1	x	0
x	x	x	x	x	x	x	1	0	1	x	0
x	x	x	x	x	x	x	1	0	0	x	1
x	x	x	x	x	x	x	0	0	0	x	0
x	x	x	x	x	x	x	0	0	1	x	0
x	x	x	x	x	x	x	1	0	1	x	0
x	x	x	x	x	x	x	1	0	0	x	1

Table A.5

C6: LUT F implements function 'F'
 LUT G implements function 'F*'

C6	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
0	0	x	0	x	0	0	0	0	1	0	1	1
0	0	x	0	x	1	0	0	0	1	0	0	0
0	0	x	1	x	1	0	1	1	0	0	1	1
0	0	x	1	x	0	0	1	1	0	0	0	0
0	1	x	0	x	0	0	0	1	0	0	1	1
0	1	x	0	x	1	0	0	1	0	0	0	0
0	1	x	0	x	1	0	0	1	0	0	1	1
0	1	x	1	x	1	0	1	1	0	0	0	0
1	1	x	1	x	1	0	1	1	0	0	1	1
1	1	x	1	x	0	0	1	1	0	0	0	0
1	1	x	0	x	0	0	0	1	0	0	1	1
1	1	x	0	x	1	0	0	1	0	0	0	0
1	0	x	0	x	1	0	0	1	0	0	1	1
1	0	x	1	x	1	0	1	1	0	0	0	0
1	0	x	1	x	0	0	1	1	0	0	1	1
1	0	x	0	x	0	0	0	1	0	0	0	0
0	0	x	0	x	0	1	0	1	0	0	0	0
0	0	x	0	x	1	1	0	1	0	0	1	1
0	0	x	1	x	1	1	1	1	0	0	0	0
0	0	x	1	x	0	1	1	1	0	0	1	1
0	1	x	1	x	0	1	1	1	0	0	0	0

0	1	x	0	x	0	1	0	1	0	1	1
0	1	x	0	x	1	1	0	1	0	0	0
0	1	x	1	x	1	1	1	1	0	1	1
1	1	x	1	x	1	1	1	1	0	0	0
1	1	x	1	x	0	1	1	1	0	1	1
1	1	x	0	x	0	1	0	1	0	0	0
1	1	x	0	x	1	1	0	1	0	1	1
1	0	x	0	x	1	1	0	1	0	0	0
1	0	x	1	x	1	1	1	1	0	1	1
1	0	x	1	x	0	1	1	1	0	0	0
1	0	x	0	x	0	1	0	1	0	1	1

Table A.6

C7: There is only one function of 5 variables allowed. This function is implemented by the CAD tool, individual control of LUTs is not allowed. Thus a special function of 5 vars A,B,C,D,E similar to 'F' is used.

C7	A	B	Qx	Qy	C	D	E	DI	EC	RD	X	Y
	0	0	x	x	0	0	0	x	x	x	1	1
	0	0	x	x	0	0	1	x	x	x	0	0
	0	0	x	x	0	1	1	x	x	x	1	1
	0	0	x	x	0	1	0	x	x	x	0	0
	0	0	x	x	1	1	0	x	x	x	1	1
	0	0	x	x	1	1	1	x	x	x	0	0
	0	0	x	x	1	0	1	x	x	x	1	1
	0	0	x	x	1	0	0	x	x	x	0	0
	0	1	x	x	1	0	0	x	x	x	1	1
	0	1	x	x	1	0	1	x	x	x	0	0
	0	1	x	x	1	1	1	x	x	x	1	1
	0	1	x	x	1	1	0	x	x	x	0	0
	0	1	x	x	0	1	0	x	x	x	1	1
	0	1	x	x	0	1	1	x	x	x	0	0
	0	1	x	x	0	0	1	x	x	x	1	1
	0	1	x	x	0	0	0	x	x	x	0	0
	1	1	x	x	0	0	0	x	x	x	1	1
	1	1	x	x	0	0	1	x	x	x	0	0
	1	1	x	x	0	1	1	x	x	x	1	1

1	1	x	x	0	1	0	x	x	x	0	0
1	0	x	x	0	1	0	x	x	x	1	1
1	0	x	x	0	1	1	x	x	x	0	0
1	0	x	x	0	0	1	x	x	x	1	1
1	0	x	x	0	0	0	x	x	x	0	0
1	0	x	x	1	0	0	x	x	x	1	1
1	0	x	x	1	0	1	x	x	x	0	0
1	0	x	x	1	1	1	x	x	x	1	1
1	0	x	x	1	1	0	x	x	x	0	0
1	1	x	x	1	1	0	x	x	x	1	1
1	1	x	x	1	1	1	x	x	x	0	0
1	1	x	x	1	0	1	x	x	x	1	1
1	1	x	x	1	0	0	x	x	x	0	0

Table A.7