In-System Testing of Configurable Logic Blocks in Xilinx 7-Series FPGAs

Harmish Rajeshkumar Modi

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Carl B. Dietrich
Michael S. Hsiao

June 22, 2015
Blacksburg, Virginia

Keywords: FPGA, Built-In Self-Test, Iterative Logic Array

In-System Testing of Configurable Logic Blocks in Xilinx 7-Series FPGAs

Harmish Rajeshkumar Modi

ABSTRACT

FPGA fault recovery techniques, such as bitstream scrubbing, are only limited to detecting and correcting soft errors that corrupt the configuration memory. Scrubbing and related techniques cannot detect permanent faults within the FPGA fabric, such as short circuits and open circuits in FPGA transistors that arise from electromigration effects. Several Built-In Self-Test (BIST) techniques have been proposed in the past to detect and isolate such faults. These techniques suffer from routing congestion problems in modern FPGAs that have a large number of logic blocks. This thesis presents an improved BIST architecture for all Xilinx 7-Series FPGAs that is scalable to large arrays. The two primary sources of overhead associated with FPGA BIST, the test time and the memory required for storing the BIST configurations, are also reduced when compared to previous FPGA-BIST approaches. The BIST techniques presented here also eliminate the need for using any of the user I/O pins, such as a clock, a reset, and test observation pins; therefore, it is suitable for immediate deployment on any system with Xilinx 7-Series FPGAs. With faults detected, isolated, and corrected, the effective MTBF of a system can be extended.

Dedicated to,
My parents…

Acknowledgements

This thesis work would not have been possible without the support of many people. I would like to thank my advisor Dr. Peter Athanas for giving me an opportunity to work in Configurable Computing Machines lab. The time that I have spent in the CCM lab to gain the knowledge and to learn various skills in the field of FPGAs will surely be a valueable experience throughout my life. He has always been motivational to me not only to learn the technical skills but also to improve my communication skills. The skills that I could learn and improve under his guidance will definitely help me to be a better person in future.

I would like to thank Dr. Michael Hsiao and Dr. Carl Dietrich for being in my committee. The learning from the classes of *Testing and Verification of Digital Systems* and *Electronic Design Automation* that I took under Dr. Michael Hsiao has helped me a lot during this work. Apart from his teaching in the class, the short stories that he told in the class about his past research experience have truly been inspirational to me. The keen interest that Dr. Carl Dietrich took in my work whenever I met him has really been motivational.

The story of my experiences at the Blacksburg would be incomplete without the amazing friends that I have got. I would like to thank all my friends for all the cherrishing days that we have spent together. I will also like to thank all my friends outside the Blacksburg

who always have stayed in my heart and gave me the courage to do better in my life. I would also like to thank all my labmates of CCM lab who patiently listened to my problems, and helped me to find the solutions for them.

The words may fall short to express my gratitude to my parents and my family who have always helped and motivated me to be a better person. Any thing that I could do in my life so far would have been not possible without the unconditional love that they showered on me.

I would like to thank Matt French and Neil Steiner at USC-ISI for their guidance and support for this work. This work was supported in part by NSF I/UCRC IIP- 1266245 via the NSF Center for High- Performance Reconfigurable Computing (CHREC). Some of this research was developed with funding from the Defense Advanced Research Projects Agency (DARPA).

- Harmish Rajeshkumar Modi

June, 2015

Contents

List of Figures

List of Tables

List of Abbreviations

FPGA - Field Programmable Gate Array

CLB - Configurable Logic Block

ILA - Iterative Logic Array

BIST - Buil-In Self-Test

TPG - Test Pattern Generator

ORA - Output Response Analyzer

BUT - Block Under Test

JTAG - Joint Test Action Group

TORC – Tools for Open Reconfigurable Computing

HDL - Hardware Description Language

XDL - Xilinx Design Language

CAD - Computer Aided Design

API - Application Program Interface

Chapter 1

Introduction

Advancement in VLSI technology has led to the ever-increasing density of logic resources in Field Programmable Gate Arrays (FPGAs), and also a significant improvement in clock speeds at which the logic resources can operate. These factors have fueled the computational capacity of FPGAs without significant increase in its power consumption. However, on the other hand, scaling down of transistor sizes, higher switching speeds and reduction in core voltage has drastically affected the vulnerability of these devices to faults.

The faults in FPGAs can be broadly classified into two categories. The first category is soft-errors [1], such as single bit flip in configuration memory of the FPGA, alternatively known as Single Event Upsets (SEUs). The primary source of soft-errors is radiation effects present around FPGAs that are used in nuclear, space and aviation applications. The configuration memory of an FPGA can be thought of as a collection of bits, in which each bit controls a routing or a logic resource within the FPGA. For example, a configuration bit associated with a multiplexer controls the selection line of

the multiplexer. Similarly, a configuration bit in a switching matrix enables the connection between two wires. Any corruption in the configuration memory mimics incorrect behavior of the logic circuit. For example, corruption of a single bit in a LUT leads to incorrect implementation of the logic function in the LUT. These faults, by nature, are correctable. In order to correct these faults, the configuration memory is read back, and is compared with the golden image of the configuration memory. If a fault is detected, then the configuration memory is refreshed. Bitstream scrubbers are well known to combat these faults [2].

Another category of faults is permanent faults. These faults arise from various electrical effects, such as aging, electromigration and time dependent dielectric breakdown (TBBD) in a transistor. These effects eventually lead to an open circuit or a short circuit in the FPGA transistor. As a result of this, a particular net of a circuit behaves as if it is stuck at either a logic-1 or a logic-0, even when an input tries to pull the net in the other direction. Unfortunately, these faults cannot be corrected, and the only alternative is to avoid the faulty location. Fortunately, the FPGAs can be reconfigured to avoid the faulty locations given that the fault can be detected and isolated in some way. Any test that can detect such a fault without involving any dedicated external hardware is known as Built-in Self-test (BIST).

Two essential conditions have to be satisfied in order to detect the fault - the fault should be excited, and the fault effect should be observed. For example, consider a stuck-at-0 fault on one of the inputs of an AND gate. In this case, the fault can be excited by applying all of the inputs of the gate as a logic-1, and the fault effect will be observed as the output of the gate as a logic-0, whereas in ideal case the output should be a logic-1.

The conventional BIST facilitates both the generation of test vectors and the validation of the output response internally within the integrated circuit by inserting extra hardware; therefore, these approaches incur area overhead. However, the BIST for an FPGA exploits its reconfigurability to mitigate the area overhead. The conventional strategy behind the BIST for FPGAs is to configure some of the logic blocks inside the FPGA as Test Pattern Generators (TPGs), and others as Output Response Analyzers (ORAs). These elements carry out the test on the rest of the logic blocks, referred to as Blocks Under Test (BUTs). Once all of the BUTs are tested for all of their functionalities, then the role of the BUTs is swapped with the TPGs and ORAs through reconfiguration. In this way, the BIST tests the entire FPGA in two test sessions.

The BIST for the FPGAs, henceforward referred to as FPGA BIST, is performed off-line and is independent of the application that is running on the FPGA. The test is performed by configuring the FPGA with a set of bitstreams, which are dedicatedly designed to perform the test on the FPGA. Once the testing is complete, the FPGA is reconfigured back into its normal operation. All of the configuration bitstreams that are required to perform the BIST are pre-computed and stored in external memory along with the user configuration. The sources of overhead associated with this approach are the external memory for storing the BIST configurations and a short system downtime required to perform the test.

1.1 Objectives

The primary goal of this thesis is to develop the BIST architecture to test all of the functionalities of Configurable Logic Blocks (CLBs) in Xilinx 7-Series FPGAs. The 7-

Series FPGAs contain more advanced CLBs when compared to previous generation of Xilinx FPGAs; therefore, the BIST architecture discussed here can easily be extended to any of the previous generation Xilinx FPGAs. Moreover, Xilinx 7-Series FPGAs contain more CLBs when compared to previous Xilinx FPGAs, and, hence, this work is scalable to test any of the previous generation Xilinx FPGAs. Although the overall work presented in this thesis is directed to enhance the applicability of the test on systems that use Xilinx 7-Series FPGAs, the proposed BIST architecture is generic to any FPGA architecture.

It is highly desired that all of the CLBs in the FPGA be tested simultaneously in order to minimize the number of test sessions and test time. This implies that a typical BIST configuration may use the maximum possible number of CLBs in the FPGA, if possible all of the CLBs. With such a high resource usage, routing congestion becomes a serious problem [3]. The study of past BIST architectures reveals that the problem of routing congestion puts an upper bound on the number of CLBs that can be tested in a single test session. If all of the CLBs cannot be tested simultaneously, then the FPGA is divided into partitions, and each partition is tested one by one in different test session. As a result of this, the total test time multiplies by the number of test sessions. The BIST discussed in this thesis addresses the problem of routing congestion, and provides a solution to make the routing independent of the number of the CLBs used in the design. By doing this, all of the CLBs in the design can be tested in a single test session, and, hence, both the external memory for storing BIST configuration bitstreams and the system downtime during the testing process can be minimized.

The concept of Iterative Logic Array to test the repeated logic cells has been used in past work[4][5], and this technique has also been applied to the FPGA BISTs [3][14].

However, these approaches did not provide fault isolation. Fault isolation is necessary in order to narrow down the location of the fault within the FPGA. Such information is useful, particularly in the case of FPGAs, in order to exploit their reconfigurability for fault tolerance. Once the fault is isolated, it can be avoided by reconfiguring the FPGA with a new configuration. The BIST discussed in this thesis enables the fault isolation in the Iterative Logic Array based FPGA BISTs.

Usually, FPGA BIST assumes the availability of a limited number of user I/O pins, such as a clock, a reset and test observation pins, in order to administer the test. This implies that the BIST has to be designed prior to the system design phase of the project. The BIST presented in this work eliminates the need for any of the user I/O pins; therefore, it is applicable for immediate deployment on any system with Xilinx 7-Series FPGA.

The primary objectives of the thesis can be summarized as:

1) Design a BIST to detect and isolate single stuck-at-fault present in any of the Configurable Logic Blocks (CLBs) in Xilinx 7-Series FPGAs.
2) Minimize the primary sources of overhead associated with the FPGA BIST – the test time and the external memory for storing BIST configurations.
3) Reduce the routing complexity associated with BIST architecture. As a result of this, the test can be developed for any size of the device.
4) Eliminate the need for using user I/O pins, such as a clock, a reset and test observation pins, to administer the test. As a result of this, the BIST can be immediately deployed on in-use systems without making any system-level changes.

5) Enable the fault isolation in Iterative Logic Array (ILA) based BIST architectures.

1.2 Organization of Thesis

This thesis is organized in the following order.

- *Chapter 2* gives the overview of Xilinx FPGA architecture and previous work in the field of FPGA BIST.
- *Chapter 3* presents the system-level overview of the BIST. This chapter also discusses the BIST configurations at architecture level.
- *Chapter 4* gives the detailed description of the BIST configurations. The microarchitecture of the BIST configurations helps in understanding the fault coverage by the BIST configurations.
- *Chapter 5* discusses about various steps involved in the process of fault isolation.
- *Chapter 6* provides the details of the test generation process. Two alternative methods have been discussed in order to automate the generation of the BIST configurations. This chapter also provides the quantitative measurements of fault coverage and various sources of overhead that are associated with the FPGA BIST.
- *Chapter 7* concludes the thesis.

Chapter 2

Background and Overview

This chapter provides the necessary details of the Xilinx FPGA architecture and configuration bitstream in order to understand the details of BIST configurations, which are discussed in the later chapters. A comprehensive summary of previous FPGA BISTs is also presented in this chapter. This summary is useful to understand some of the factors that affect the selection of BIST architectures.

2.1 Overview of Xilinx 7-Series FPGAs

An FPGA consists of thousands of Configurable Logic Blocks (CLBs) arranged in a regular 2-D array as shown in Figure 2.1. Each CLB can be reconfigured to implement different logic functions. Each CLB also has an adjacent switch matrix that connects it to the rest of the FPGA. The number of CLBs in the FPGA varies across different devices of the same FPGA family. For example, the smallest Zync device (Zync is a SoC in Xilinx 7-Series FPGAs) has 2,200 CLBs, whereas the largest Zync device has 34,650 CLBs.

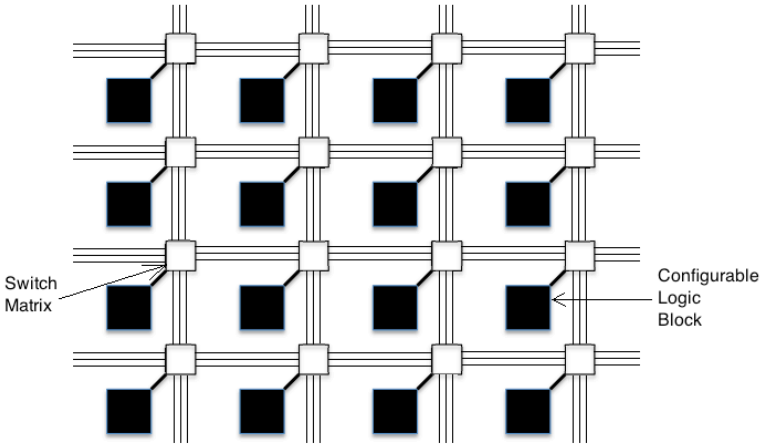


Figure 2.1: Arrangement of CLBs and Switch Matrices in Xilinx FPGA

2.1.1 Configurable Logic Blocks

CLBs are considered as the primary logic resources to implement combinational and sequential logic circuits. Each CLB contains two SLICE blocks [6], which in-turn are composed of four identical logic circuits, known as circuits A, B, C and D.

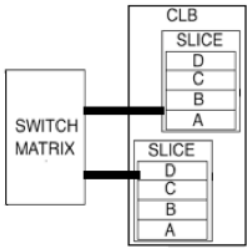


Figure 2.2: CLB in Xilinx FPGAs

The logic diagram of one of the circuits is shown in Figure 2.3. Each circuit contains a 6-input LUT, which can implement a single 6-variable combinational logic function or two independent 5-variable combinational logic functions. Each output of the LUT can also be independently registered into a flip-flop. Besides the LUT and the flip-flops, each circuit contains one XOR gate in order to facilitate the implementation of

arithmetic carry chain logic. The signals from one circuit can selectively be forwarded to the other circuits through various multiplexers present into horizontal and vertical data-paths. These functionalities are common to all of the SLICEs in the FPGA. Approximately one-third of the SLICEs, known as SLICEMs, have additional capability to use their LUTs as distributed RAMs and shift-registers.

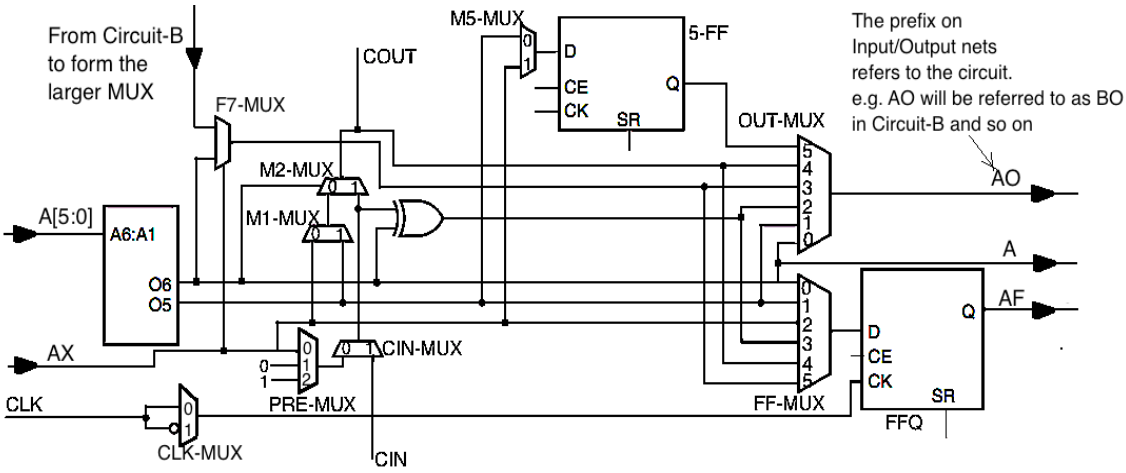


Figure 2.3: Logic diagram of Circuit A in a SLICE. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

2.1.2 Routing Resources

In contrast to fixed routing resources used in ASICs, the FPGAs contain programmable routing resources. The wires in the FPGA are preplaced, but different wire segments can be connected with each other using the programmable switch matrix. The switch matrix is a collection of thousands of tiny switches, each of which is controlled by a bit in configuration memory, and can connect two different wires passing through the switch matrix.

Each wire connects two CLBs, and is identified by the distance between the CLBs that it connects. As shown in Figure 2.4, single wires connect two adjacent CLBs, whereas double wires connect the CLBs that are two CLBs apart from each other. Similarly, quad and hex wires connect two CLBs that are four and six CLBs apart from each other respectively. Besides these wires, there are horizontal and vertical long wires that span across the device width and height.

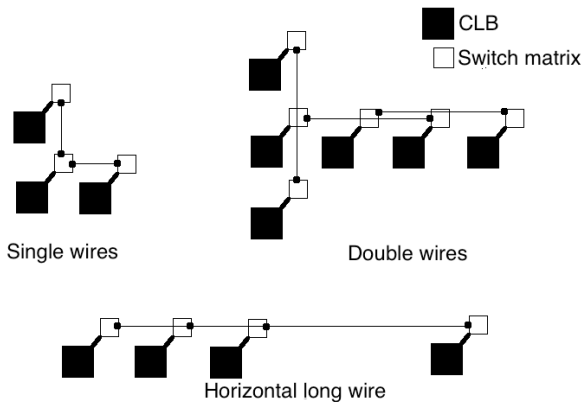


Figure 2.4: Wires in Xilinx FPGAs

Singles, doubles, quads and hexes are proportional to the number of switch matrices in the device, i.e. each switch matrix has certain numbers of these wires originating from it. As each CLB has an adjacent switch matrix, these wires are also proportional to the number of CLBs in the device. Moreover, the singles and doubles are more in number as compared to the quads and hexes. The long wires are available in limited quantity, and do not increase in proportion to the number of CLBs in the device. The single and the double wires are termed as local wires throughout the thesis because their role is to provide local routing. The rest of the wires are referred to as global wires.

2.1.3 Configuration Bitstream and Partial Reconfiguration

Implementation of different logic circuits on a single FPGA is made possible by the use of the configuration bitstream. Each bit in the configuration bitstream controls the state of a particular logic resource or a routing resource. The configuration bits associated with the CLBs are used as LUT content, selection channel of various multiplexers, and initial values of flip-flops. Each tiny switch inside a switch matrix that connects two different wire segments is controlled using a configuration bit.

Each configuration bit is assigned a logical address in order to perform read and write operations on it. The configuration bits are grouped together to form a frame in order to simplify the addressing. The length of the frame varies across different FPGA families. The frame is considered as the fundamental unit of the configuration bitstream, i.e. all reads and writes to the configuration memory are referenced by frame addresses. The frame address is determined by the physical location of a resource that the frame controls. All of the logical and routing resources in the FPGA are arranged in columns as shown in Figure 2.5. The position of the column in the device determines the major address of the frame. Each column contains multiple frames, each of which is identified by minor address. Each column can have different number of minor frames depending on the type of the resource that it contains.

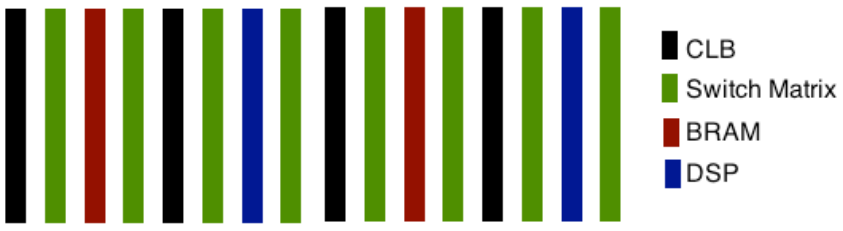


Figure 2.5: Columns in Xilinx FPGAs

Full configuration refers to the process of writing all of the frames to the device. The full configuration is a lengthy process, which takes around one to three seconds depending on the speed of the configuration interface for the largest Xilinx 7-Series FPGA. In order to minimize the configuration time, alternatively known as download time, the FPGAs are sometimes configured partially. If the incremental changes in the design are confined only to the particular locations in the FPGA, then the frames corresponding to only those locations are likely to change. In such scenarios, instead of writing all of the frames to the FPGA, only the changed frames are written to the FPGA. The process of writing the configuration memory partially is known as partial configuration of the FPGA [7].

2.2 Previous Work

The concept of BIST for the FPGAs was first introduced by Stroud [8]. The conventional strategy behind the FPGA BIST is to configure each CLB in the FPGA into Test Pattern Generator (TPG), Output Response Analyzer (ORA) or Block Under Test (BUT). The function of the TPG is to provide necessary input vectors to the BUTs in order to excite the faults. If a fault exists in the BUT, then the response of the BUT deviates from an ideal response, and this is validated in the ORA. Each configuration bitstream configures the BUTs in one of their possible modes, and, hence, multiple configuration bitstreams are needed to test all of the functionalities of the BUTs. Once the BUTs are tested for all of their functionalities, then the role of the BUTs is swapped with the TPGs and ORAs through reconfiguration. This way, the entire FPGA is tested in multiple test sessions. The arrangement of TPGs, BUTs, and ORAs is the key factor in determining the number

of test sessions required to test the entire FPGA. Two kinds of arrangement have been prevalent in all of the previous work. This section presents the brief overview of these two arrangements rather then discussing the details of all of the previous work.

2.2.1 BIST Architecture using Global TPG

Figure 2.6 shows the diagram of this BIST architecture. The term "global" is used to highlight the global nature of the TPG, i.e. a single TPG provides the input vectors to multiple BUTs in the FPGA. The TPG is configured to provide all of the necessary test vectors to perform pseudo exhaustive testing [13] of the BUTs. In the given configuration, all of the BUTs are configured identically, and the outputs of two alternate BUTs are compared with each other in adjacent ORAs. If identical BUTs are provided with a similar input vectors, then the outputs of both of the BUTs should ideally be identical. If a fault exists in any of the BUTs, then it will be observed in the ORA as unequal responses of the BUTs. The alternate BUTs, whose responses are compared in the same ORA, are provided the input by different TPGs so that the faulty BUT doesn't escape by the faulty TPG. In a single test session, different configurations test different possible modes of the BUTs. In another test session, the role of the BUTs is swapped with the TPGs and ORAs through reconfiguration, i.e. the CLBs that were configured as the TPGs and ORAs will now be configured as the BUTs and vice versa.

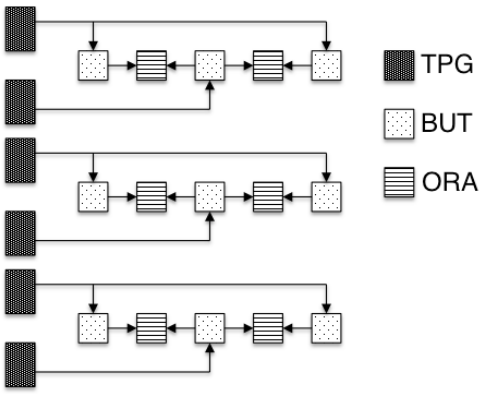


Figure 2.6: BIST using global TPG

In this architecture, half of the CLBs are configured as the ORAs. As a result of this, at maximum, only half of the CLBs can be configured as the BUTs, and, hence, only those many CLBs can be tested simultaneously in a single test session. Moreover, because of the "global" nature of the TPG, it is difficult to test the entire device in a single test session. It was shown in [3] that the routing congestion problem arising as a result of the "global" nature of this architecture strictly limits the scalability of this BIST architecture. As the device size increases, a single TPG has to provide the inputs to increasing number of BUTs, thereby the fan-out per TPG increases. Additionally, the TPG provides the inputs to the distant BUTs; therefore, routing heavily relies on long wires. Limited availability of the long wires, synchronization difficulties related to them, and high fan-out per TPG reduces the probability of successful routing in the larger devices. Modern FPGAs have the number of CLBs on order of 10^4; therefore, it is clearly evident that this architecture is not suitable for modern FPGA architectures.

Several examples of BIST have been published using this architecture, however, almost all of these BISTs were published for the previous generation FPGA architectures

such as Xilinx 4000, Spartan, Virtex-I, Spartan-II and Virtex-V architectures[9][10] [11][12]. However, because of the limitations due to routing congestion, this approach, in general, is less attractive to create the test for modern FPGAs.

2.2.2 Iterative Logic Array (ILA) Architecture

The primary motivation for coming up with the ILA architecture was to combat the routing congestion problem. The ILA, as it name implies, is the cascaded connection of similar logic cells. Each row of the FPGA is configured as the 1-D ILA as shown in Figure 2.7. In this architecture, the TPG provides inputs to only the first BUT in the array instead of providing inputs to all of the BUTs. The output of one BUT is provided as the input to its successor; this way, each BUT acts as a local TPG for its successor. If a fault is present in any of the BUTs, then that BUT will provide incorrect inputs to its successive BUT, and in this way, the error propagates through the entire array until it reaches the final ILA output. The ORA observes the final ILA output, and determines the result of the test.

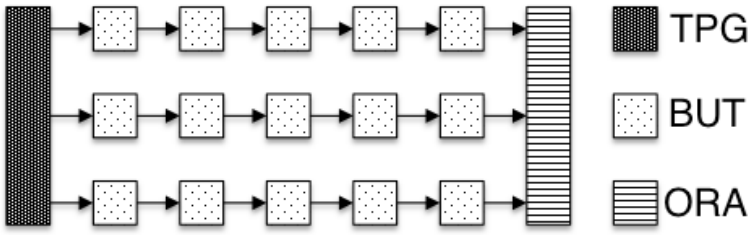
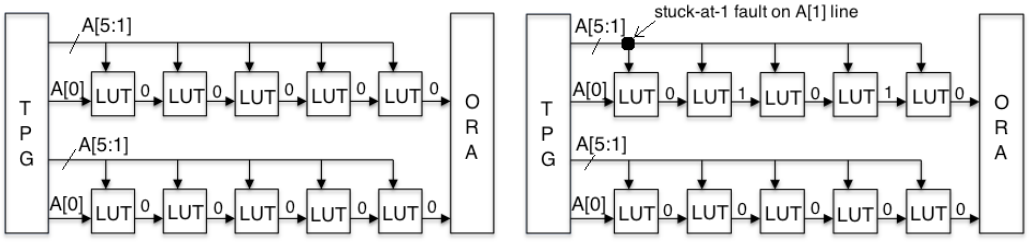


Figure 2.7: Testing of FPGA using multiple 1-D ILAs

As each BUT provides inputs to only its successor BUT, the average fan-out requirement per output pin reduces drastically. Additionally, the successive BUTs are placed adjacent to each other in order to constrain the routing locally. The switch matrix

that is available adjacent to each of the CLBs can easily accomplish local routing without using long wires; therefore, the problem of routing congestion reduces considerably.

However, in the ILA architecture, it is challenging to satisfy both the controllability of the inputs and the observability of the outputs in each BUT. For example, consider the testing of LUTs using the ILA architecture as shown in Figure 2.8. Each LUT has six inputs and only one output; therefore, each LUT can control only one input of its successor. In one approach [14], a common shared bus was used to provide the remaining inputs to each LUT. Because of the global nature of the shared bus, this approach also had the similar routing problems as discussed in the previous BIST architecture.



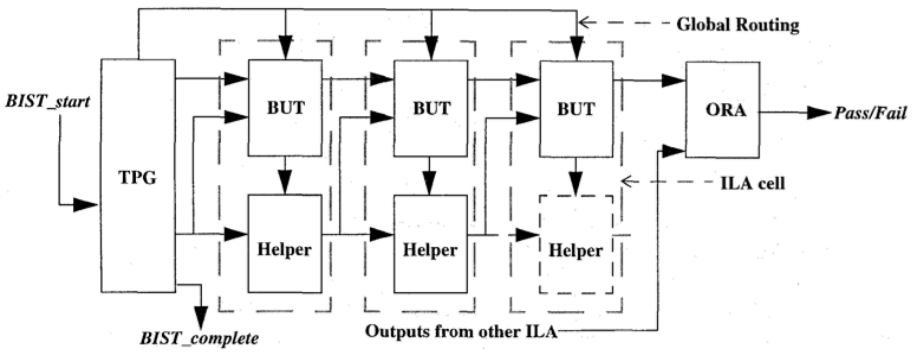
(a) ILA formation using shared bus (b) Fault masking in ILAs

Figure 2.8: An Example of fault masking in an ILA with shared bus

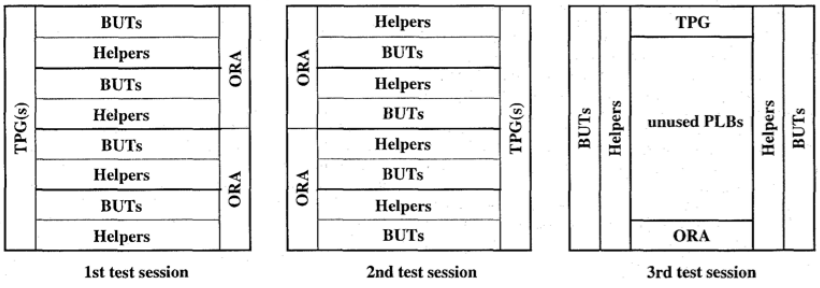
Another problem with the ILA architectures that uses the shared bus is fault masking. For example, while testing the LUTs using the shared bus as shown in Figure 2.8, the LUTs are configured to implement XOR function of its inputs [15]. Consider that input A[5:1] is set to 5'b00000, and input A[0] is set to 1'b0. In an ideal case, the output of both of the rows in the ILA is a logic-0, and, hence, the ORA finds equal response from both of the rows. However, consider a fault on the shared bus as shown in Figure

2.8(b). Such a fault can occur, if a switch in the switch matrix that connects two wires has a stuck-at-1 fault. Because of this fault, the subsequent LUTs in the first row will have their input A[5:1] as 5'b00001. The output of each LUT is also shown in the figure. It can be seen that even in the presence of this fault, the output of the first row is still a logic-0, and hence, the fault on the shared bus is not visible to the ORA. Huang et al. [16] used the XOR based tree as an local ORA to compare the output of two adjacent BUTs, and to propagate the faulty response. However, this technique also used shared bus, and requires an additional test session in which the roles of the local ORA and the BUTs is swapped.

Stroud et al. [3] reduced the number of shared (global) inputs in the ILA by using an extra helper cell for each CLB. The function of the helper cell is to provide missing inputs to the intermediate BUTs in the ILA. Multiple 1-D ILAs were created, and each ILA spanned across a single row of the FPGA as shown in Figure 2.9(a) (Figure is taken from [3] for illustration). Each of the ILAs is provided an input by the TPGs that are implemented at the edge of the FPGA. The output of two identical ILA rows is compared in the ORA in order to identify the presence of a fault. This architecture requires at least two test sessions to test all of the CLBs in the device because in a single test session, half of the CLBs have to be configured as the helper CLBs. The actual work used an additional third test session to test the TPGs and the ORAs. The floor plan used in each of these test session is shown in Figure 2.9(b).



(a) ILA formation with helper cell



(b) Floor-plan of different test sessions

Figure 2.9: ILA formation using helper cells. Stroud C., Lee E., Konala S., Abramovici M., "Using ILA testing for BIST in FPGAs," in proceedings of International Test Conference, 1996. Used under fair use, 2015.

In another approach [17] to test the CLBs, the FPGA is configured as an ILA multiplier, and then pseudo exhaustive test patterns were applied to the ILA. This model, however, is not a directed test, and could provide only limited fault coverage in the CLBs. Huang and Lombardi [18] configured the FPGA as multiple parallel 1-D ILAs, and then used different configurations to test the different modes of the CLBs. This method used the external test vectors, and, hence, the test is suitable on the systems where the environment to provide the external test vector to the FPGA exists.

Chapter 3

System Overview and BIST Architecture

This chapter provides the high-level overview of the test design process. First, the system-level overview of the test setup is discussed in detail. The later sections in this chapter provide an overview of the BIST architecture that is used to design the test. The detailed discussion of the BIST architecture provides the necessary background to understand the BIST configurations that are discussed in later chapter.

3.1 System Overview

Figure 3.1 shows the system-level diagram of the test setup. A limited number of assumptions are made in order to maximize the applicability of the test. The test assumes that there exists an external programming agent, referred to as a Host, to administer the test. The host needs to have some kind of configuration connection to the FPGA configuration port. The configuration ports supported by this project are SelectMAP and JTAG. The host doesn't need any other connection to the FPGA and the FPGA should also not have any special connection except a properly connected power supply.

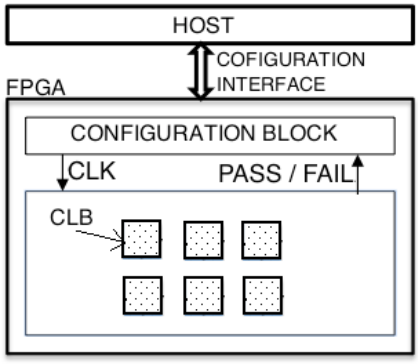


Figure 3.1: System-level diagram of test setup

The primary objective of the system design is to eliminate the need for using user I/O pins, such as a clock, a reset, and test observation pins, to perform the test. By eliminating the need for user I/O pins, the test can be immediately deployed on already in-use systems that don't have the user I/O pins available for implementing the BIST environment. All of the test administration tasks are performed through the FPGA configuration interface. The required signals to administer the test are derived through `STARTUPE2` hardware primitive provided by Xilinx [19]. This primitive provides an interface between the user logic and the FPGA configuration control and status signals. The internal ring oscillator present in the FPGA configuration block provides a 65MHz internal clock on `CFGMCLK` pin of the `STARTUPE2` primitive. This clock is used as the primary clock for the entire test.

Final result of the test indicates if there exists a fault in the FPGA or not; therefore, it can be stored in only a single bit. The `DONE` bit of the FPGA status register is "hacked" for this purpose. During the normal operation of the FPGA, the `DONE` bit is controlled through the FPGA configuration control logic, and it goes high to indicate that

the configuration process has completed successfully. However, the STARTUPE2 primitive provides a way to control this bit through the user logic instead of the configuration control logic. The USRDONETS signal on the STARTUPE2 controls the driver for the DONE bit. If the USERDONETS signal is driven active-high, then the value of DONE is controlled through the configuration logic. If the value of USERDONETS is set to active-low, then the value of DONE is controlled through the user logic. To write the test result into the DONE, the value of USERDONETS is kept active-low, and the test result is written through the USERDONEO signal. The value of DONE bit is visible in the FPGA status configuration register. This register can be read through the configuration interface as shown in Figure 3.1.

3.1.1 Host

The host is responsible for administrating the entire test. The host can be an external agent such as a PC with JTAG cable or it can also be an internal agent such as the ARM core (running Linux) in Zync-7000 series.

The host should have following capabilities:

- Need access to external storage of enough capacity to store BIST configuration bitstreams for each FPGA (the configuration bitstreams are pre-computed off-line, so the host does not need to be capable of executing any significant EDA software). The current storage requirement is in the range of mega-bytes, depending on the size of the target FPGA.
- Have access to an API controlling the FPGA configuration port. The following are the minimum requirements:

- `bool DownloadBitstream(filename)`: this should be able to download a bitstream and indicate when the bitstream has started up.
- `word ReadStatusRegister()`: this should be able to read (poll) the STATUS configuration register (register 0x7). Only the DONE bit is actually needed.
- `ReadBack()`: this capability is required for fault isolation. This is discussed in detail in Chapter 5.

3.1.2 Test Procedure

The host initiates the test and it performs the following sequence of steps on each BIST configuration bitstream.

1) The host programs the FPGA with the BIST configuration bitstream using the configuration interface and `DownloadBitstream(filename)` API.
2) The test runs for a short duration and the test result is written into the FPGA status register.
3) The host uses `ReadStatusRegister()` API to read back the test result through the configuration interface.
4) If a fault is found in Step-3, then the host performs fault isolation. The host reads back the state of the FPGA configuration memory using `ReadBack()` API, and further analyzes the captured state to narrow down the fault location.

All of the BIST configurations are pre-computed using a script, and are stored in external memory. The script takes the FPGA part name and the rectangular coordinates as the input parameters, and then generates all of the required BIST configurations to test all

of the CLBs within the specified rectangular area. Chapter 4 discusses the details of each BIST configuration, and Chapter 6 discusses the script that generates the BIST configurations.

3.2 BIST Architecture

The primary objective of the project is to create the BIST environment for Xilinx 7-Series devices. However, the BIST architecture has been selected in such a way that it can be easily extended to any other Xilinx device. The term BIST architecture in abstract terms refers to the arrangement and the interaction between TPGs, BUTs and ORAs. The selection of the BIST architecture is an important aspect of the design because it determines the number of test sessions to test the entire FPGA and the scalability of the test. If the BIST architecture is designed such that only some fraction of the CLBs can be tested simultaneously [3], then the FPGA has to be tested in multiple test sessions, in which each test session tests the different subset of the CLBs. As a result of this, both the test time and the external memory to store the BIST configuration bitstreams are multiplied by the number of test sessions, and, hence, one factor that affected the selection of the BIST architecture was to minimize the number of test sessions.

Different Xilinx devices contain different number of CLBs; therefore, the scalability of the BIST architecture is an important factor in order to ensure the versatility of the test. Scalability is largely dependent on the routing complexity of the BIST configurations; therefore, the BIST architecture should be designed in such a way that the routing complexity is reduced as much as possible. Another desired property for the FPGA BIST is to test the maximum possible number of the CLBs simultaneously. This

means that the usage of the CLBs in a typical BIST configuration should be maximum, if possible 100%. Routing of such a resource rich design is a significantly difficult task. Moreover, in order to leverage the capabilities of the router provided by the Xilinx in their PAR tool, the behavior of PAR was studied [20]. While discussing the routing resources in Section 2.1.2, it was highlighted that the single, double, quad and hex wires increase in proportion to the number of CLBs in the device. It was found that the numbers of single, double, quad and hex wires that originate from a switch matrix are 36, 32, 8 and 24 respectively. A simple experiment was carried out to understand how the length of the wire impacts the routability. In the first phase of the experiment, a large number of nets were created between every two successive CLBs in the FPGA. By doing so, the usage of single wires can be maximized during the routing. Similarly in the second phase of the experiment, the nets were created between the pair of CLBs, which are at a distance of two CLBs, to study the impact of the double wires on the routing. In this way, the impact of using the different wires of different lengths on the routability was studied. It was observed that while routing the design with longer nets the router takes more time when compared to routing the design with shorter nets. By this study, it was found that by maximizing the usage of the single and the double wires, the routing complexity could be reduced considerably.

Another aspect that affects the routing complexity is the average fan-out in the design. It was shown[3][21] that high fan-out requirement on a single TPG affects the routability of the design. In order to simplify the routing problem, the average fan-out in the design should be reduced as much as possible [20].

3.2.2 Phases of Test

As discussed in Chapter 2, the CLBs in Xilinx FPGA consist of basic logic elements namely LUTs, multiplexers and flip-flops. Different logic elements are tested in different phases of the test as follows:

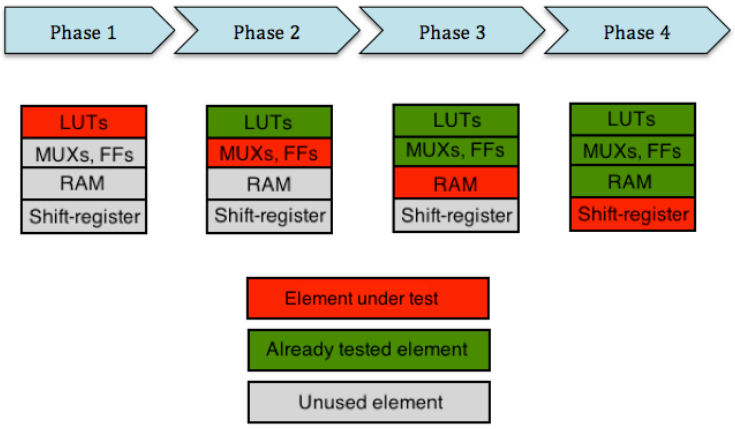


Figure 3.2: Different phases of testing

1) LUT testing: All memory locations, input address pins, and output pin O6 of the LUTs are tested for a stuck-at-1 and a stuck-at-0 fault during this phase. Any fault in the address decoder logic of the LUT is also covered during this phase. No other component in the CLB is used while testing the LUTs, i.e. the output of the LUT is not propagated through any other component in the CLB.
2) Testing of multiplexers, flip-flops and XOR gate: All inputs, outputs, and selection channels of various multiplexers are tested in this phase. All of the gate-level faults in the multiplexers are also covered during this phase. During this phase, the LUTs that have been found to be fault free are used to test the multiplexers, flip-flops, and XOR gate.

3) RAM testing: A subset of SLICEs, typically one-third of the total SLICEs [6], has capability to use their LUTs as distributed RAM. These SLICEs are known as SLICEMs. The RAMs are tested using MARCH test sequences [22]. The logic resources that are found to be fault free in the previous two phases are used as the complementary resources while testing the RAMs.
4) Shift-register testing: The shift-register functionality of the SLICEMs is tested in this phase.

Phase 1 and Phase 2 use the ILA architecture in order to test all of the SLICEs in the device in a single test session. The ILA architecture is not suitable for error propagation while testing the RAMs; therefore, the RAMs are tested using another architecture, referred to as Distributed TPG Architecture. The following sections provide the details of each of the architectures.

3.2.3 ILA Architecture

The primary reason behind selecting the ILA architecture is to reduce the problem of routing congestion. It was discussed in Chapter 2 that the BIST architecture using the global TPG cannot test all of the CLBs in the device in a single test session due to the routing congestion problem. The regular arrangement of the CLBs and the switching matrices in the FPGA provides an excellent opportunity to make the maximum use of the local routing resources present between the two adjacent CLBs rather then relying on the global routing resources. All of the CLBs within the device are cascaded as shown in Figure 3.3 to form the Iterative Logic Array (ILA). The TPG provides the input to only the first CLB in the ILA, and the output of each CLB is provided as the input to its

adjacent CLB. The adjacency between the CLBs is measured by the connectivity between them. The study of the routing resources reveals that the two successive CLBs in the same column have the maximum connectivity, and, hence, the successive BUTs in the ILA are always placed in the column first order.

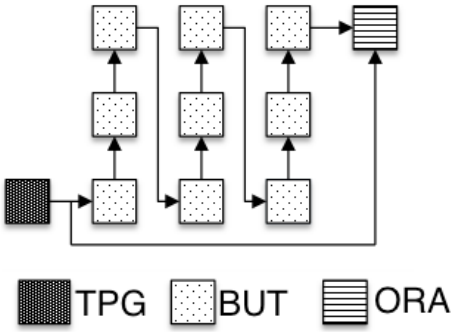


Figure 3.3: 2-D ILA architecture

In order to cascade the CLBs, the output bus width of one CLB should match the input bus width of its successor CLB. However, each CLB has a different number of output pins as compared to the number of input pins [6]. To combat this problem, in each configuration, a subset of input lines and a subset of output lines are selected such that the output bus width of the CLB matches the input bus width of its successor CLB. The rest of the inputs of the CLB are tied to either a local GND or to a local VCC. It is found that each switching matrix contains a TIEOFF site that can provide both the GND and the VCC connections to the adjacent CLB. By selecting the different subset of inputs in each configuration, the different subsets of logic nodes within the CLB are tested in each configuration. A set of BIST configurations cumulatively tests all of the logic nodes within the CLB. The inputs, which are propagated from the predecessor CLB, are

referred to as non-controlled inputs. The rest of the inputs, which are statically tied to a logic-1 or a logic-0, are referred to as controlled inputs. This is explained by an example in Figure 3.4. By selecting the subset of inputs (AX, BX, CX and DX) and the subset of outputs (AO, BO, CO and DO), the input bus-width is kept same as the output bus-width. By doing so, the ILA can be formed by connecting the outputs AO, BO, CO and DO of one SLICE to the inputs AX, BX, CX and DX respectively of its successor SLICE. In this way, the AX, BX, CX and DX inputs of the SLICE (abstract view of Figure 2.3) are the non-controlled input, and all other inputs A[5:0], B[5:0], C[5:0] and D[5:0] are the controlled inputs.

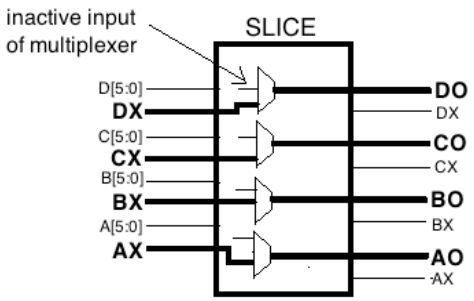


Figure 3.4: Example of controlling and non-controlling inputs

Another important challenge with the ILA architecture is to propagate a fault to the ORA without masking it. The non-controlled and the controlled inputs are selected such that the possibility of masking a fault that is propagated from the predecessor CLBs is eliminated. Once the CLBs are cascaded, then the final output response of the ILA is a cumulative function of all of the individual functions that are implemented in each of the CLBs. To simplify the design of the ORA, each CLB in the ILA is constrained to implement the identity function. The implementation of the identity function in each CLB is discussed in detail in Chapter 4 while discussing the BIST configurations. In the above

example, the multiplexers are configured to pass the inputs AX, BX, CX and DX to the outputs AO, BO, CO and DO respectively, and, hence, the SLICE implements the identity function. In order to eliminate the masking of a fault in the multiplexers, the inputs A[5:0], B[5:0], C[5:0] and D[5:0] are tied to a VCC or a GND in such a way that the inactive input of the multiplexers is set to a logic-1. By keeping the inactive inputs of the multiplexer to a logic-1, the masking of a fault in the multiplexer can be eliminated. This is also shown in Chapter 4 (Section 4.2.1).

Once all of the CLBs are configured to implement the identity function, the final response of the ILA should be same as the input applied to the first CLB in the ILA. In such cases, the presence of a fault can be identified by comparing the final ILA output to the input provided by the TPG. This means that the ORA can simply be implemented as a comparator. The input vectors required in each BIST configuration are generated using an up counter. The TPG and the ORA are implemented in embedded DSP resources, and, hence, no CLBs are used to implement the TPG and the ORA. This way, all of the CLBs in the device can be tested in a single test session. The ILA architecture brings the following improvements as compared to the previous approaches:

- Eliminate the helper CLBs for the ILA formation. This way, the separate test session required for testing the helper CLBs is completely eliminated.
- Enables fault isolation in the ILA architecture.
- Provide the ILA formation for the state of the art Xilinx FPGAs.
- The previous ILA approaches used multiple 1-D ILAs, each implemented in the separate row of the FPGA. As a result of this, the architecture required multiple TPGs to provide inputs to each of the ILA and multiple ORAs to validate the

output of each row. As a significant amount of logic resources has been used as the TPGs and the ORAs, a separate test session was required to test the TPGs and the ORAs in previous ILA based approaches. The ILA presented in this thesis builds a single 2-D ILA. This means that only a single TPG and ORA are required, and, hence, the logic resources required to implement the TPG and the ORA are minimized. Moreover, the input vectors in all of the BIST configurations can be derived using an up counter. Each of the TPG and the ORA is implemented in a single DSP resource, thereby eliminating the separate test session to test the CLBs occupied by the TPG and ORA.

- Although this project doesn't use any of the user I/O pins, by implementing a single ILA, the total I/O pin count of an ILA can also be minimized. If multiple 1-D ILAs are created, then the total pin count multiples by the number of ILAs when compared to the pin count in the case of a single ILA. Minimizing the pin count can be useful in situations, where the TPG and the ORA are implemented in the embedded processor such as the one available in Zync-7000 series FPGAs.

3.2.4 Distributed TPG Architecture

The distributed TPG architecture is used to test RAMs that are present in a subset of SLICEs, known as SLICEMs. Xilinx specifies that the RAM functionality is usually available in only one-third of the total SLICEs. It was also found that these RAMs are distributed in the FPGA, and are not clustered over the particular area of the FPGA. That is the reason these RAMs are sometimes referred to as distributed RAMs in FPGA literature.

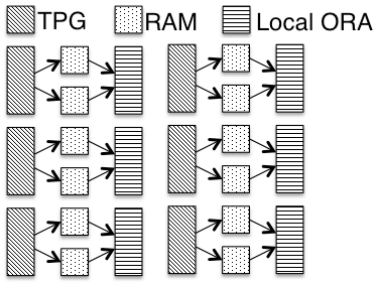


Figure 3.5: BIST architecture to test distributed RAMs

The SLICEs that don't have the capability to configure their LUTs as the RAM, and have already been tested by the ILA architecture are used to build multiple distributed instances of the TPGs as shown in Figure 3.5. Each TPG provides the required vectors to test the distributed RAMs. Each TPG is used to provide inputs to only the RAMs in the adjacent column as shown in Figure 3.5. The outputs of two RAMs, which are provided the input vectors by the same TPG, are compared with each other in an adjacent SLICEL in the same CLB. Such a comparator can easily be implemented in the adjacent SLICEL in the same CLB as shown in Figure 3.6. Multiple configurations are used in order to test each mode of the RAM in a separate configuration.

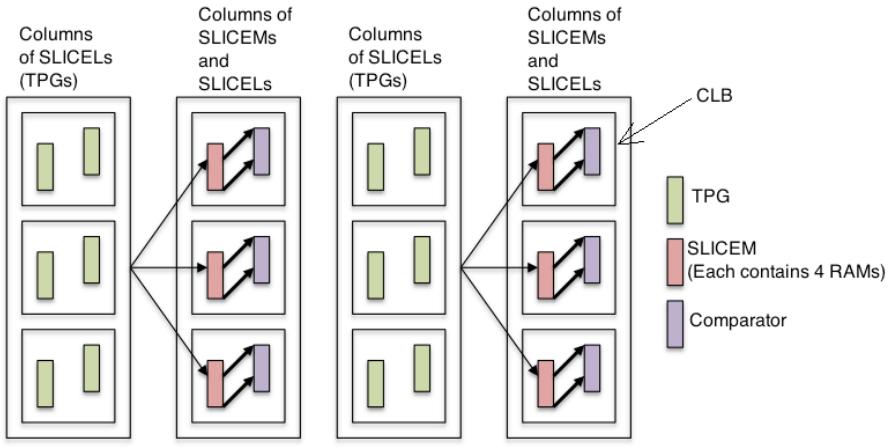


Figure 3.6: Arrangement of TPG, RAMs and local ORAs in FPGA columns

3.2.5 BIST Architecture for Shift-registers

The LUTs in the SLICEMs can be configured as a shift-register in addition to distributed RAM. All of the shift-registers can be cascaded in order to implement a longer chain of shift-registers. To verify the shift-register mode, the FPGA is divided into two regions. All of the shift-registers in each region are cascaded to form a long circular chain of the shift-registers, and this way, two long circular chains of the shift-registers are created. Each chain is initialized with the alternate 1s and 0s bit-pattern (1010…) in the configuration bitstream. During the test, both of the chains are provided same clock signal, and the 0^{th} bit of both of the chains is compared with each other as shown in Figure 3.7. If the test runs for enough time, then any fault present in the chain should eventually be observed at the bit-0. If the ORA finds unequal values in the 0^{th} bit of both of the chains in any clock cycle, then it is identified as the presence of a fault.

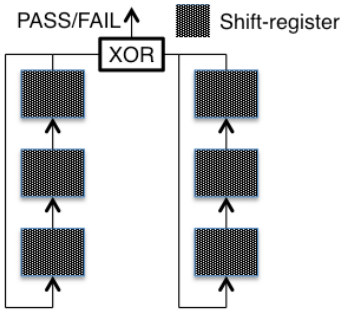


Figure 3.7: BIST architecture to test shift-registers

Chapter 4

BIST Configurations

This chapter provides the details of the BIST configurations that are used to test the FPGA. The necessary conditions to test all of the logic nodes within the CLB are illustrated, and the necessary input vectors in order to meet those conditions are also discussed.

4.1 LUT Testing

LUTs are used to implement the combinational logic functions in the FPGA SLICEs. Each LUT in Xilinx 7-Series FPGAs has six inputs. Internally, the 6-input LUT is implemented as two different 5-input LUTs, and, hence, each LUT can implement either a combinational function of six variables or two independent combinational functions of five variables.

Figure 4.1 shows the gate-level logic diagram of the 4-input LUT. The example of the 4-input LUT is discussed for illustration purpose, and later, the reasoning can be

extended for the LUTs with any number of inputs. The LUT contains memory cells and address decoder logic, and accordingly, the faults in the LUT can be classified into the faults in the memory cells and the faults in the address decoder logic.

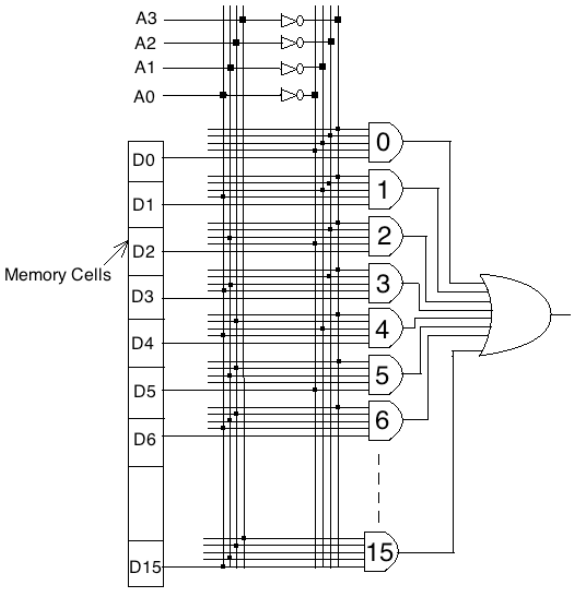
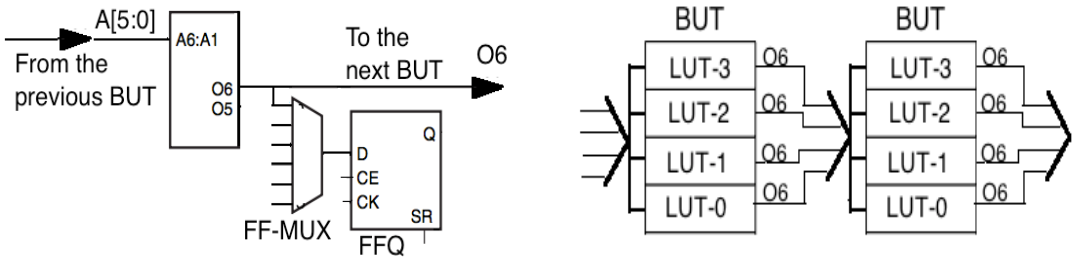


Figure 4.1: Gate-level logic diagram of the 4-input LUT

4.1.1 Faults in Memory Cells

In order to test a memory cell, the memory cell is programmed to a logic-0 or a logic-1, and then, the memory cell is readback to check if it contains the correct value or not. In order to test all of the LUTs in the FPGA simultaneously, the LUTs are cascaded to form an ILA. However, the cascading of the LUTs in an ILA is not straightforward because the LUT has different number of input pins as compared to the number of output pins. For example, the 4-input LUT has four inputs and one output. In order to cascade the multiple LUTs in an ILA, four LUTs are grouped together to form a BUT. The output of each LUT in the BUT is used to form an output bus of width four, which can be connected as

the input to its successor BUT as shown in Figure 4.2(b). The input of the BUT is shared across all of the LUTs in the BUT.



(a) Individual LUT in a BUT (b) Cascading of the LUT in an ILA

Figure 4.2: ILA formation for LUTs

The LUTs in a BUT are programmed in such a way that the BUT implements the identity function. In the example of the 4-input LUT, four LUTs can be programmed as follows such that the output of the BUT is similar to the input applied to the BUT.

$O(i) = A[i], \ 0 <= i <= 4$

where, $O(i)$ refers to the output of i^{th} LUT in a BUT

$A[i]$ refers to the i^{th} input to the BUT

These functions can be interpreted like this: If the input 4'b0000 is applied to the LUT, then the 0^{th} memory location in each LUT is selected. In order to get the output of the LUT as 4'b0000, the 0^{th} address location of each LUT should be programmed to a logic-0. This reasoning can be applied for each memory location in the LUTs, and the value of each memory location in the LUTs can be found. The LUT content for the above functions is specified in Table 4.1.

Table 4.1: Content of the LUTs in a BUT to implement the identity function

LUT address	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
LUT-0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
LUT-1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
LUT-2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
LUT-3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

The identity function covers either a stuck-at-0 or a stuck-at-1 fault in each memory location of the LUT, but it doesn't cover both the faults. For example, the identity function can activate only a stuck-at-1 fault in the 0^{th} memory location of each LUT, but it can't activate the other fault in the 0^{th} memory location. To test the alternate fault in each memory location of the LUT, the LUTs are programmed in such a way that each BUT implements the complement function. After implementing the complement function, if the input address 4'b0000 is applied to the LUTs, then the 0^{th} memory location of each LUT is selected. In order to get the output of the BUT as 4'b1111 (complement of the 4'b0000), the 0^{th} memory location of each LUT should be programmed as a logic-1. Table 4.2 specifies the contents of the LUTs in a BUT to implement the complement function.

Table 4.2: Content of the LUTs in a BUT to implement the complement function

LUT Address	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
LUT-0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
LUT-1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
LUT-2	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
LUT-3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

4.1.2 Faults in Address Decoder Logic

If there exists any fault on the output of the NOT gate in the address decoder, then it can be thought as the presence of a fault on the input of the AND gate. On the other hand, if there exists a fault on the input of the NOT gate, then it can be thought of as an opposite fault on the input of the AND gate. The faults in the AND gate and OR gate are discussed below.

Stuck-at-0 fault:

If there exists a stuck-at-0 fault in the address decoder logic (AND gates or OR gate), then the output of the LUT will be a logic-0 even when a memory location with data as a logic-1 is selected. Referring to Figure 4.3, consider the LUT-0 in a BUT that is configured to implement the identity function. If the gate-1 has a stuck-at-0 fault on any of its selection line, then it will be observed when the input vector 4'b0001 is applied to the LUT. In this case, the gate-1 will never be enabled, and, hence, the output of the LUT will be at a logic-0 even the data input of the gate-1 is set to a logic-1.

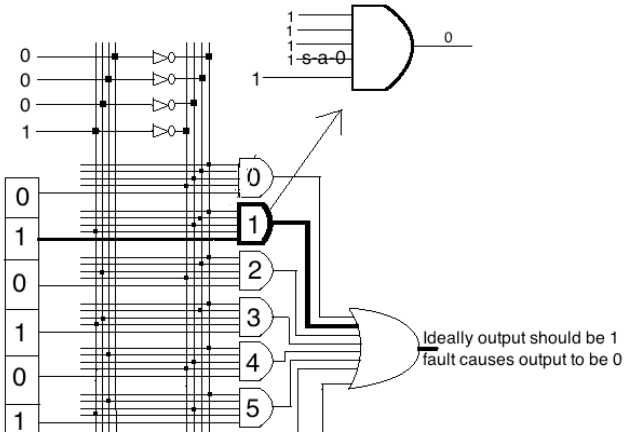


Figure 4.3: Stuck-at-0 fault in the address decoder logic of the LUT

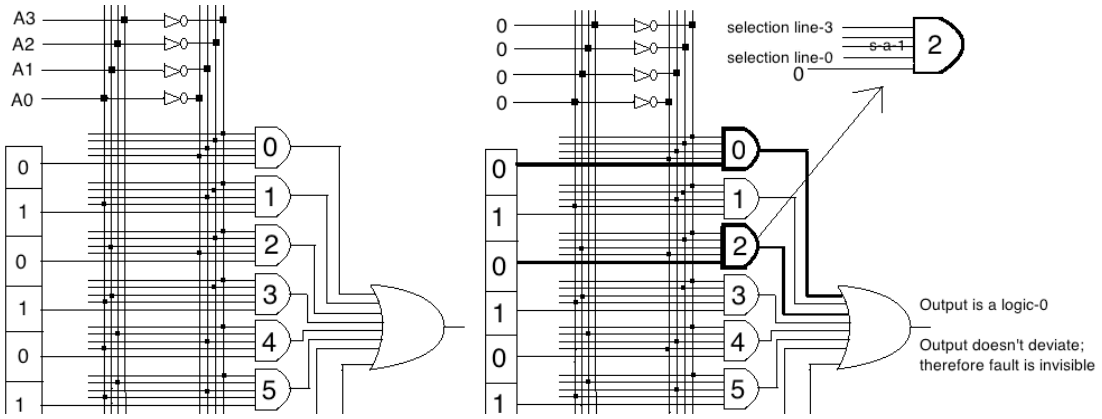
Stuck-at-1 fault:

If there exists a stuck-at-1 fault in the OR gate, then the output of the LUT will be a logic-1 for all input vectors even when the memory location with the data as a logic-0 is selected by the input vector; therefore, a stuck-at-1 fault on the inputs and the output of the OR gate becomes observable.

Observing a stuck-at-1 faults on the selection lines of the AND gates is slightly complicated. This becomes clearer with the following example. Figure 4.4(a) shows the logic diagram of the LUT-0 in a BUT that is configured to implement the identity function. In this case, an ideal output of the LUT should be a logic-0 when the input address 4'b0000 is applied to the LUT-0. Assume that there exists a stuck-at-1 fault on the selection line-1 of the gate-2 as shown in Figure 4.4(b). If the input 4'b0000 is applied to the LUT, then both the gate-0 and the gate-2 will be enabled. The output of the LUT has to deviate from an ideal value (logic-0) in order to observe the effect of fault that is present on the gate-2. However, as the data inputs of both the gate-0 and the gate-2 are configured as a logic-0, the output of the LUT will not deviate; therefore, the fault is not observed.

The important thing to notice here is that if there exists a stuck-at-1 fault on the i^{th} selection line of any AND gate, then it would result in enabling another gate, which is at the distance of 2^i from the selected gate, simultaneously. This is because in the input vector designated as {bit-3, bit-2, bit-1, bit-0}, if there exists a fault in i^{th} position, then it will result in another input vector, which should be at the distance of 2^i from the original input vector. In the above example, the selection line-1 (i = 1, bit-1 in the input vector) of the gate-2 has a stuck-at-1 fault, and, hence, the gate-2, which is at a distance of 2 (2^i =

2^1) from the gate-0, is also enabled when the input vector 4'b0000 is applied. In order to observe the effect of the fault in this case, the gate selected by the input vector (gate-0) should have the data input as a logic-0, whereas the gate enabled because of the fault (gate-2) should have the data input as a logic-1.



(a) Fault free LUT (b) stuck-at-1 fault on the selection line-1 of gate-2

Figure 4.4: Detection of a stuck-at-1 fault in the address decoder logic

If the pattern in the data inputs of the LUT-0 is observed in both of the configurations that implements the identity and the complement function, then it will be found that the data inputs of the alternate AND gates (distance of 1) have opposite values. This means that any stuck-at-1 fault on the selection line-0 (2^i =1 → i=0) of any AND gate in the LUT-0 will be visible. Similarly, in the LUT-1, the data inputs of the AND gates, which are at distance of 2, have opposite values. As a result, any stuck-at-1 fault present on the selection line-1 of any AND gate would be visible in the LUT-1. The same reasoning can be extended to all of the four LUTs. In order to test a stuck-at-1 fault on all

of the selection lines of all of the AND gates in all of the LUTs, the LUT functions are swapped among themselves as shown in Table 4.3.

Table 4.3: Swapping of the LUTs in a BUT to cover the faults in address decoder logic

Configuration	LUT-0	LUT-1	LUT-2	LUT-3
1	Distance-1	Distance-2	Distance-4	Distance-8
2	Distance-2	Distance-4	Distance-8	Distance-1
3	Distance-4	Distance-8	Distance-1	Distance-2
4	Distance-8	Distance-1	Distance-2	Distance-4

A total of 8 configurations (4 for identity functions and 4 for complement function) are needed in order to test all of the faults in the gate-level circuit of the LUT. For the LUT with n-inputs, a total of 2n configurations are needed to test all of the gate-level faults in the LUT.

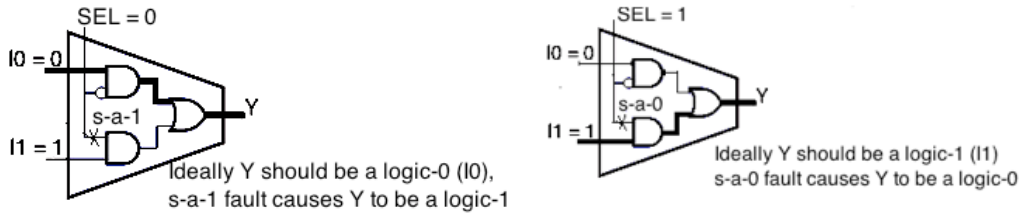
4.2 Testing of Multiplexers, Flip-Flops and XOR gate

Testing of the multiplexer, flip-flops and XOR gate is carried out by creating a data path through these components, and then by exciting the data path to a logic-0, and then to a logic-1. The data paths of one SLICE are cascaded to the other SLICE in order to form an ILA.

4.2.1 Necessary Conditions to Test Multiplexer

Usually, the multiplexer is thought of as a switch that can connect any of its input to the output depending on the values of the selection lines. Testing a multiplexer by enabling one data path through the multiplexer at a time, and then by toggling the data path between a logic-0 and a logic-1 is incomplete for testing the gate-level model of the multiplexer. Consider a gate-level model of the multiplexer, as shown in Figure 4.5(a).

Assume that there exists a stuck-at-1 fault on the selection line of the multiplexer as shown in the figure. In the presence of this fault, both the data paths will be enabled simultaneously, however, if the input I1 is kept to a logic-0, then the fault on the selection line is not visible to the output of the multiplexer. In order to detect the fault on the selection line, the active input I0 has to be kept to a logic-0, and the inactive input I1 has to be kept to a logic-1. By dong so, the output Y of the multiplexer would be at a logic-1, whereas in an ideal case, the output Y should be a logic-0, and, hence, the fault becomes visible in the multiplexer output. A stuck-at-0 fault on the input line or any of the selection line of any AND gate will be observable when the AND gate is selected by the selection lines and the input line is set to a logic-1 as shown in Figure 4.5(b). In general, while testing a data path through the multiplexer, all inactive inputs of the multiplexer should be kept to a logic-1 in order to cover the faults on the selection lines of the multiplexer. By testing all data paths through the multiplexer one after one, and by keeping the inactive inputs to a logic-1, all of the faults in the gate-level model of the multiplexer can be tested.



(a) Stuck-at-1 fault (b) Stuck-at-0 fault

Figure 4.5: Detection of a fault on the selection line of the multiplexer

4.2.2 ILA Formation for Data paths

Each SLICE is used to form a BUT in an ILA. As discussed in Chapter 3, in each BIST configuration, a subset of inputs, a subset of outputs and a subset of logic nodes are selected from the SLICE to form the ILA. Consider one of the configurations as shown in Figure 4.6. The outputs {DO, CO, BO, AO} forms the output bus, which is connected as the common input address (A[3:0]) to all of the LUTs in the successive BUT. These connections are symbolically represented as follows.

S0(AO) → *S1(A/B/C/D[0])*

S0(BO) → *S1(A/B/C/D[1])*

S0(CO) → *S1(A/B/C/D[2])*

S0(DO) → *S1(A/B/C/D[3])*

The first line can be interpreted as: AO pin of SLICE S0 is connected to the address pin-0 of all of the LUTs in successor SLICE S1.

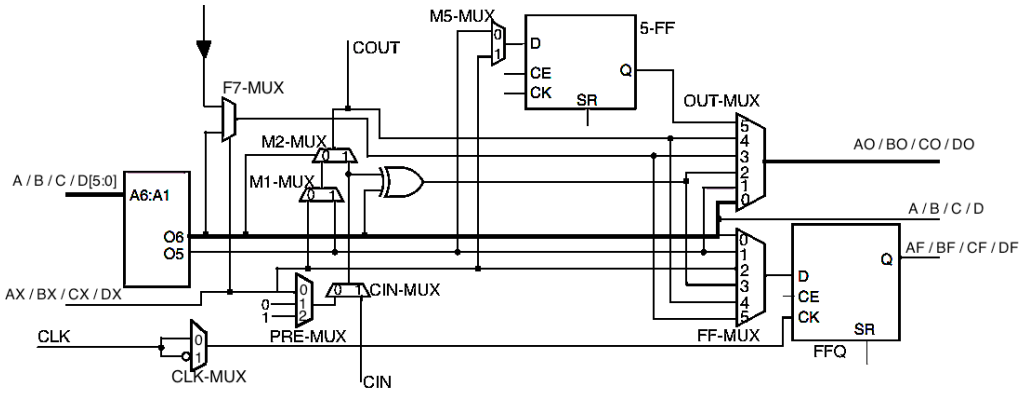


Figure 4.6: ILA formation for data path testing. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

Each BUT in an ILA is constrained to implement the identity function in a fault free condition. The data paths in the SLICE originate either from the DX, CX, BX and AX lines or from the LUT outputs. If the data paths are originating from the DX, CX, BX and AX lines, then implementing the identity function is straightforward. In this case, the inputs can simply be propagated to the output pins by just enabling the appropriate inputs of the multiplexers as shown by the highlighted path (in thick black) in Figure 4.7.

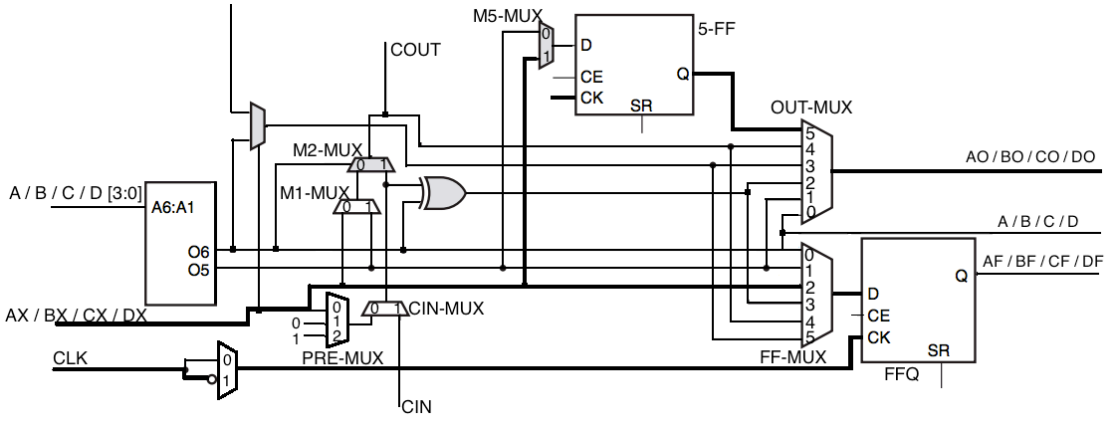


Figure 4.7: Testing of the data path that doesn't involve the LUT. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

If the data paths are originating from the LUT outputs, then the output of the predecessor SLICE can be connected as the LUT inputs in its successor SLICE, and then the LUTs can be constrained such that the SLICE implements the identity function. It should be noted that the LUTs have already been independently tested in the Phase-1, and, hence, the LUTs can be used to implement any functions. One such example is shown in Figure 4.6. The LUT functions that implement the identity function in the SLICE are as follows.

```
LUT-A: O6 = A[0]  //A[0] is the address pin-0 of the LUT-A
LUT-B: O6 = B[1]  //B[1] is the address pin-1 of the LUT-B
LUT-C: O6 = C[2]  //C[2] is the address pin-2 of the LUT-C
LUT-D: O6 = D[3]  //D[3] is the address pin-3 of the LUT-D
```

Here, the LUTs are simply configured as a pass-through. As there are only four propagated outputs and there are four LUTs in the successor SLICE, each LUT can act as a pass-through for one of the propagated outputs from its predecessor SLICE. By doing this, only one data path in the SLICE is tested at a time, and, hence, the total number of configurations required to test the SLICE is equal to the total number of data paths in the SLICE.

4.2.3 Input Vectors

To test a data path, the data path has to be toggled between a logic-0 and a logic-1. As circuits A, B, C and D are configured identical and are tested in parallel, all of them can be applied the similar input vectors simultaneously, i.e. all of the configured data paths can be excited by a logic-0, or a logic-1 at a time. That means that the input vectors 4'b0000 and 4'b1111 should be used to test the data paths. However, these vectors are not sufficient to test the vertical multiplexers, in which one of the inputs comes from the other circuit. One such example is shown in Figure 4.8. Consider the highlighted multiplexer in the figure. It was earlier discussed in Section 4.2.1 that in order to test a particular data path for a logic-0 through the multiplexer, the inactive inputs of the multiplexer should be kept to a logic-1. The inactive input of the vertical multiplexer in any circuit comes from the circuit below it. In this example, the inactive input for the multiplexer in the circuit B comes from the circuit A (indicated by green line), and,

hence, the circuit A and the circuit B should be tested by the input vectors which have opposite values. In other words, while testing the circuit B with a logic-0, the circuit A has to be tested by a logic-1. By this reasoning, the alternate circuits in a SLICE should be tested with the input vectors of opposite values, and, hence, the input vectors 4'b0101 and 4'b1010 are selected for testing the horizontal and vertical multiplexers. While testing the data paths that originate from the LUTs, the alternate LUTs in the SLICE should output the opposite values for the given input to the LUTs.

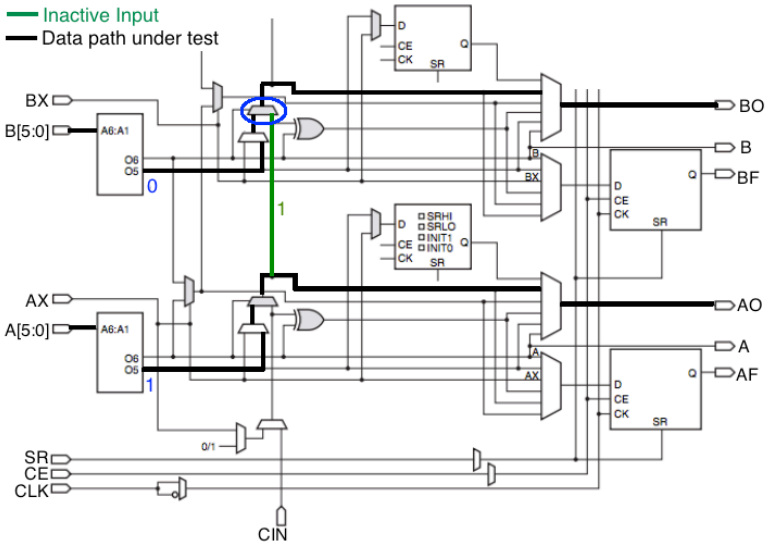


Figure 4.8: Testing of the vertical multiplexer. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

4.2.4 Merging Two Configurations

In order to reduce the total number of configurations, two data paths are tested simultaneously. If only a single data path is tested at a time, then only one output from the circuits A, B, C and D is propagated to the next SLICE. However, each circuit has

two independent outputs that can be propagated to the next SLICE; therefore at most two data paths from each circuit can be tested simultaneously. Consider an example as shown in Figure 4.9. Instead of testing the output O6 from AOUTMUX and AFFMUX separately in two configurations, both of them can be tested simultaneously. The new connections between the two successive SLICEs can be described as follows.

S0(AO, AF) → *S1(A/B[0], C/D[0])*

S0(BO, BF) → *S1(A/B[1], C/D[1])*

S0(CO, CF) → *S1(A/B[2], C/D[2])*

S0(DO, DF) → *S1(A/B[3], C/D[3])*

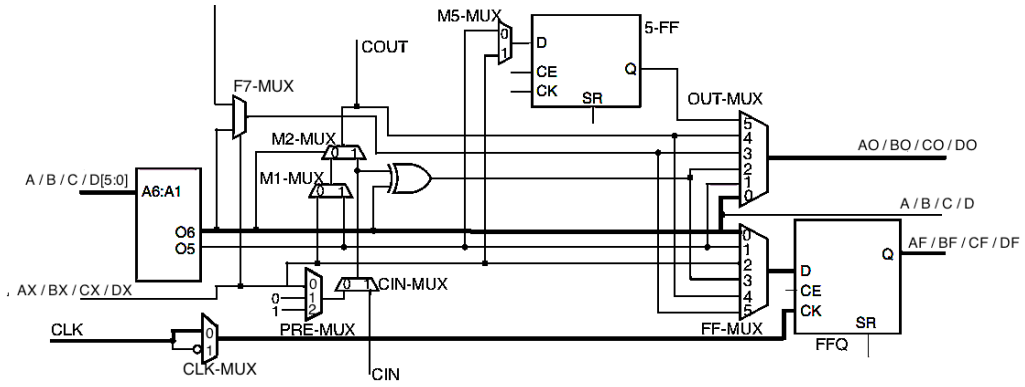


Figure 4.9: Testing of two data paths in a single configuration. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

These connections can be interpreted like this. The output AO of the SLICE S0 is connected as the common input to the 0^{th} address pin of the LUT-A and the LUT-B (A/B[0]) in the successor SLICE S1, and the output AF of the SLICE S0 is connected as

the common input to the 0^{th} address pin of the LUT-C and the LUT-D (C/D/[0]) in the successor SLICE S1. The required LUT functions are discussed in the next section.

4.2.5 LUT Functions for Testing Data paths

If eight outputs from the SLICE are propagated to the next SLICE, then each successor SLICE will have eight inputs. However, as there are only four LUTs in the SLICE, the LUTs cannot be configured as the pass-through functions as implemented earlier. A different function has to be configured into the LUT such that it meets the following requirements.

- Controllability and identity condition: If there doesn't exist a fault in the predecessor SLICE, then the LUTs should provide the same vector from the test vector set that the LUTs in its predecessor SLICE used. For example, test vector set while testing a configuration as shown in Figure 4.8 is {1010, 0101}. If the LUTs in the S0 used the input vector 1010, and if there was no fault, then the LUTs in the S1 should also provide the input vector 1010.
- Observability condition: If there exist a fault in the predecessor SLICE, then the LUTs should detect it, and then accordingly should provide the output such that the fault is not masked.

It can be noticed that there are only two input vectors that are needed to test all of the data paths, however, each LUT has 64 memory locations. This means that, each LUT can implement the conditional functions to meet the above two requirements. In the example, as shown in Figure 4.9, the LUTs are configured to implement the comparator as follows. The LUT function can be specified as,

if (input belongs to the test vector set {0101, 1010}) then

the output of the LUT-i should be the i^{th} bit of the input vector

else

the output of the LUT-i should be a logic-1

These conditions can be satisfied using the below truth table.

Table 4.4: Truth table to implement the conditional LUT output

LUT input				**Output of each LUT**			
A[3]	**A[2]**	**A[1]**	**A[0]**	**A**	**B**	**C**	**D**
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1
0	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	1
1	**0**	**1**	**0**	**1**	**0**	**1**	**0**
1	0	1	1	1	1	1	1
1	1	0	0	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1

The idea behind these functions is to pull all of the outputs of the SLICE into all 1s state once the fault is detected. Ideally, the outputs of the alternate circuits in the SLICE should have the opposite values (alternate 1s and 0s bit pattern). At any time a fault causes the output of the SLICE to deviate from the ideal bit pattern (alternate 1s and 0s), then the LUTs in the successor SLICE detects it, and then tries to pull the output of the SLICE to all 1s state.

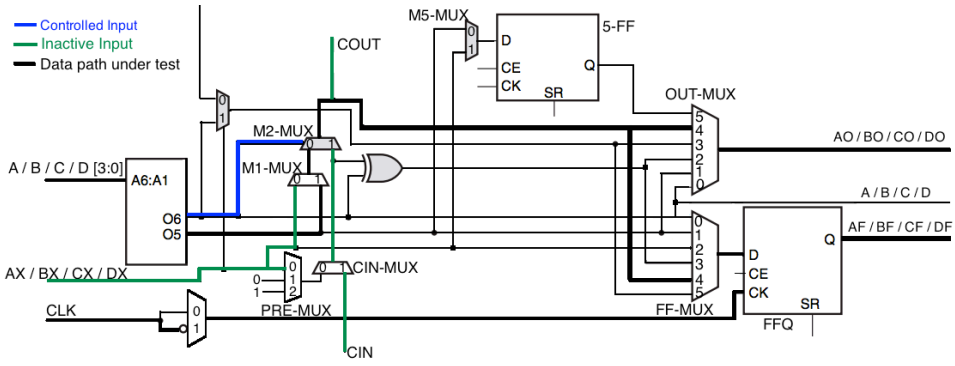


Figure 4.10: Configuration with conditional input O6. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

Another example is shown in Figure 4.10. In this configuration, the output O6 of each LUT is kept to a logic-0, and, hence, the highlighted path is selected. Once the highlighted path is selected, the output O5 of the LUT is configured to implement the functions that were discussed in the earlier configuration. The outputs O5 and O6 can be implemented as the different functions by keeping the A6 pin of the LUT to a logic-1. This can be done by connecting the A6 pin of each LUT to the local VCC that is available in the TIEOFF site adjacent to the CLB.

Consider another data path as shown in Figure 4.11. In this configuration, none of the LUT outputs is propagated to the SLICE output. However, the LUTs can still be used to implement the conditional constraints on the SLICE outputs as follows.

```
O5 = 1 (opposite value to the value selected in the PREMUX)
AX = 1 (opposite value to the value selected in the PREMUX)
if (input to the LUT is all 0s, i.e. the predecessor output is all 0s)
      O6 = 1
else
```

```
O6 = 0
```

By applying the input to the ILA as all 0s, the highlighted path should be selected, and the output of all of the SLICEs should be a logic-0. If there exists a stuck-at-1 fault (this configuration can only activate stuck-at-1 fault, because PREMUX is selected to output a logic-0, a stuck-at-1 fault is detected by another configuration) in the highlighted data path, then it will be observed in the SLICE output. This SLICE output will cause the output O6 of the LUTs in the successor SLICE to be a logic-0; therefore, the data path with the inactive input of the multiplexer is selected. This data path is set to a logic-1 because both the O5 and the AX are kept to a logic-1. As a result of this, the outputs of all successor SLICEs will be a logic-1, and, hence, the fault can be detected.

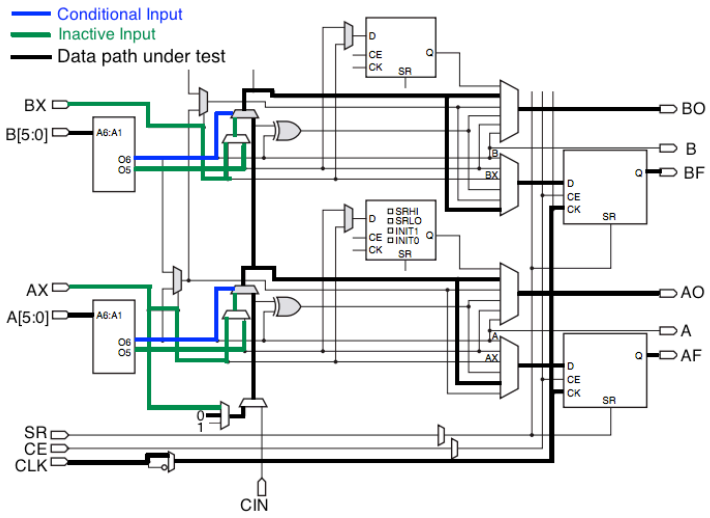


Figure 4.11: Testing of data path using conditional LUT functions. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

Different configurations have been created, each of which covers a subset of logic nodes in the SLICE, to test all of the logic nodes in the SLICE. These configurations can be summarized by the MUX settings as shown in Table 4.5. An example of a data path in Circuit A of a SLICE using these MUX settings is shown in Figure 4.12. The diagram of entire SLICE can be found in [6]. The LUT functions that are implemented in each of these configurations are discussed in Appendix A. The input vectors that are required in each of these configurations are also specified in Appendix A.

Table 4.5: Different configurations to test the data paths

#	MUX Settings								Connectivity Between Successive SLICEs/ Comments
	CLK	***OUT***	***FF***	***M1***	***M2***	***M5***	***CIN***	***PRE***	
1	0	0	0	X	X	X	X	X*	S0(AO, AF) → S1(A/B[0], C/D[0])** S0(BO, BF) → S1(A/B[1], C/D[1]) S0(CO, CF) → S1(A/B[2], C/D[2]) S0(DO, DF) → S1(A/B[3], C/D[3])
2	1	1	1	X	X	X	X	X	Same as 1
3	0	4	4	1	0	X	X	X	Same as 1
4	1	4	4	X	1	X	0	2*	Same as 1
5	0	4	4	X	1	X	0	1	Same as 1
6	1	5	2	X	X	1	X	X	S0(AO, BO, CO, DO)→ S1(AX, BX, CX, DX)
7	0	2	3	0	0	X	0	0	Same as 6
8	1	2	3	0	0	X	0	0	S0(AF, BF, CF, DF)→S1(AX, BX, CX, DX)
9	0	5	X	X	X	0	X	X	S0(AO) → S1(A/B/C/D[0]) S0(BO) → S1(A/B/C/D[1]) S0(CO) → S1(A/B/C/D[2]) S0(DO) → S1(A/B/C/D[3])
10	1	5	2	X	X	1	X	X	Same as 8
11	0	3	5	X	X	X	X	X	- Same as 1 and AX/BX/CX/DX are tied to 0 - DOUTMUX and DFFMUX are configured to select input 0 and not the one that is specified in MUX settings
12	1	3	5	X	X	X	X	X	- Same as 1 and AX/BX/CX/DX are tied to 1 - DOUTMUX and DFFMUX are configured to select input 0 and not the one that is specified in MUX settings
13	X	X	X	1	1	X	1	X	S0(COUT) → S1(CIN)

*X means MUX is configured to be OFF, any other number means the particular MUX input line of the MUX is selected.

** S0(AMUX, AQ) → S1(A/B[0], C/D[0]) can be interpreted as: The output net AO of SLICE S0 is connected to the 0th address bit of the A and B LUTs in the successive SLICE S1, whereas the output net AF of SLICE S0 is connected to the 0th address bit of the C and D LUTs in the successive SLICE S1.

*** Shared net is indicated by the symbol '/'. For example, C/D[1] means that C[1] and D[1] shares the common net.

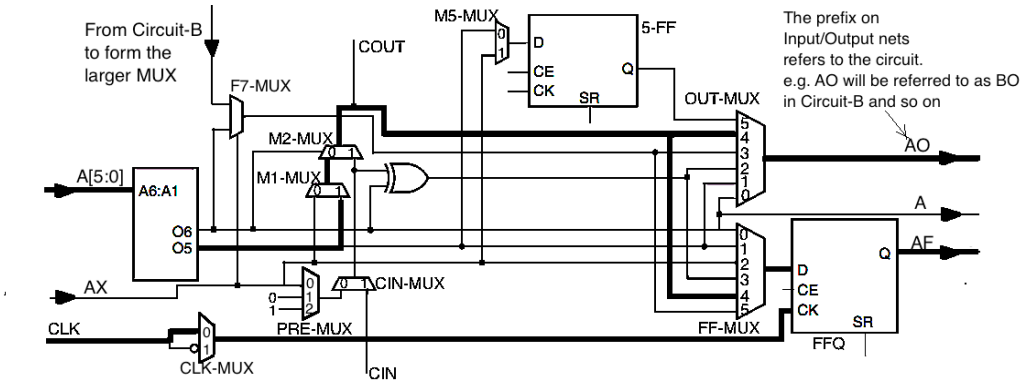


Figure 4.12: Logic circuit to demonstrate Configuration 3 of Table 4.5. Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014. Used under fair use, 2015.

4.3 MATS Test Vector Generation for RAM Testing

The RAMs are tested by multiple distributed TPGs as discussed in Chapter 3. For each column of the SLICEMs, there is an adjacent column of the SLICELs available, and this column is used to implement the instance of the TPG, which provides the inputs to the limited number of the SLICEMs.

A total of three configurations are required to test different modes of the RAM. In each configuration, all of the RAMs are configured in one of the three possible RAM modes(32x2 single port RAM, 64x1 single port RAM, 64x1 RAM with one read-write port and another read port). For testing the single port functionality of the RAM logic,

MARCH X test sequence is used, and for testing the dual port functionality of the RAM logic, the dual-port RAM test algorithm [22] is used.

The outputs of all of the RAMs in the same SLICEM are compared with each other in the adjacent SLICE, referred to as a local ORA. As all of the RAMs are configured identically, and are provided the identical test vectors, the outputs of RAMs should be identical. At any time, if the outputs of the RAM are not equal, it is registered as a logic-1 into the local ORA. The outputs of all of the local ORAs is OR together to indicate the final result of the test.

Chapter 5

Fault Isolation

The fault detection process just gives the information about if there exists a fault in the system or not, but it doesn't give any information about the location of the fault. If the fault location can be identified, then the proper steps can be taken to avoid the fault. This motivation has led to the development of fault isolation process. The modern FPGAs are so large in terms of the number of CLBs that the typical designs usually don't use 100% of the CLBs. In such scenarios, if any of the CLBs can be identified to have a fault within it, then it can be bypassed in future designs by applying location constraints. As a result of this, the FPGA can still work in the presence of a fault. Such systems are known as fault tolerant systems. Fault tolerance is an important property for the systems that are used in space, aviation, nuclear, and military applications because it is very difficult to replace the faulty FPGA in such systems.

This chapter provides the details of the fault isolation process. The initial sections give brief details about some of the hardware primitives, and the procedures that are used

to implement fault isolation. The later sections discuss about various ways to identify the fault location.

5.1 System-level Overview

The fault isolation process that is discussed in this thesis relies on the partial readback capability of the FPGA [19], which enables an outside entity to readback the configuration memory of the FPGA. The state of all of the flip-flops within the device can be analyzed by reading back the configuration memory. If a fault is detected, then the signal indicating the presence of the fault (the write signal to the DONE pin) is used to de-assert the Clock Enable (CE) signal of the TPG. Once the TPG is shutdown, no new input vectors are applied to the BUTs, and, hence, the FPGA remains in the faulty state. At this time the state of all of the flip-flops is captured into the configuration memory. The state of the flip-flops can be analyzed by reading back the configuration memory, and this data are used to determine the location of the fault in the FPGA. Figure 5.1 shows the system-level diagram of the fault isolation process.

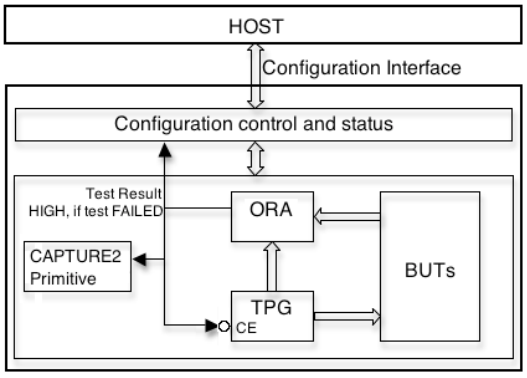


Figure 5.1: System-level diagram for fault isolation

5.2 Capturing the State of Flip-flops

Xilinx provides two ways to capture the state of the flip-flops. These two ways are briefly discussed here. The detailed information about these methods can be found in Xilinx User's Guide [19].

5.2.1 Using Configuration port Commands

The state of all of the flip-flops within the device can be readback by applying GCAPTURE command to the configuration port of the FPGA. Xilinx FPGA has a CMD register (address = 5'b0100) in their configuration port, which can be written using the configuration interface such as SelectMAP or JTAG. In order to issue the GCAPTURE command, the CMD register is written with the value 5'b01100. In order to execute the command that is written into the CMD register, the value in the FAR register (address = 5'b00001) has to be changed. Once the GCAPTURE command is executed, the state of all of the flip-flops will be captured into the configuration memory on the next configuration clock cycle. The following sequence summarizes the steps to capture the state of the flip-flops.

1) Write the GCAPTURE command into the CMD register using the configuration interface.
2) Write some dummy address into the FAR register.
3) FPGA captures the value of all of the flip-flops into the capture cells on the next configuration clock cycle.

5.2.2 CAPTUREE2 Primitive

The procedure described in the previous section can also be performed internally within the FPGA. The CFG_CENTER_MID tile of each SLR (Super Logic Region) in the FPGA contains a CAPTUREE2 hardware primitive. This primitive has two inputs - CLK and CAP. An active-high value on the CAP signal indicates that the values of all of the flip-flops have to be captured on the next positive edge of the CLK signal. The CAPTUREE2 primitive performs essentially the same operations that were described in the previous section, but it does it internally within the FPGA without using the configuration interface. The capture operation is synchronous to the CLK signal and not to the configuration clock. By default, the capture operation is performed on each positive edge of the CLK signal until the CAP is asserted. In order to restrict the capture operation to a single capture, the ONESHOT attribute of the CAPTUREE2 primitive has to be set to TRUE.

5.3 Extracting Flip-flop Values

After the flip-flop values have been sampled into the configuration memory, the configuration memory has to be read back in order to analyze the values of the flip-flops. The actual readback operation involves a specific command sequence as specified in Xilinx User's Guide [19]. Another easy way to readback the configuration memory is to use iMPACT utility provided by Xilinx. The following environment variables have to be modified in order to perform the readback operation.

```
XIL_IMPACT_VIRTEX_DUMPBIN=1
XIL_IMPACT_IGNORE_MASK_FILE=1
```

The readback data from the FPGA has the format as specified in Figure 5.2. The bit offset of each flip-flop value from the starting of the readback data has to be known in order to determine the flip-flop values. Xilinx specifies the offset of the flip-flop bits from the first bit of the configuration data. However, the readback data contains a padding frame at the beginning. In order to determine the starting point of the configuration data, the length of the padding frame has to be known. Xilinx stopped providing the length of the frame starting from 6-Series FPGAs. This information is extracted using the following procedure.

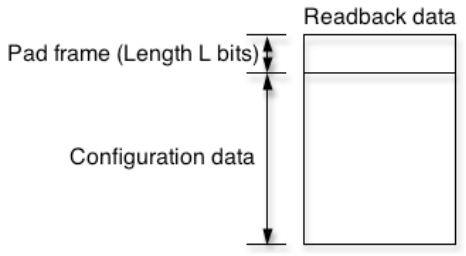


Figure 5.2: Format of readback data

The information about the bit-offsets from the first bit of the configuration data is present in the .ll file, which can be generated along with the configuration bitstream by using –l switch in bitgen utility [23]. One example line from the .ll file is shown below.

```
Bit 70997347 0x004a011f      3 Block=SLICE_X0Y0 Latch=AQ
```

This line can be interpreted as: the value of the flip-flop AQ in the SLICE_X0Y0 is present in a bit, which is at the offset of 70997347 bits from the starting of the

configuration data. This bit is part of a frame addressed by 0x004a011f and the offset of the bit in the frame is 3.

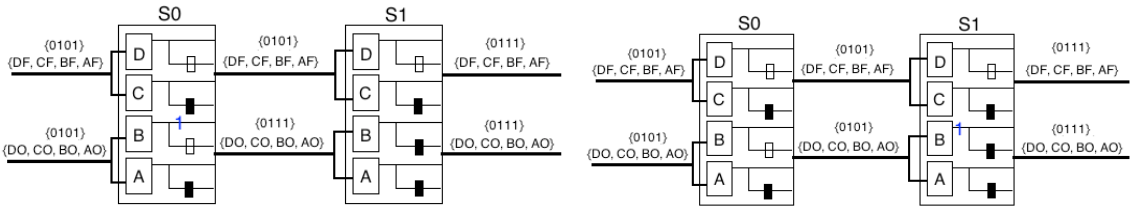
If the length of the padding frame is L, then the above bit should be at the offset of $(L + 70997347)$ bits from the first bit in the readback data. Two separate experimental configurations are created, in which one has the value of the AQ flip-flop in the SLICE_X0Y0 set to a logic-1, and in another the value of the same flip-flop is set to a logic-0. Each of these configurations is programmed into the FPGA, followed by the readback operation. The difference between both the readback data is taken. Ideally only one bit should change in both of the configurations, and, hence, the difference should indicate the location of the bit from the starting of the readback data. This bit location, which differed in both the data, was found to be 71000579^{th} bit in the readback data. This means that $L + 70997347 = 71000579$ bits, and the value of the L is determined to be 3232 bits.

5.4 Fault Isolation in ILA Architecture

All of the BIST configurations in the ILA architecture are designed such that a fault always propagates through the flip-flops. Because of the regularity present into the ILA architecture, all of the flip-flops should always form a regular pattern in the absence of a fault. For example, assume that Configuration 3 from Table 4.5 is in operation. If the input vector {1010} is applied to the ILA, then the alternate flip-flops in the ILA should have the alternate 1s and 0s bit pattern in the steady state. The term steady state refers to the condition reached after waiting long enough so that the effect of the input vector

propagates through all of the BUTs in the ILA. The other kind of regular pattern will be observed when the other input vector {0101} is applied to the ILA.

However, if there exists any fault in the ILA, then some flip-flops would deviate from the regular pattern. This irregularity in the pattern can be used to narrow down the fault location. If there exists a fault at any point and is excited by the input vector, then all of the successive flip-flops, to which the fault is propagated, will deviate from the regular pattern that their predecessor flip-flops have followed. Moreover, the order of the BUTs in the ILA is known; therefore, the first BUT in the ILA order, in which the flip-flop values deviated from the regular pattern, can be narrowed down.



(a) Fault in SLICE S0 (b) Fault in SLICE S1

Figure 5.3: Similar flip-flop values for different faults

One example is shown in Figure5.3(a). In this configuration, if the LUTs detect a fault in their inputs, then the LUTs try to pull the circuit to all 1s state (LUT function as specified in Section 4.2.5). Consider that there doesn't exist any fault in the SLICEs prior to the SLICE S0 when the input {D, C, B, A} = 0101 is applied. All of the flip-flops in the predecessor SLICEs to the SLICE S0 will have the alternate 1s and 0s bit pattern. Next, consider that there exists a stuck-at-1 fault on the BO line of the SLICE S0, then this fault will cause the LUT-A and the LUT-B in the successor SLICE S1 to drive a logic-1, and, hence, both the A-FF and the B-FF in the SLICE S1 will have a logic-1

value. However, there is another possibility that can cause the similar flip-flop signature. In the other situation, as shown in Figure 5.3(b), a stuck-at-1 fault on the output of the LUT-B in the SLICE S1 can also cause the flip-flops in the FPGA to have the similar flip-flop signature. From this example, it can be seen that the location of the fault within the FPGA can be narrowed down to the pair of two successive SLICEs, but it cannot be narrowed down to a particular SLICE. The modern FPGAs have thousands of SLICEs, and, usually not all of the SLICEs are used in the typical FPGA application; therefore, the diagnostic resolution of two SLICEs is considered practical enough in the modern FPGAs. However, if the finer diagnostic resolution is needed, then the LUTs can be configured with other functions. One such function is discussed below.

In the above example, the fault location cannot be narrowed down to a single SLICE because the fault changed the output of only a single flip-flop in the successor SLICE. However, if a single fault from the SLICE can cause two flip-flops in the successor SLICE to change their values, then the fault can be narrowed down to a single SLICE. This can be done using the following LUT functions. The outputs of the LUTs are designated as A, B, C and D. Also, the LUTs A, B, C and D are referenced by index-0, 1, 2 and 3 respectively.

D = F3 (DF, CF, BF, AF) C = F2 (DF, CF, BF, AF)

B = F1 (DO, CO, BO, AO) A = F0 (DO, CO, BO, AO)

Then the following conditions are imposed on the LUT functions.

Fi(inputs) = i^{th} bit in the input vector, if there is no error or 2-bit error in the input vector

= flip the i^{th} bit in the ideal test vector, if there is 1-bit error in the input vector

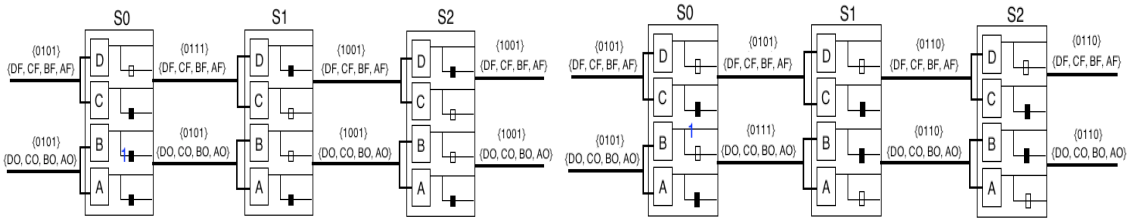
where, $0 <= i <= 3$

The functions can be explained as follows by an example. Consider that the input vector that is applied to the ILA is 1010. The complete test vector set is {0101, 1010}.

Table 5.1: Example of the LUT functions for fault isolation

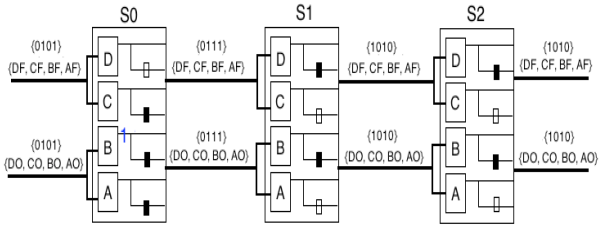
Input seen by LUT-0 and LUT-1	Number of errors	Output LUT-1	Output LUT-0	Comments
1010	0	1	0	Output of LUT-i = i^{th} bit in the input vector
0010	1	0	1	Ideal vector = 1010, Output of LUT-i = complement of i^{th} bit in the ideal vector
0000	2	0	0	Output of LUT-i = i^{th} bit in the input vector
1101	3	1	0	It will be seen as 1-bit error with respect to ideal vector = 0101 Output of LUT-i = complement of i^{th} bit in the ideal vector
0101	4	0	1	It will be seen as 0-bit error with respect to ideal vector = 0101 Output of LUT-i = i^{th} bit in the input vector

The idea here is if there exists a single bit error in any SLICE, then it should change the values of either two (if a fault exists on the branch) or four flip-flops (if a fault exists on the stem) in its successor SLICE. Once two or four bits are in error, then the faulty output will propagate to the successive SLICEs as shown in Figure 5.4. It can be seen that the flip-flop signature is different in all three cases, and, hence, the fault can be isolated to a stem or a branch in the particular SLICE.



(a) Fault on the branch BF in S0

(b) Fault in the branch BO in S0



(c) Fault on stem B in S0

Figure 5.4: Different flip-flop values for different faults

The LUT functions described above are good enough to detect all single stuck-at-fault in the FPGA. However, the problem with these functions is that it may not detect the presence of multiple simultaneous stuck-at-faults in a SLICE. One such example is shown in Figure 5.5. As a 3-bit error in the input vector is treated as a 1-bit error with respect to the other vector in the test vector set, one fault masks the effect of the other fault when it is propagated to the next SLICE. The previous LUT functions (pulling circuit to all 1s), provided the better tolerance to fault masking, i.e. it can detect the simultaneous faults that are present into a single SLICE. However, this configuration had lesser diagnostic resolution as compared to the later configuration. Whereas, the later configuration has better diagnostic resolution, but it has lesser tolerance to the fault masking. The reason behind the loss in the diagnostic resolution or the loss in the fault detection is that the eight inputs to a SLICE are transformed to only four independent outputs by the LUTs in the SLICE.

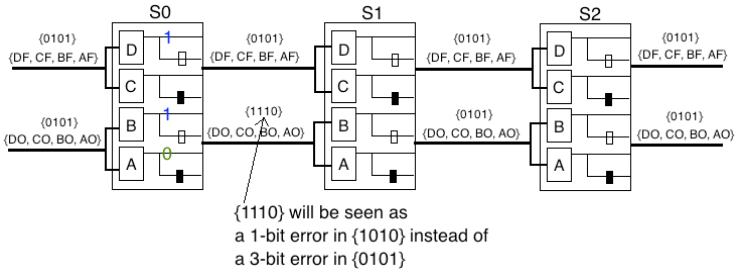
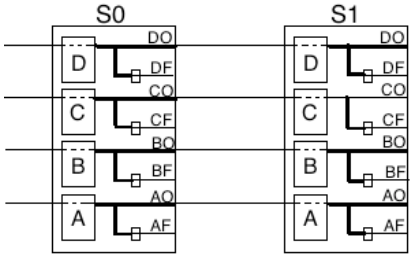


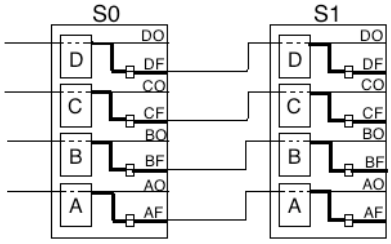
Figure 5.5: Fault masking when propagating eight outputs to successor SLICE

The configurations that propagate only four outputs (number of outputs equal to number of LUTs) to its successor SLICE have better tolerance to the fault masking and have better fault diagnostic resolution. In this case, as there are only four LUTs in the circuit, and there are four outputs propagated from the predecessor SLICE, each LUT can be configured as the pass through. This way, the entire BIST configuration can be thought of as four parallel scan chains as shown in Figure 5.6. Moreover, as all four circuits are tested independently, the fault can be isolated to a particular circuit within the SLICE. However, as discussed in Chapter 4, by propagating only a subset of the SLICE outputs at a time, two separate BIST configurations are needed to test the paths through {DO, CO, BO, AO} and the paths through {DF, CF, BF, AF} separately as shown in Figure 5.6. As the number of BIST configurations increases, the time for the fault detection also increases. This configuration, which has better fault diagnosis resolution and better tolerance to the fault masking, requires more time to complete the testing. In order to achieve the best results, the combination of two approaches can be used. For the fault detection, the configurations, which test multiple data paths simultaneously, can be

used. Only if a fault is detected, but cannot be isolated, then the configurations, which propagate only a subset of the outputs to the successor SLICE, can be used.



(a) Testing of data paths through {DO, CO, BO, AO}



(b) Testing of data paths through {DF, CF, BF, AF}

Figure 5.6: Testing of a single data path at a time

5.5 Fault Isolation in SelectRAM

While testing the SelectRAM, the outputs of two adjacent RAMs are compared in the adjacent SLICE, referred to as a local ORA. The output of the local ORA is also registered into the flip-flop, and the registered output is provided back to the local ORA as shown in Figure 5.7. The register that stores the output of the local ORA is initialized to a logic-0. At any time, if the outputs of two RAMs are not equal, then it will be latched as a logic-1 in the local ORA register. The location of the fault can be narrowed down to a SLICE by reading back all of the local ORA registers, and by finding out the local ORAs with the registered value as a logic-1.

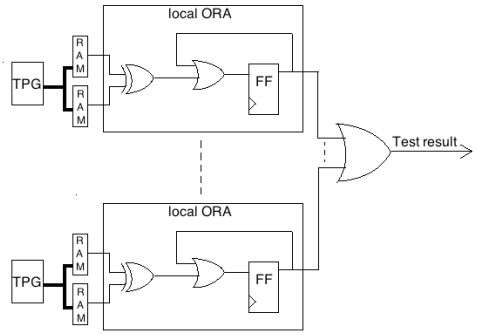


Figure 5.7: Registering a fault in local ORA during RAM testing

5.6 Fault Isolation in Shift-register

The shift registers are initialized with the alternate 1s and 0s bit pattern at the beginning of the test. If there exists any stuck-at fault in the shift register, then it will result in all of the successive bits after the fault location to have the same value. The fault in the shift register can be narrowed down by reading back the shift register content, and then by finding the location, from where the deviation in the bit pattern occurs.

Chapter 6

Implementation and Results

Test generation for BIST configurations is one of the difficulties that is usually encountered in the development of FPGA BIST [8]. This is because there is no way to specify some high level description to FPGA CAD tools to generate the BIST configurations. The CAD tools usually optimize the logic, and, hence, the redundant logic for testing, such as keeping the inactive input of the multiplexer to a logic-1 in order to test the gate-level model of multiplexer (Section 4.2.1), is trimmed by the CAD tool. Moreover, the CAD tools have the choice of implementing the specified logic in different ways in the CLB, and, hence, users don't have the tight control over selecting the desired data path in the CLB. This chapter discusses about the scripts that are developed for the generation of BIST configurations. The later sections in the chapter discuss about the fault coverage and the sources of overhead that are associated with the FPGA BIST.

6.1 Scripts for Test Generation

The generation of the BIST configurations is automated using a script. This script takes the FPGA part name and the rectangular coordinates of the test area as the input parameters, and generates all of the BIST configurations to test all of the CLBs within the specified test area. Two alternative approaches are developed to generate the BIST configurations. In the first phase of the project, the script generated HDL code using Xilinx HDL primitives, and then the code is compiled by Xilinx tool chain. The intermediate files in the compilation process are modified in order to meet the required conditions of the BIST configurations. Another approach was designed later to bypass the HDL synthesis, MAP, and placement operations involved during the generation of the configuration bitstream. The placed design is generated directly using XDL (Xilinx Design Language) [24], and it is further modified to generate the final configuration bitstream. The next sections provide the overview of both of these approaches.

6.1.1 Test Generation using HDL

In this approach, the script generates the HDL code for the design, which is then synthesized, mapped, placed and routed. Once the placed and routed design (NCD file) is available, then the design is converted into the XDL file, which is a human readable equivalent of the NCD file. Various scripts are run on the XDL file to make necessary changes in the CLB configurations, and then the modified XDL file is converted back to the NCD file. This NCD file is converted into the configuration bitstream using Xilinx bitgen tool. Figure 6.1 summarizes all of the steps involved in the generation of the bitstreams.

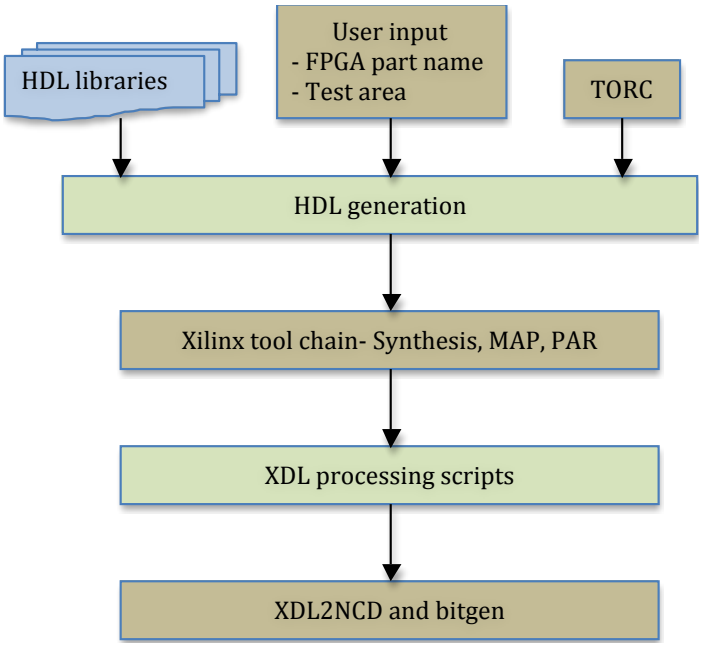


Figure 6.1: Test generation using HDL

Components of the HDL

The HDL code contains the following components.

1. Controller.v: This file implements the BIST controller logic. The controller drives the necessary signals to the DUT, observes the response from the DUT, and then accordingly indicates the PASS/FAIL status of the test into the DONE bit of the status register.
2. Cells: Various cells, such as slicel, srl32, srl16, s_ram32, and s_ram64, are created. Each of these cells exercises the different logic nodes and the functionalities of the CLB, and all of them together cover all of the logic nodes and the functionalities of the CLB.
3. DUT: These files, beginning with "testgen_" prefix, implements the top-level device-under-test (DUT). This design instantiates the appropriate cells and

specifies the connectivity between them. This file is generated using a C++ program, which relies on TORC APIs. TORC [27] is an open source CAD tool for Xilinx FPGAs. This tool provides the necessary APIs to extract the tile map of the FPGA. Once the tile map is extracted, then the instantiated cells are associated with the location constraints [26] that direct the placer to place the cell to the site specified by the constraint. By associating various constraints to the cells, the template of placed and routed design is generated, which can further be modified in the later stages using XDL.

XDL processing scripts

Using the above HDL files and the Xilinx tool chain, an initial XDL file is first generated. Various scripts are then run to modify it. A brief summary of each script is given below.

1. extract_dut.sh: This script extracts the various components of the design, such as an instance of the controller, instances of the DUT, nets of the controller, nets of the DUT, clock nets, and so on, from the XDL file.
2. swap_outpin.sh <config>: Once the instances and the nets of the DUTs are extracted, then this script modifies the data paths within the SLICEs. For example, in the initial XDL file, the O5 input in the AOUTMUX is enabled. However, to test the O6 path through the AOUTMUX, the setting of the AOUTMUX has to be modified. This script can be used to switch the O5 path to the O6 path in the SLICE. This script takes the Id of the BIST configuration as the input, and makes necessary changes in the configuration of the SLICEs for the specified BIST configuration.

3. combine_xdl.sh: Once the extracted XDL parts are modified, then this script merges back all of the modified XDL parts, and generates a new XDL file.
4. config_lut.sh <config>: This file is used to modify the LUT content in the DUT SLICEs. While generating the HDL file, a dummy LUT content is specified in the HDL so that the Xilinx tool chain doesn't optimize the logic. Later (post routing), the LUT content is changed in the XDL file using this script. This script takes the Id of the BIST configuration bitstream, and then programs the LUT equations that are derived for the specified BIST configuration.
5. compile.sh <config>: This script is a top-level script, which compiles the HDL files using standard Xilinx tool chain and applies the required scripts to generate the final bitstream file. For example, if "compile.sh 1" command is executed in the lut directory, then the configuration-1 of the LUT configurations is generated into the lut directory.

The final NCD file, which is converted to bitstream, was opened in the FPGA Editor [28], and it was verified that the NCD file is generated correctly.

6.1.2 Test Generation using XDL

The primary motivation behind the test generation using the XDL is to bypass the synthesis, map and place operations. The typical BIST configuration uses all of the SLICEs in the FPGA. If the FPGA size is larger, then the synthesis of the design alone takes hours. Sometimes, even 32 GB of the system memory is not sufficient for the Xilinx Synthesizer, and it crashes arbitrarily. In order to mitigate all these issues, the different approach was required to generate the BIST configurations.

The XDL based approach directly generates the placed design by bypassing the synthesis, map, and placement phases of the design. Figure 6.2 shows the various steps involved in the test generation using XDL. First, the XDL template libraries for the BIST controller and the DUT are created. The libraries for the BIST controller include the XDL templates for the TPG, the ORA, the CAPTURE2 primitive, and the STARTUPE2 primitive. The DUT libraries include the XDL templates for the SLICEL and the SLICEM.

In order to generate the XDL template libraries, a HDL design is created to test a small area within the FPGA. As only the small area of the FPGA is instantiated, the design can be compiled quickly using the Xilinx tool chain. The intermediate XDL file generated during the compilation process contains the XDL instances of various components such as the TPG, the ORA, and other primitives. These components are extracted and stored as the XDL template library.

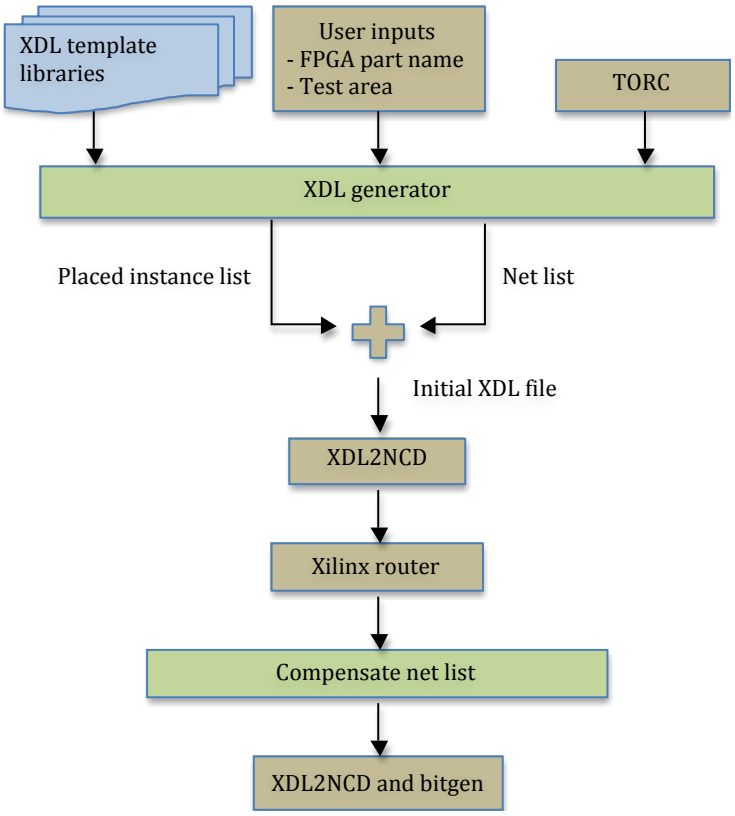


Figure 6.2: Test generation using XDL

XDL generation

The generation of the BIST configurations is automated using a script, which takes the FPGA part name and rectangular coordinates of the test area as the input parameters. A C++ program is created, which uses the TORC APIs to determine the tile map of the design. The locations of various components of the BIST controller and the SLICEs within the specified test area are determined using the extracted tile map, and the XDL template libraries for various components are modified to create the placed instance list of all of the components. The program also generates the net list, which specifies the connectivity between all of the components. The instance list and the net list are put

together to create an initial XDL file. This XDL file is converted to the NCD file, and then the Xilinx router is invoked to route the net list that was specified in the XDL file.

Compensate net list

The Xilinx router swaps the pins of the LUTs in the net list to simplify the routing as shown in Figure 6.3. While using the HDL flow, "LOCK_PINS" constraint [26] is associated with the instantiated cells, and, hence, the Xilinx tool chain doesn't modify the nets during the entire flow. However, if the direct XDL is generated, then specifying such constraint is not possible. As a result of this, all of the nets that were specified in the initial XDL file are disturbed. The routed NCD file is converted into the XDL file to read back the modified net list. The modified net list is traced using a C++ program, and the LUT pins that were swapped during the routing are determined. Once the information about the swapped pins is extracted, then the program modifies the LUT equations to compensate for the swapped LUT pins as shown in Figure 6.3. After the LUT equations are modified, the modified XDL is converted back to the NCD file, and is converted into the configuration bitstream using Xilinx bitgen tool.

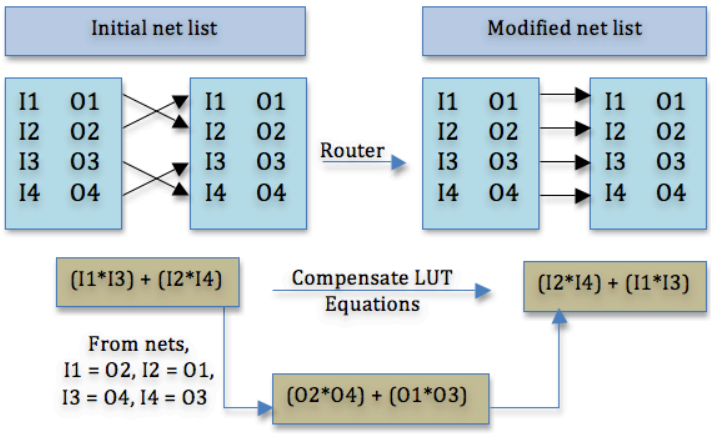


Figure 6.3: Example of pin swapping

6.2 Fault Coverage

A total of 30 BIST configurations have been created in order to test the logic resources in any of the 7-Series FPGAs. The FPGA that was used for the experiment is Xilinx Zync XC7Z020 FPGA. These 30 configurations are grouped according to the functionality that they cover in the CLB. Out of these 30 configurations, 12 configurations test the LUTs, 13 configurations test the data paths, 3 configurations test the SelectRAM logic, and 2 configurations test the shift register logic. The faults in the FPGA can be emulated using configuration memory bit fault injection. During the generation of the configuration bitstream, the intermediate files can intentionally be modified in order to emulate some faults in the FPGA. For example, a particular memory location in the LUT can be forced to a logic-0 to emulate a stuck-at 0 fault in the particular memory location of the LUT. The necessary conditions to cover the internal faults in the logic nodes (LUTs and Multiplexers) are discussed while discussing about the BIST configurations to test the particular logic nodes. By applying the input vectors that are specified in Appendix A, it can be verified that this conditions are also satisfied.

A gate-level model of all components in the CLB is considered, and fault coverage was calculated. The graph in Figure 6.4 shows the fault coverage in the CLB by the BIST configurations. The left Y-axis shows the number of single stuck-at-faults faults in each CLB covered by these configurations. The right Y-axis shows the percentage fault coverage for the single stuck-at faults.

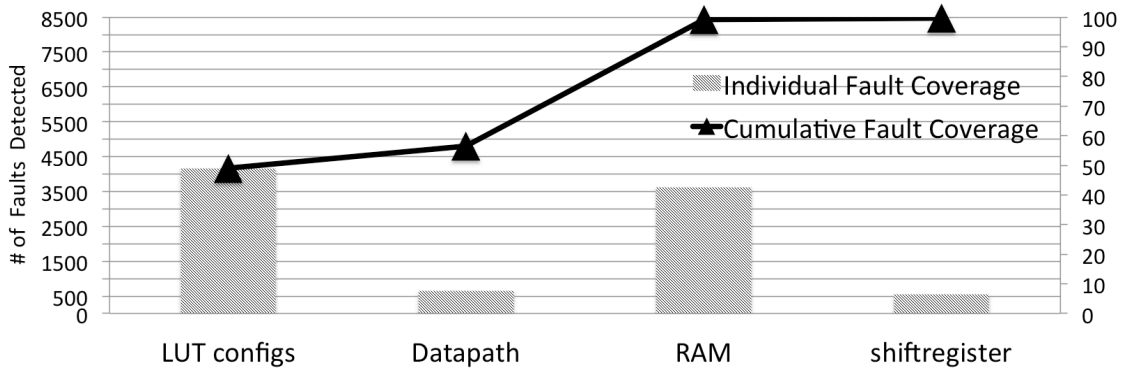


Figure 6.4: Fault coverage in a CLB by BIST configuration group

6.3 Sources of Overhead

The primary sources of overhead associated with the offline BIST are the test time and the external memory required to store the BIST configurations. During the test, the application on the FPGA cannot run, and, hence, it is desired that the test should take minimum possible time. The graph in Figure 6.5 shows the breakdown of the time involved in different operations of the test for a BIST configuration. The bitstreams were downloaded using xc3sprog [29], which is an open source software tool to download bitstreams on FPGAs. The status register was readback using Xilinx iMPACT utility. Figure 6.5 shows that downloading the BIST configuration to the FPGA takes the majority of the test time, and running the test takes only a small fraction of the total test time; therefore, the number of test vectors and the test frequency is not of primary concern, but reducing the download time of the BIST configurations to the FPGA has more impact on the test time. Download time is directly proportional to the size of the BIST configurations and the speed of the configuration interface. In order to reduce the test time, the total size of the BIST configurations should be reduced.

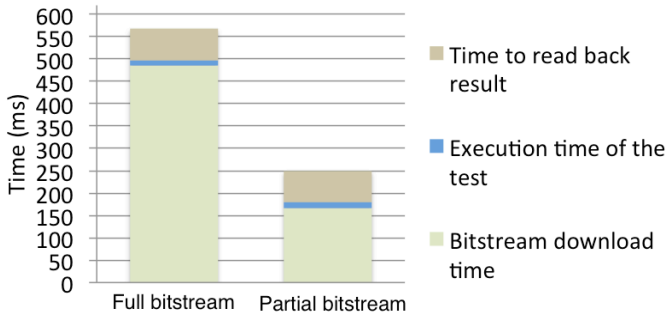


Figure 6.5: Time involved in different operation while testing an FPGA using a BIST configuration (configuration interface used is JTAG)

The total size of the BIST configurations can be reduced by reducing the number of BIST configurations and the average size of the BIST configurations. Partial reconfiguration of the FPGA is used in order to reduce the average size of the BIST configurations. The majority of the frames in a BIST configuration contain the routing information. If the routing is kept constant across multiple configurations, then only one configuration has to be stored as full bitstream, and other bitstreams can be stored as partial bitstreams. For example, the first five configurations in Table 4.5, has similar routing. The size of a BIST configuration for Xilinx XC7Z020 device is 3.9 MB. If all of the five configurations are stored as full bitstreams, then the total size of the five BIST configurations will be 18 MB. However, by using the partial reconfiguration, the total size of all of the five configurations is reduced to 5.85 MB. Out of the 30 configurations, only 12 configurations have unique routing, and only those configurations need to be stored as full bitstreams.

If all of the CLBs in the FPGA cannot be tested simultaneously, then the FPGA has to be tested in multiple test sessions, each of which tests a subset of the CLBs at a

time. As a result of this, the total test time increases by the factor of the number of test sessions. For example, the BIST architecture that uses the helper cells to form an ILA (discussed in Section 2.2.2) can test only half of the CLBs in a single test session. In this case, a minimum of two test sessions are required to test the entire FPGA. The BIST configurations that are discussed in this thesis can test all of the CLBs simultaneously, and, hence, it is possible to test all of the CLBs in a single test session. Sometimes, the routing congestion puts the upper bound on the number of CLBs that can be routed in a single test session, as highlighted in [3][21]. If all of the CLBs couldn't be routed in a single configuration, then the FPGA has to be divided into partitions, and then each partition has to be tested separately in a different test session. By increasing the usage of the local routing resources, and reducing the average fan-out in the design, the routing complexity is reduced in all of the BIST configurations that are discussed in this thesis; therefore, the probability of successfully routing all of the CLBs in a single configuration increases. As a result of this, the total number of test sessions reduces, which results in reduction in the total test time. It was determined that all of the BIST configurations could be routed on different Xilinx devices of Zync family, such as the XC7Z010, XC7Z020 (the device in which these experiments were confirmed upon), XC7Z030, and XC7Z045, without creating the partitions.

The external memory requirement is dependent on the size of the BIST configurations. By reducing the average size of the BIST configurations and the total number of BIST configurations, the total size of the BIST configurations can be reduced, and, hence, the external memory required to store the BIST configurations also reduces.

Chapter 7

Conclusion and Future Scope

This work provided the complete test to detect and isolate a single stuck-at-fault in the CLBs of the Xilinx 7-Seireis FPGAs. This test is suitable for any level of testing starting from the manufacturing test to the in-system testing. All of the test administration tasks can be performed through the configuration interface of the FPGA, and the need for user I/O pins to perform the test is eliminated. As a result of this, the test can be immediately deployed on any in-use systems without making any system-level changes.

An Iterative Logic Array (ILA) based BIST architecture enables the testing of all of the CLBs in the device in a single test session. By finishing the testing in a single test session, the total test time to perform the test and the external memory required to store the BIST configurations are reduced. Also, the BIST architecture discussed in this thesis reduces the routing complexity of the BIST configurations, and, hence, the architecture is scalable to the different sizes of the FPGA devices. The ILA architecture minimizes the other sources of overheads such as the resources used by TPGs and ORAs, and the number of I/O pins of the circuit under test.

The necessary conditions to test various logic nodes within the FPGA at gate-level are discussed in detail, and different BIST configurations are implemented to satisfy those conditions. The test also provides the capability to isolate a single stuck-at-fault in the FPGA device. Fault isolation is the primary step towards building the fault tolerant FPGA systems. The use of partial reconfiguration capability of the FPGA is used in order to minimize the test time and the external memory required to store the BIST configurations.

7.1 Future Scope

The area of FPGA BIST has many opportunities for the future developments. This work was developed for Xilinx FPGAs because a lot of the resources and the information regarding the Xilinx FPGAs are available in public. If the similar information is provided for other FPGA vendors, such as Altera, then the similar kind of test can also be created for other FPGAs.

The majority of the FPGA BISTs are performed offline. This means that the application on the FPGA cannot run during the testing process. In future, the online test can be created, which detects the unused sites in the FPGA during the operation of the FPGA, and tests those sites while the application is still running. Once the unused sites are tested, then the application can dynamically be migrated over these tested sites, and the testing of the previously used sites can be carried out. A BIST architecture that can dynamically be reshaped has to be developed to build this kind of test environment.

Bibliography

[1] Jameel Hussein and Gary Swift, "Mitigating Single-Event Upsets (white paper)", WP395 (v1.1), May 19, 2015

[2] Berg M., Poivey C., Petrick D., Espinosa D., Lesea A., LaBel K.A., Friendlich M., Kim H. , Phan A., "Effectiveness of Internal Versus External SEU Scrubbing Mitigation Strategies in a Xilinx FPGA: Design, Test, and Analysis," in IEEE transactions on Nuclear Science, vol. 55, issue 4

[3] Stroud C., Lee E., Konala S., Abramovici M., "Using ILA testing for BIST in FPGAs," in proceedings of International Test Conference, 1996

[4] Premchandran R. Monon, Arthur D. Friedman, "Fault Detection in Iterative Logic Arrays," in IEEE transactions on computers, vol. c-20, no. 5, May 1971

[5] Arthur D. Friedman, "A Functional Approach to Efficient Fault Detection in Iterative Logic Arrays," IEEE transactions on computers, vol. 43, no. 12, December 1994

[6] Xilinx Inc., "7 Series FPGAs Configurable Logic Block User Guide, UG474 (v1.7)," November 17, 2014

[7] Xilinx Inc., "Partial Reconfiguration User Guide, UG702 (v14.1)" April 24, 2012

[8] Stroud C., Konala S., Ping Chen, Abramovici M., "Built-in self-test of logic blocks in FPGAs (Finally, a free lunch: BIST without overhead!)," in proceedings of VLSI Test Symposium, 1996.

[9] S. Dhingra, D. Milton, and C. Stroud, "BIST for logic and memory resources in Virtex-4 FPGAs," Proc. IEEE North Atlantic Test Workshop, pp. 19-27, 2006.

[10] S. Dhingra, S. Garimella, A. Newalker, and C. Stroud, "Built-in self-test of Virtex and Spartan II FPGAs using partial reconfiguration," Proc. IEEE North Atlantic Test Workshop, pp. 7-14, 2005.

[11] Stroud C. E., Leach K. N., Slaughter T. A., "Bist for Xilinx 4000 and spartan series fpgas: a case study," in proceedings of International Test Conference, 2003.

[12] Dutton B. F., Stroud C. E., "Built-In Self-Test of configurable logic blocks in Virtex-5 FPGAs," in proceedings of 41st southeastern symposium on System Theory, 2009.

[13] E. J. McCluskey, "Verification Testing A Pseudoexhaustive Test Technique," in IEEE transactions of Computers, vol. 33, issue 6, June 1984, p.p. 541-546

[14] Huang W. K., Meyer F.J., Park N., Lombardi F., "Testing memory modules in SRAM-based configurable FPGAs," in proceedings of International Workshop on Memory Technology, Design and Testing,1997

[15] Renovell M., Portal J.M., Figueras, J., Zorian, Y., "RAM-based FPGAs: a test approach for the configurable logic," in proceedings of Design, Automation and Test in Europe, 1998.

[16] Huang W. K.,Zhang M.Y., Meyer, F.J., Lombardi F., "A XOR-Tree Based Technique for Constant Testability of Configurable FPGAs," in proceedings of Sixth Asian Test Symposium, 1997.

[17] Jordan C., Marnane, W. P., "Incoming inspection of FPGA's," in proceedings of European Test Conference, 1993.

[18] Huang W K., Lombardi F., "An approach for testing programmable/configurable field programmable gate arrays," in proceedings of 14th VLSI Test Symposium, 1996.

[19] Xilinx Inc., "7 Series FPGAs Configuration, User Guide, UG470 (v1.9)," November 14, 2014

[20] Michelle Fernandez, Peggy Abusaidi, "Virtex-6 FPGA Routing Optimization Design Techniques (white paper)," WP381 (v1.0), October 28, 2010

[21] Stroud. C., Ping Chen, Konala. S., Abramovici. M., "Evaluation of FPGA Resources for Built-In Self-Test of Programmable Logic Blocks," in proceedings of ACM Fourth International Symposium on Field-Programmable Gate Arrays, 1996.

[22] L-T Wang, C. Stroud, and N. Touba, System-on-Chip Test Architectures, Morgan Kaufmann, 2007.

[23] Xilinx Inc., "Command Line Tools User Guide, UG628 (v 14.1)," April 24, 2012

[24] Beckhoff C., Koch D., Torresen, J., "The Xilinx Design Language (XDL): Tutorial and use cases," in 6th international Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011

[25] Xilinx Inc. "Xilinx 7-Series FPGA and Zynq Libraries Guide for HDL Designs, UG768 (v 14.2)," July 25, 2012

[26] Xilinx Inc. "Constraints Guide, UG625 (v. 13.4)," January 18, 2012

[27] TORC, available at http://torc-isi.sourceforge.net/

[28] FPGA Editor User Guide, available at

http://www.xilinx.com/support/sw_manuals/2_1i/download/fpedit.pdf

[29] xc3sprog, available at http://xc3sprog.sourceforge.net/

Appendix A

Details of Configurations

LUT functions for the configurations in Table 4.5 are discussed here. Using the following functions, the truth table for the LUT can be derived. Based on the truth table, the LUT can be configured with Boolean equation.

Configuration 1

LUT functions:

- For all LUTs, O5 = 1

- LUT-A:
  if (A[3:0] == 1010 || A[3:0] == 0101) then
  O6 = bit[0] of input vector
  else
  O6 = 1

- LUT-B:
  if (B[3:0] == 1010 || B[3:0] == 0101) then
  O6 = bit[1] of input vector
  else
  O6 = 1

- LUT-C:

if (C[3:0] == 1010 || C[3:0] == 0101) then
O6 = bit[2] of input vector
else
O6 = 1

- LUT-D:
if (D[3:0] == 1010 || D[3:0] == 0101) then
O6 = bit[3] of input vector
else
O6 = 1

Input vector set:
A/B[3:0] = {1010, 0101}
C/D[3:0] = {1010, 0101}

Configuration 2

LUT functions:

- For all LUTs, O6 = 1

- LUT-A:
if(A[3:0] == 1010 || A[3:0] == 0101) then
O5 = bit[0] of input vector
else
O5 = 1

- LUT-B:
if(B[3:0] == 1010 || B[3:0] == 0101) then
O5 = bit[1] of input vector
else
O5 = 1

- LUT-C:
if(C[3:0] == 1010 || C[3:0] == 0101) then
O5 = bit[2] of input vector
else
O5 = 1

- LUT-D:
if(D[3:0] == 1010 || D[3:0] == 0101) then
O5 = bit[3] of input vector
else
O5 = 1

Input vector set:

A/B[3:0] = {1010, 0101}
C/D[3:0] = {1010, 0101}

Configuration 3

LUT functions:

- For all LUTs, O6 = 0
- For all LUTs, O5 is same as Configuration 2

Input vector set:

A/B[3:0] = {1010, 0101}
C/D[3:0] = {1010, 0101}

Configuration 4

LUT functions:

- For all LUTs, O5 = 0
- For all LUTs, if ADDRESS[3:0] == 4'b1111 then O6 = 1, else O6 = 0

Input vector set:

A/B[3:0] = 4'b1111
C/D[3:0] = 4'b1111

Configuration 5

LUT functions:

- For all LUTs, O5 = 1
- For all LUTs, If ADDRESS[3:0] == 4'b0000 then O6 = 1, else O6 = 0

Input vector set:

A/B[3:0] = 4'b0000
C/D[3:0] = 4'b0000

Configuration 6

LUT functions:

- For all LUTs, O6 = 1, O5 = 1

Input vector set:

{DX, CX, BX, AX} = {1010, 0101}

Configuration 7

LUT functions:

- For all LUTs, O6 = 0, O5 = 1

Input vector set:
{DX, CX, BX, AX} = {1010, 0101}

Configuration 8

LUT functions:

- For all LUTs, O6 = 0, O5 = 1

Input vector set:
{DX, CX, BX, AX} = {1010, 0101}

Configuration 9

LUT functions:

- LUT-A: O6 = 1, O5 = A[0]
- LUT-B: O6 = 1, O5 = B[1]
- LUT-C: O6 = 1, O5 = C[2]
- LUT-D: O6 = 1, O5 = D[3]

Input vector set:
A/B/C/D[3:0] = {1010, 0101}

Configuration 10

LUT functions:

- For all LUTs, O6 = 1, O5 = 1

Input vector set:
{DX, CX, BX, AX} = {1010, 0101}

Configuration 11

LUT functions:

- For all LUTs, O5 = 1

- LUT-A:
  if (A[3:0] == 4'b0100 || A[3:0] == 4'b1011) then
  O6 = bit[0] of input vector
  else
  O6 = 0

- LUT-B:
  if(A[3:0] == 4'b0100 || A[3:0] == 4'b1011) then
  O6 = complement of bit[0] of input vector
  else
  O6 = 0

- LUT-C:
  if(A[3:0] == 4'b0100 || A[3:0] == 4'b1011) then
  O6 = complement of bit[0] of input vector
  else
  O6 = 0

- LUT-D:
  if(A[3:0] == 4'b0100 || A[3:0] == 4'b1011) then
  O6 = bit[0] of input vector
  else
  O6 = 0

Input vector set:
A/B[3:0] = {0100, 1011}
C/D[3:0] = {0100, 1011}

Configuration 12

LUT functions:

- For all LUTs, O5 = 1

- LUT-A:
  if(A[3:0] == 4'b1110 || A[3:0] == 4'b0001) then
  O6 = bit[3] of input vector

else
O6 = 0

- LUT-B:
  if(A[3:0] == 4'b1110 || A[3:0] == 4'b0001) then
  O6 = complement of bit[3] of input vector
  else
  O6 = 0

- LUT-C:
  if(A[3:0] == 4'b1110 || A[3:0] == 4'b0001) then
  O6 = complement of bit[3] of input vector
  else
  O6 = 0

- LUT-D:
  if(A[3:0] == 4'b1110 || A[3:0] == 4'b0001) then
  O6 = bit[3] of input vector
  else
  O6 = 0

Input vector set:
A/B[3:0] = {1110, 0001}
C/D[3:0] = {1110, 0001}

Configuration 13

LUT functions:

- For all LUTs, O6 = 1, O5 = 1

Input vector set:
{CIN} = {1, 0}