Institute of Computer Engineering and Computer Architecture
Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich
Pfaffenwaldring 47, 70569 Stuttgart

Master Project Nr. 3161

Evaluation of Advanced Techniques for Structural FPGA Self-Test

Mohamed ABDELFATTAH

M S C   T H E S I S

in partial fulfillment of the requirements
for the degree of **Master of Science**

Supervisors :	Dipl.-Inf. Michael IMHOF
	Dipl.-Inf. Michael KOCHTE
	Dipl.-Inform. Claus BRAUN
Examiner :	Prof. Dr. rer. nat. habil. Hans-Joachim WUNDERLICH
Start Date :	March 01, 2011
Submission Date :	August 31, 2011
CR Classification :	B.5.2, B.6.1, B.8.1, B.8.2, C.4
Study Program :	M.Sc. Information Technology (INFOTECH)

To my mom and dad

Abstract

This thesis presents a comprehensive test generation framework for FPGA logic elements and interconnects. It is based on and extends the current state-of-the-art. The purpose of FPGA testing in this work is to achieve reliable reconfiguration for a FPGA-based runtime reconfigurable system. A pre-configuration test is performed on a portion of the FPGA before it is reconfigured as part of the system to ensure that the FPGA fabric is fault-free. The implementation platform is the Xilinx Virtex-5 FPGA family.

Existing literature in FPGA testing is evaluated and reviewed thoroughly. The various approaches are compared against one another qualitatively and the approach most suitable to the target platform is chosen. The array testing method is employed in testing the FPGA logic for its low hardware overhead and optimal test time. All tests are additionally pipelined to reduce test application time and use a high test clock frequency. A hybrid fault model including both structural and functional faults is assumed.

An algorithm for the optimization of the number of required FPGA test configurations is developed and implemented in Java using a pseudo-random set-covering heuristic. Optimal solutions are obtained for Virtex-5 logic slices. The algorithm effort is parameterizable with the number of loop iterations each of which take approximately one second for a Virtex-5 sliceL circuit.

A flexible test architecture for interconnects is developed. Arbitrary wire types can be tested in the same test configuration with no hardware overhead. Furthermore, a routing algorithm is integrated with the test template generation to select the wires under test and route them appropriately.

Nine test configurations are required to achieve full test coverage for the FPGA logic. For interconnect testing, a local router-based on depth-first graph traversal is implemented in Java as the basis for creating systematic interconnect test templates. Pent wire testing is additionally implemented as a proof of concept. The test clock frequency for all tests exceeds 170 MHz and the hardware overhead is always lower than seven CLBs. All implemented tests are parameterizable such that they can be applied to any portion of the FPGA regardless of size or position.

Contents

List of Figures

List of Tables

List of Abbreviations

ATE	Automatic Test Equipment
BIST	Built-In Self Test
BUF	Buffer
CAD	Computer-Aided Design
CE	Clock Enable
CF	Coupling Fault
CF_{dyn}	Dynamic Coupling Fault
CF_{id}	Idempotent Coupling Fault
CF_{in}	Inversion Coupling Fault
CFM	Cell Fault Model
CIM	Configurable Interface Module
CLB	Configurable Logic Block
CLK	Clock
CUT	Circuit Under Test
DFS	Depth First Search
DRC	Design Rule Check
DRF	Data Retention Fault
FF	Flip-Flop
FF	Functional Fault
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HW	Hardware
ILA	Iterative logic arrays
IOB	Input/Output Block
IP	Intellectual Property

JTAG	Joint Test Action Group
LF	Linked Fault
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
MUX	Multiplexer
NCD	Netlist Circuit Description
NGD	Native Generic Database
ORA	Output Response Analyzer
PAR	Place and Route
PLB	Programmable Logic Block
PR	Partial Reconfiguration
PSM	Programmable Switch Matrix
RAM	Random Access Memory
RST	Reset
SA-0	Stuck-At Zero
SA-1	Stuck-At One
SAF	Stuck-At Fault
SCF	State Coupling Fault
SF	Structural Fault
SOF	Stuck-Open Faults
SR	Shift Register
SRAM	Static Random-Access Memory
TC	Test Configuration
TF	Transition Fault
TPG	Test Pattern Generator
WUT	Wire Under Test
XDL	Xilinx Design Language

Chapter 1

Introduction

Contents

1.1 Motivation and Objectives

To speed up particular applications, algorithm-specific hardware (HW) accelerators are being used alongside a general purpose processor. These HW accelerators are tailored for a specific algorithm, therefore, many of them are required for complex systems to enhance their performance. However, this comes at the high price of the additional silicon. Recently, field programmable gate arrays (FPGA) are being used for implementing reconfigurable architectures in which a fixed FPGA area can be reprogrammed at runtime to change the circuit function; thereby implementing multiple HW accelerators without additional area requirements.

The reconfiguration process is done through a runtime system implemented either on-chip or on an external processor core [1]. The runtime system is also responsible for ensuring a reliable reconfiguration process and dynamic adaptability to avoid using defective blocks in the FPGA fabric. This establishes fault tolerance of the dynamic, in-field adaptation to the application by reconfiguration. The hardware overhead is reduced compared to classical fault tolerance schemes such as those which involve structural redundancy.

The proposed methodology involves the pre-configuration test (PRET) of the existing un-programmed FPGA configurable logic blocks (CLB), memory, crossbar switches, etc. If the target fabric is fault free, the reconfiguration process is executed, followed by a post reconfiguration test (PORT). PORT is a functional test focusing on delay faults and correct module integration, not covered by PRET. All the test structures in the target area are removed after running a PRET test so that their area is usable by application logic later. Note that PRET, PORT and the actual reconfiguration process are defined and executed on one part of the FPGA fabric or "container" at a time.

Fig. 1.1 shows an FPGA with a fixed processor core and a reconfigurable container. The processor contains a runtime system which has access to the FPGA partial reconfiguration (PR) port and is able to dynamically reconfigure portions of the FPGA online. Before configuring the container into a HW accelerator, PRET is performed to ensure structural integrity of the container under consideration.

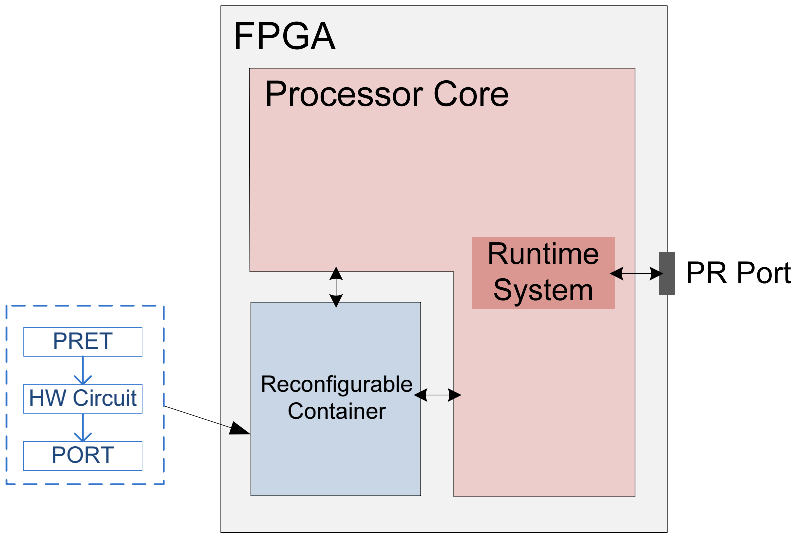

Figure 1.1: A runtime reconfigurable system implemented on an FPGA

1.2 Reliability Threat

The need for testing arises from the vulnerability of electronic devices to fault occurrence. The transistor feature size gets smaller in every new technology node and the manufacturing process is becoming more complex resulting in silicon variations within and between dies. During the lifetime of an electronic device, reliability decreases and behavior may differ from the intended one [2]. These aging effects are also critical. In addition, transient faults could occur as a result of particle strikes, and environmental factors such as fluctuations in temperature or power supply are all threats to the reliability of FPGAs.

FPGAs are advancing as an implementation platform for digital circuit implementation because of their increased capacities and improved computer-aided design (CAD) tools [3]. They are also finding applications in safety-critical reconfigurable systems which drives the need to create a fault-tolerant platform for implementation. Online test is used in this thesis to create this fault-tolerant reconfigurable FPGA system by validating the FPGA fabric before a module reconfiguration is performed.

1.3 Thesis Organization

After the introduction, the necessary background information is briefly stated in Chapter 2. This is followed by an extensive literature review of state-of-the-art

FPGA testing methods in Chapter 3. Chapter 4 explains the fault models adopted in testing the various FPGA components.

Chapters 5 and 6 are dedicated to presenting the test concepts used in this work. The concepts are based on the current state-of-the-art of the field and have been extended where required. Implementation details and results are combined in Chapter 7. Finally, the thesis is concluded with some brief notes about possible future work in the field.

Chapter 2

Background

Contents

2.1 FPGA Overview

The reconfigurability of FPGAs is a result of its re-programmable architecture. This section introduces the required prerequisite information to guide the rest of this work. A general view of FPGAs is given with explanation of the various building blocks. The Xilinx Virtex-5 FPGA architecture is also considered specifically as it is the implementation platform used in this thesis. Finally, a short introduction on built-in self test (BIST) for FPGAs is given with some basic definitions.

2.1.1 FPGA Architecture

Fig. 2.1 shows a simplified circuit schematic of an FPGA. The main components in an FPGA are the configurable logic blocks (CLB). These programmable units implement the logic of a digital circuit. Each CLB communicates with another through the interconnect network that consists of programmable switch matrices (PSM) and interconnect wires. Finally the FPGA communicates with other logic components through input/output blocks (IOBs). This makes it possible for the FPGA to implement arbitrary digital logic circuits.

The presented schematic (Fig. 2.1) shows that each CLB consists of two logic slices. Although this is true for Virtex-5 FPGAs, it is not the general rule. This is

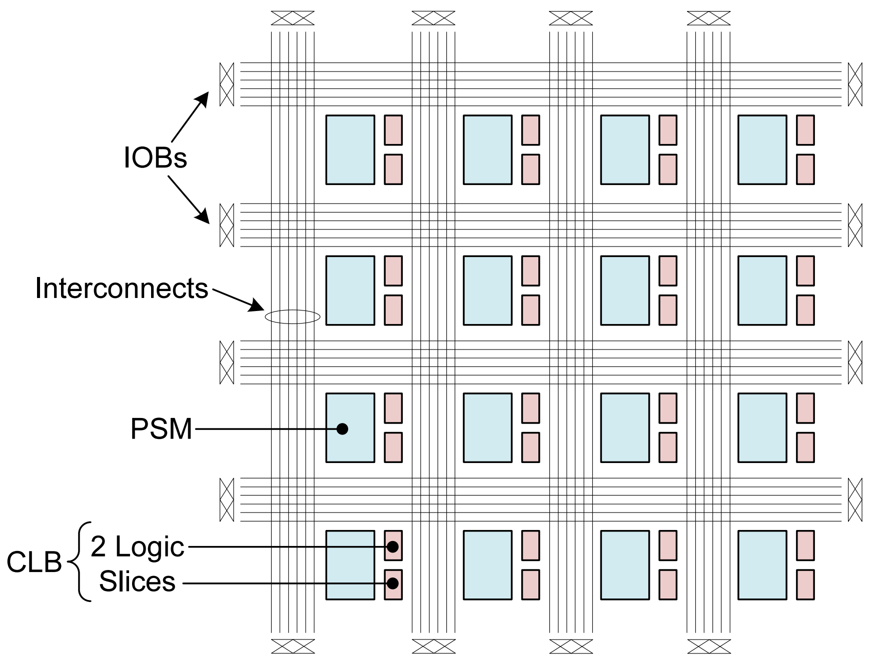

Figure 2.1: FPGA schematic diagram showing basic building blocks

just a partitioning of the CLB such that signal routing and other parameters are optimized [3].

SRAM-based FPGAs are reconfigured by rewriting its SRAM configuration cells. This process is done using one/multiple scan chains going through all the programmable components [3]. The following subsection explains this while presenting each of the FPGAs subcomponents.

2.1.2 Configurable Logic Blocks

CLBs consist of three main subcomponents: Multiplexers, lookup tables (LUT) and sequential elements such as flip-flops. Each is presented in this subsection separately then combined to illustrate an entire CLB.

2.1.2.1 Multiplexers

Multiplexers are used to specify the connection of signals to one another inside the CLB. Fig. 2.2 shows a four-input multiplexer with the two select inputs tied to SRAM configuration cells. This means that the multiplexer inputs are specified when downloading the configuration and stays the same when a circuit is active on the FPGA. An n-input multiplexer requires $log_2(n)$ SRAM configuration inputs.

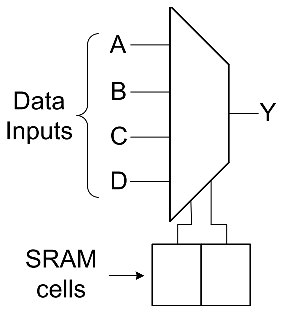

Figure 2.2: A four-input FPGA multiplexer

2.1.2.2 Lookup Tables

Depending on its number of inputs, an LUT implements any combinational logic function. This is also done through SRAM configuration cells that store the truth table values for the logic function. A multiplexer selects the appropriate truth table value depending on the input combination.

Fig. 2.3 demonstrates a typical two-input LUT. The configuration SRAM cells are connected to the data inputs of a multiplexer of which the select inputs act as the function inputs. In this way, any two input logic function is implemented [3].

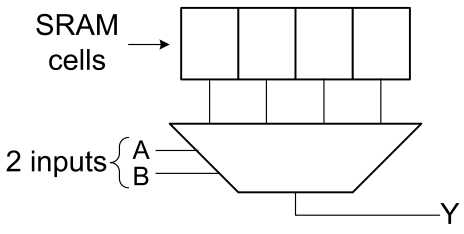

Figure 2.3: A two-input LUT

Similarly, any n-input function can be implemented using a similar circuit with 2^n SRAM cells and a 2^n-input multiplexer.

2.1.2.3 Sequential Elements

Sequential elements are essential for any digital logic design. This dictates that they must be present on the FPGA. They are usually preceded by a multiplexer so that any signal from the logic portion of a slice can be routed through. Newer FPGAs have sequential elements that can be configured into either a flip-flop or a latch to implement both edge and level sensitive designs.

2.1.3 Switch Matrix and Interconnect

The interconnect topology is becoming a critical factor in new FPGAs. They account for approximately 80% of the configuration SRAM cells, indicating their importance [4]. The purpose is to connect CLBs to each other and be flexible so that any point in the FPGA circuitry can be connected to any other point.

Routing is carried out by programmable switches that route the signals in their correct path, switch connections on or off, and buffer the interconnect wires.

Fig. 2.4 shows three kinds of programmable interconnect resources found in the FPGA [3]. The multiplexer has already been introduced in context of intra-CLB routing, but it is also an essential component in routing the global interconnects found on the FPGA. It is clear that it picks which signal to drive the output depending on its configuration. Fig. 2.4 also demonstrates a programmable pass-transistor that can make or break connections. In addition a tri-state buffer is also shown.

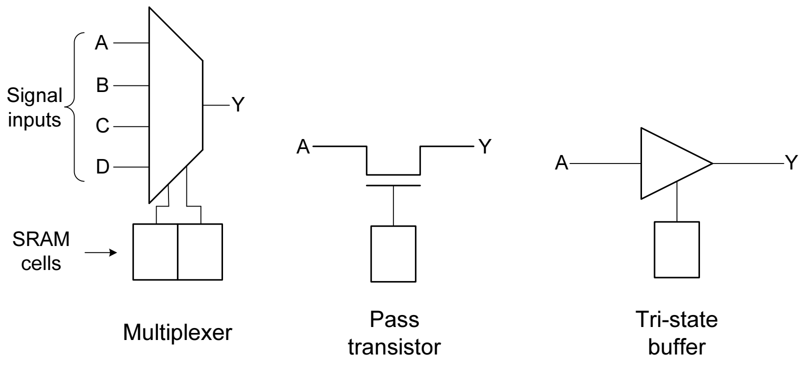

Figure 2.4: Three types of FPGA programmable switches

Xilinx FPGAs use *island-style* interconnects. This means that CLBs are surrounded by fixed interconnect wires [3]. Between the CLB input/output pins, and the wires are programmable switches. Altogether, these are grouped into a so-called programmable switch matrix (PSM).

The PSM is able to make connections between the various pins attached to it so that it connects CLB pins to interconnects. The programmable connections inside the PSM are called programmable interconnect points (PIP). PIPs are implemented using combinations of programmable switches such as the circuits shown in Fig. 2.4.

Fig. 2.5 illustrates the *island-style* interconnect architecture. Four CLBs are shown as well as four PSMs. A possible PIP implementation is also shown. This variant can make any connection between the four wires attached to it using five pass transistors. Each pass transistor is controlled using an SRAM configuration cell.

2.2 Xilinx Virtex-5 FPGA

The implementation platform of this work is the Xilinx Virtex-5 FPGA [5]. This FPGA is capable of many advanced features such as partial reconfiguration [6] and memory readback [7, 8, 9]. It also contains many advanced components such as

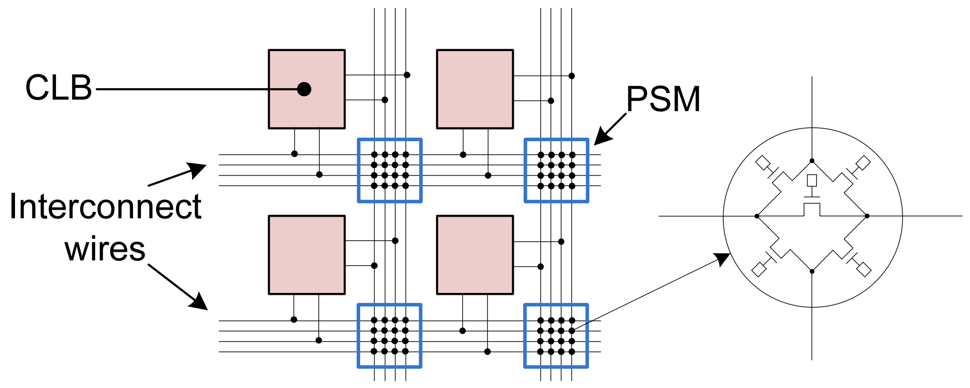

Figure 2.5: Island style FPGA interconnects and possible implementation of a PIP

digital signal processing slices and block random-access memory (RAM). This work considers the CLB logic and interconnects. This section is dedicated to present the Virtex-5 FPGA architecture and configuration.

2.2.1 CLB Architecture

The logic components introduced in the previous section are combined together to form a logic slice. The Virtex-5 CLB consists of two logic slices: sliceL and sliceM [5]. Both are connected to a single PSM as shown earlier in Fig. 2.1.

Fig. 2.6 shows sliceL. It consists of a circuit repeated four times. This circuit consists of a 6-input LUT connected to multiplexers and finally a sequential element (configured as either a flip-flop or latch). A chain of multiplexers and XOR gates runs through the middle of the slice to perform fast carry computations.

Fig. 2.7 depicts sliceM, which contains more functionality than the sliceL. In addition to LUT functionality, sliceM LUTs can be configured into RAM or shift register (SR). This is done using the storage elements present within each LUT.

2.2.2 Programmable Routing Resources

Virtex-5 routing is organized in an *island-style* architecture [5]. Neither the details of the PSM nor the interconnect wires are given in the documentation because of its complexity. However, from the details provided from Xilinx computer-aided design (CAD) tools, many of the interconnect details are inferred.

2.2.2.1 Wire Classification

There are five main interconnect types: Global, long, pent, double and bounceacross. They differ in length, buffering, number of connections and number of hops. Table 2.1 summarizes their essential properties.

Global and long lines are bidirectional and can broadcast signals to multiple CLBs depending on the configuration. Pent, double and bounceacross wires are unidirectional. Pent and double lines span five and two CLBs respectively. This

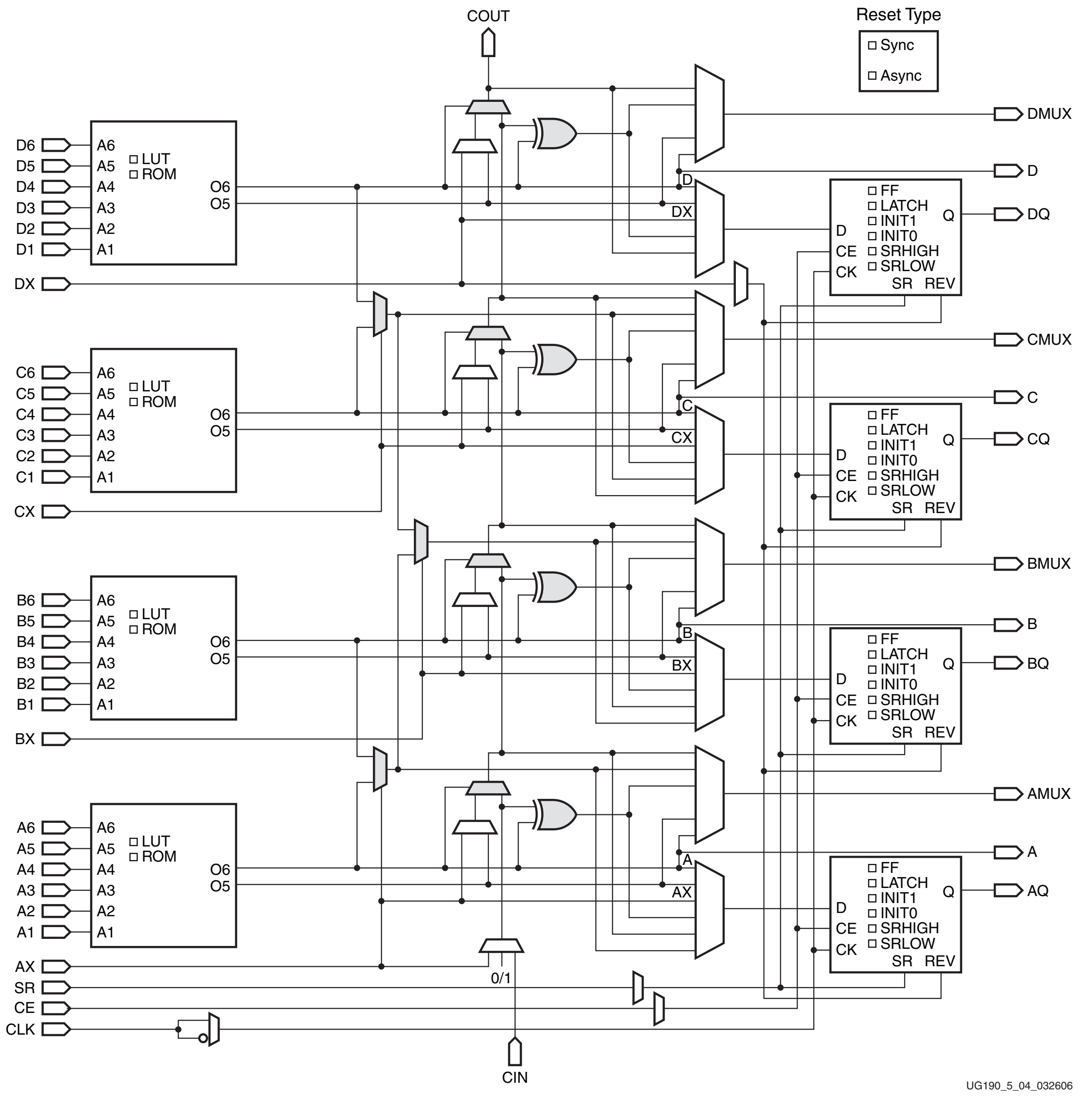

Figure 2.6: Virtex-5 sliceL diagram (from [5])

Wire Type	Length (CLBs)	# Connections	# Hops
Global	20	20	1
Long	24	4	6
Pent	5	2	2,5
Double	2	2	1,2
Bounceacross	1	1	1

Table 2.1: Virtex-5 wire properties

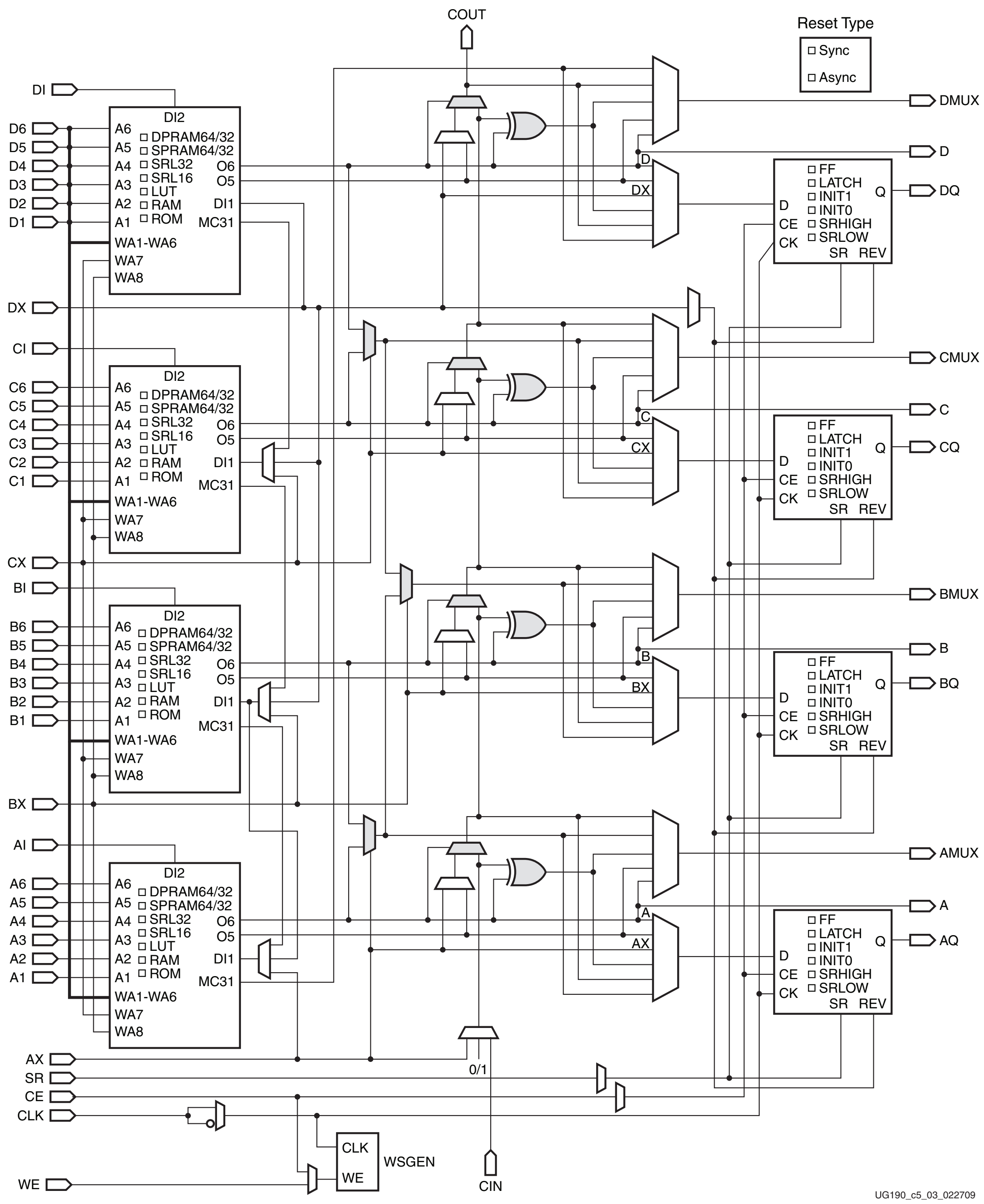

Figure 2.7: Virtex-5 sliceM diagram (from [5])

distance is the Manhattan distance from source to sink and they can be in any direction. There is additionally an intermediate middle connection of distance 2 and 1 for pent and double wires respectively. A connection can either be established from the beginning (BEG) terminal to this middle (MID) connection or to the final (END) connection.

Appendix B illustrates the wire types and some connection possibilities for each classification. It is clear that each wire type can connect in any of the four directions (north/south/east/west). In addition, double and pent wires can make diagonal connections as long as the Manhattan distance abides to their classification. Appendix B describes the Xilinx naming conventions used in naming interconnect pins.

2.3 Built-In Self Test

Semiconductor testing can either be controlled on-chip or through external test machines. It is necessary to use on-chip testing for applications that require in-field testing because it would not be possible to connect large external test machines in that case. This test scheme is called built-in self test (BIST).

To test a digital circuit, test patterns are applied at the circuit inputs and the responses are observed. The test response is analyzed and compared to the expected output to indicate whether the circuit failed to produce the correct result or if the test was passed.

This section covers some basic definitions about BIST but omits the details and specifics of FPGA testing. This is explained later in detail in the "state of the art" chapter as well as chapters 5 and 6.

2.3.1 FPGA Testing

The different FPGA components were presented earlier in this chapter. BIST is employed to test these different components. Fig. 2.8 illustrates a basic test setup. It consists of a test pattern generator (TPG) and an output response analyzer (ORA). These components provide the test vectors and examines the test results to indicate test status (passed/failed).

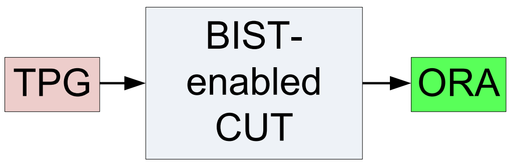

Figure 2.8: BIST setup

Fig. 2.8 states that the circuit under test (CUT) must be BIST enabled. This means that there must be test infrastructure inside the circuit to facilitate test vector application and ensure observability of faults at circuit outputs.

Due to FPGA reconfigurability, it is possible to reconfigure a CUT into a BIST enabled one by reprogramming the fabric. It is crucial to introduce the term "test configuration" (TC) in this context. A test configuration is an FPGA setup that ensures that the targeted CUT is BIST-enabled and includes the configuration for the corresponding TPG and ORA.

CLB components were presented earlier in this chapter. Each subcomponent requires a different TPG, ORA and FPGA configuration. This dictates the use of multiple TCs for CLBs as well as interconnects. Each TC guarantees coverage of a subset of the faults by targeting only one or two subcomponents each time. The targeted subcomponents are configured into BIST-enabled CUTs and a valid TPG and ORA are configured for testing the CUT. The complete set of TCs is designed such that full-coverage of CLB faults is achieved after all TCs are executed.

The number of TCs is the main parameter for optimization of FPGA testing because it determines test speed. FPGA configuration time is approximately 1000 times slower than test application time.

2.3.2 Test Terminology

The relevant terminology and definitions typically used in the field of testing are listed below. These terms will be used in the following sections.

- **Defect:** Distortion of the material shape in a chip.
- **Fault:** Abstraction of defects at logic level.
- **Error:** Incorrect circuit state during computation.
- **Online Test:** A test that is performed in-field without interrupting normal circuit operation.
- **Fault Coverage:** Portion of detected faults out of the total number of assumed faults.
- **Test Vector/Pattern:** Bit-vector that exposes potential faults while testing a logic circuit.
- **Test pattern Generator (TPG):** Circuit that generates test vectors for a CUT.
- **Output Response Analyzer (ORA):** Circuit that analyzes test response and indicates whether a fault is detected from the running test.
- **Test Configuration (TC):** An FPGA setup that ensures that the targeted CUT is BIST-enabled and includes the configuration for the corresponding TPG and ORA.
- **C-testability:** A C-testable array of logic circuits is one that can be tested using a fixed number of test patterns and test configurations irrespective of array length.

Chapter 3

State of the Art

Contents

The subject of FPGA test has been rigorously researched in the past decade. FPGA test is primarily divided into two parts: CLB testing and interconnect testing. In this chapter, literature representing the current state-of-the-art will be reviewed and briefly compared.

3.1 CLB Test Approaches

The logic portion of the FPGA consists of memory elements, multiplexers and some logic gates. These components are packed in the CLBs which are repeated in an array through the FPGA structure as discussed in chapter 2.

There are three main approaches to testing FPGA logic components found in the literature. Either by using conventional logic testing combined with the use of test response compaction [10, 11, 12] or by using concepts from testing iterative logic arrays (ILA) [13, 14, 15, 16, 17, 18, 19, 20, 21] or by using advanced memory read back methods for response analysis [12, 22]. The three approaches are restated below:

– **Approach 1:** Conventional CLB test with test response compaction.

– **Approach 2:** Using iterative logic arrays.

– **Approach 3:** Using memory read back methods for response analysis.

Optimization of logic testing aims at reducing the number of required test configurations and the required BIST hardware overhead. For these reasons, testing with ILAs has been most popular thus far. The third approach is relatively new since it is based on memory readback which has only been available for newer FPGAs. The first approach is the simplest one but it requires the most BIST infrastructure as well as the longest test time.

3.1.1 CLB Test with Response Compaction

The first methods for testing FPGAs are simple. The basic idea is to configure one row (or column) of the FPGA as the circuits under test (CUT) and the next row (or column) as the BIST infrastructure. This hardware infrastructure is composed of response compactors [10, 11] or test pattern generators (TPG) and output response analyzers (ORA) [12].

Response compaction could be in the form of AND and OR trees designed to compact a response consisting of all 1's or all 0's respectively [10]. This approach is advantageous in detecting multiple faults but requires at least three configurations for each test type. This allows the rows (or columns), previously configured as compaction trees, to be tested. To optimize the AND/OR trees, the authors in [10] propose compaction using so-called "majority gates". These 3-input gates act as binary AND or OR gates depending on the control signal on its third input thereby reducing the number of test configurations for architectures with LUTs having three or more inputs.

Instead of using separate response compaction methods for the "1" output and the "0" output from the CLB, a parity tree is used in [11] for response compaction. As the name suggests, the XOR tree computes the parity of the signals input to it. It will therefore flip the output for any odd number of bit flips input to it [11]. This approach has lower hardware overhead and less test time when compared to [10] due to the simpler compaction method. This testing scheme is illustrated in Fig. 3.1; two CLB rows are shown in which the first contains the CUTs and the second contains the compaction tree, in this case a parity tree. The final output is observable through an IOB.

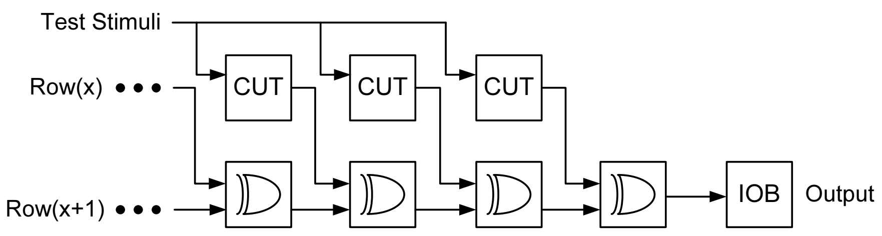

Figure 3.1: Testing scheme using parity tree for test response compaction

The approach in [12] does not use response compaction but relies on the use of automatic test equipment (ATE) and is suitable for offline test only. Similarly to

[11, 10] it requires two test phases to complete one test type on the CLBs. This is to alternate between the CUTs and the BIST hardware on the chip. In this case, half the FPGA is configured as TPGs and ORAs to test the other half: the CUTs. TPGs are simple counters and ORAs compare two identical CLBs under test and stores the response in a flip-flop. The boundary scan test access port is then used to readback and analyze the results [12].

3.1.2 Array-based CLB Test

In the previous section, the term “CLB test” is used loosely with no details of the actual test performed on each CUT. The literature that will now be introduced, however, goes into the details of sub-CLB component test and coverage according to the single stuck-at fault model. The CUTs are then connected in an array.

Because they all follow the same idea, the publications [13, 14, 15, 16, 17] are discussed collectively in this section. CLBs are divided into three main subcomponents, each of which can be separately exhaustively tested [14, 13, 16]. These components are the LUTs, the multiplexers and the sequential elements (flip-flops or latches). The following test methodology follows the “divide and conquer” approach in testing FPGAs, the component tests are therefore introduced each under a separate title.

3.1.2.1 Multiplexer

Multiplexers are used extensively in FPGAs to route signals to their appropriate terminals. The multiplexer select inputs are tied to SRAM configuration memory cells and can only be changed by reconfiguring the FPGA [13, 14, 16]. It is important to distinguish between configuration inputs (such as the select inputs of a multiplexer) and the operation inputs (such as the actual multiplexer inputs) since the former determines the number of configurations, whereas the latter specifies the number of test patterns. As previously mentioned and now restated for emphasis, FPGA test time is measured by the number of required reconfigurations, that is, the number of patterns on the configuration inputs.

An exhaustive test guaranteeing detection of all single stuck-at faults and ensuring proper function without knowledge of the implemented multiplexer structure is achieved by applying the exhaustive test set to the configuration inputs and observing the output for both 0 and 1 input patterns [13, 14, 16]. That means that for a multiplexer of n select inputs, 2^n configurations are required each with only two test patterns. This is shown in Fig. 3.2 in which $2^2 = 4$ different configurations are required because there are two select inputs.

3.1.2.2 Lookup Table: Function Mode

LUTs are the main building blocks of CLBs. As explained in chapter 2, LUTs contain sequential elements and a large multiplexer to store and select the function values respectively. From this viewpoint; the same test methods for the multiplexer

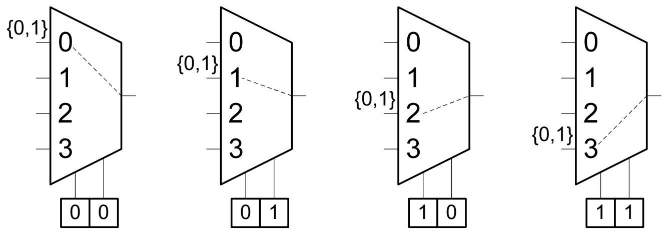

Figure 3.2: Multiplexer testing configurations

can be used for the LUTs [15, 13, 14, 16]. The difference is that the multiplexer select inputs are the operation inputs whereas the data inputs are tied to configuration cells specifying the LUT function. This is made clear in Fig. 3.3.

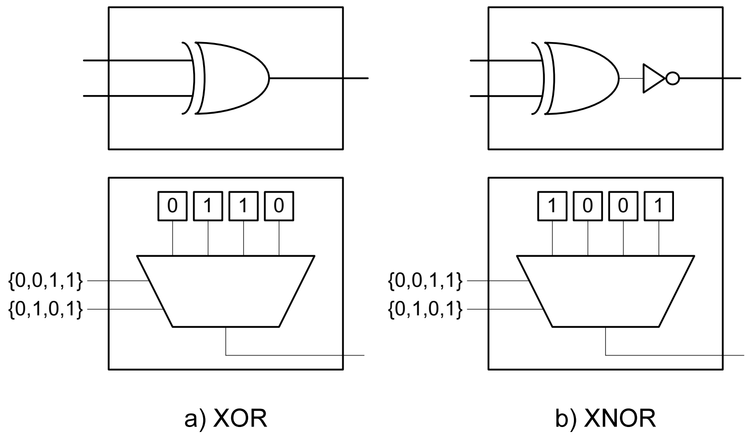

Figure 3.3: LUT testing configurations

2^n configurations are required for a n-select multiplexer with only two test patterns necessary. For a n-input LUT the opposite is true: only two test configurations are required with 2^n test patterns [15, 13, 14, 16]. The two configurations must exercise both the “0” and “1” values which may be placed in the SRAM configuration cells. These configuration bits also determine the logic function of the LUT so the authors use the XOR and $XNOR$ configurations for two reasons [15, 13, 14, 16]. The first reason is that these configurations test for all stuck-at faults (0 and 1) since they are the inverse of one another. The second reason is that XOR/XNOR gates have no controlling value; if a single fault occurs at their input, it always inverts the output. This paves the way for connecting them in a C-testable array capable of testing for single faults. The two configurations described are shown in Fig. 3.3. The first configuration can be repeated once more to test additionally for transition faults in the SRAM configuration cells [16, 15]. This makes a total of three test configurations for the LUTs in function mode.

The mentioned publications then state that the LUTs should be connected together in a C-testable array which guarantees propagation of a single fault and suggest the reduction of an FPGA from a two dimensional array into a one dimensional array of testable ILAs as shown in Fig. 3.4.

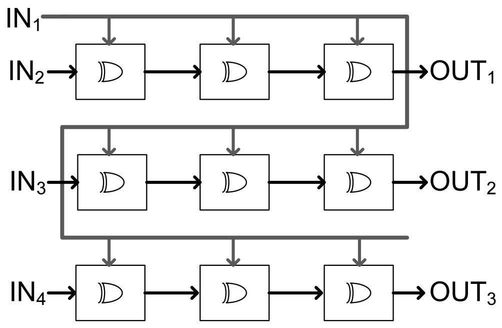

Figure 3.4: Three one-dimensional arrays of three CLBs

Although it is proven using boolean logic expressions that the ILAs repeat their logic function output every second CUT [16], it still remains to provide a formal proof for the C-testability of the XOR arrays. In addition, the description of the arrays in [15, 13, 16] is not very clear. These shortcomings are remedied within this thesis.

3.1.2.3 Lookup Table: RAM Mode and Flip-Flops

As discussed earlier, advanced LUT functions include RAM mode. In this configuration, the LUT acts as a random-access memory of size 2^n for a n-input LUT. Testing RAM modules is a very well-researched subject and mature algorithms exist for it such as the march tests. Only one test configuration is required to test the LUTs in this mode using one of the mentioned tests [15, 13]. The authors choose to implement the MATS++ algorithm with a small modification: the output of the RAM is registered with the slice flip-flop. This adaptation is called the *shifted* MARCH++ algorithm and allows for simultaneous testing of the flip-flops.

RAM modules can be configured in an array, called the *pseudo shift register* [13, 15]. This is done by connecting the output of the flip-flop to the data input of the next RAM module as shown in Fig. 3.5. For an array of size m it is shown that it takes $2m$ clock cycles for each address per test element to be tested [13, 15]. The MATS++ has three test elements meaning that the total test time adds up to $6m \times 2^n$ clock cycles (where n if the number of address bits).

Another approach handles the flip-flop test separately by configuring them in a scan chain [23]. This test is additionally adaptive, able to detect and diagnose the position of multiple faults. When a faulty flip-flop is detected, the chain is reconfigured starting from the next fault-free flip-flop. The number of configurations can therefore be any number between 1 and N (N being the length of the flip-flop chain) [23]. This test is advantageous for its multiple-fault detection and diagnosis capabilities. It is devised in the context of full coverage manufacturing testing.

Only in [21] are the various enable and set/reset control signals mentioned for

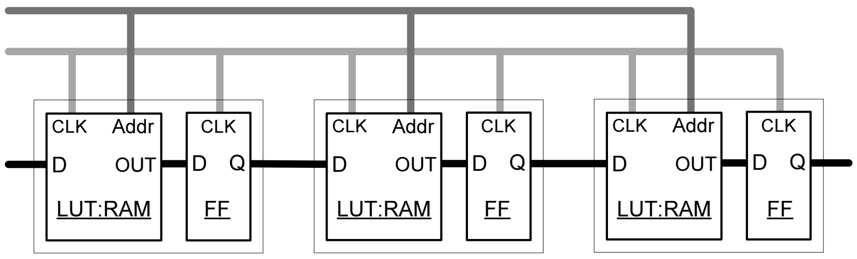

Figure 3.5: The *pseudo shift register*

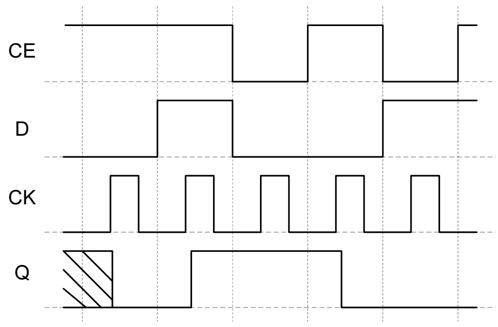

Figure 3.6: Testing the clock enable

the flip-flops. In order to perform an exhaustive functional test for the flip-flops, they are connected in an array and the different modes are used with sufficient input stimuli to expose any functional faults [21]. For instance, Fig. 3.6 shows five clock cycles which are necessary to test all possible transitions which would functionally test the clock enable (CE) input [21]. Taking the Xilinx XC4000 FPGA as an example, it is highlighted that the flip-flops must be tested with all the following considerations:

– Testing the "input and hold" functions (flip-flop storage behavior).

– Rising- and falling-edge triggered flip-flops.

– Set/reset input and functionality.

– Set/reset enable and disable.

– Clock enable function

Tests should be overlapped where possible to reduce configurations [21], but there are no results on the number of configurations achieved by the authors for flip-flop testing separately.

3.1.2.4 Other Array Test Methods

CLB inputs are always greater in number than their outputs. To overcome this problem in array testing, while assuring full observability of errors, *helper* CLBs are used to generate the missing outputs for the next cell in an array [18]. This also means that the *helper* CLBs require a separate test session in which they become the CUTs. Compared to previous methodologies presented in this section, this test requires double the number of configurations and therefore double the test time. In fact, a third test configuration is also necessary to test the FPGA area used by the TPG and ORAs [18]. One TPG is used to feed the test stimuli and one ORA is used to compare the output of each pair of arrays [18]. This means that a large number of IOBs ($N/4$ IOBs for N rows) are still required to observe the response.

The concepts of ILA testing [24] are utilized in [21] to derive test configurations for LUTs. To test a logic array, such as Fig. 3.7, the logic functions of blocks f, g and h must be constrained such that $h(g(f(v))) = v$. That means that the input test pattern v repeats after the array period, which is three in this example. The test pattern v must be chosen to satisfy this property, furthermore; the functions f, g and h are constrained to be identical so that the condition becomes $f(f(f(v))) = v$. In this way, each element in the array can receive the test pattern v by the additional application of $f(v)$ and $f(f(v))$ to the input of the array [21]. Appropriate test patterns must be applied on the non-propagating inputs to these cells (which are not shown on the figure) and that separate arrays can be configured for sequential and combinational elements [21]. The publication lacked however to present examples of such arrays although results were reported on them; however, an unpublished document was referenced with this data.

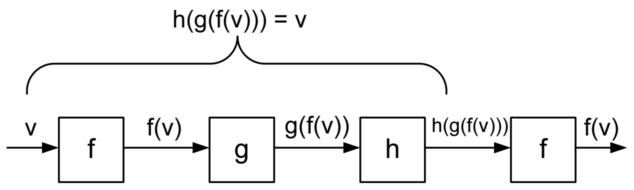

Figure 3.7: One dimensional ILA of length 3

Pipelining of the arrays under test is introduced in [20]. For a CLB with two outputs, the LUT output goes out of the CLB in a direct connection and another branch of it passes through a clocked flip-flop. The proposed test arrays are configured such that the outputs of the CLBs in an array alternate between the registered and the unregistered ones [20].This forces the same path delay for both branches and simplifies the construction of the TPG and ORA [20]. Three CLBs using this connection, each consisting of two LUTs, are shown in Fig. 3.8. Test configurations consist mainly of identity and inverting functions to test each LUT input separately for stuck-at faults [20].

A very interesting and different approach for CLB testing is used in [19]. First, to test the LUTs a *partial chain* is defined as four LUTs and flip-flops connected

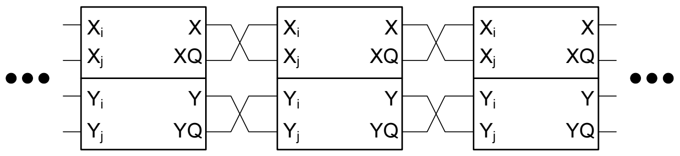

Figure 3.8: Interconnection of pipelined CLBs in an array

in series after a TPG counter. The configurations are chosen such that they completely test the LUTs under test, in addition, the output of this *partial chain* always toggles between "1" and "0" in the fault-free case [19]. The *partial chains* are then connected in series with the output connected to the clock input of the next *partial chain*. Any fault will distort the output such that a clock pulse becomes missing. The resulting error propagates through the array [19]. Multiple errors accumulate and are detected by analyzing the pulse of the final output [19]. The shortcomings of this approach are the test time and complexity. To test the LUTs eight configurations are necessary which take more than double the time compared to [13] which only require three configurations. In addition, test configuration is complicated; specific details have to be taken into account for each configuration such that the faults are not masked in the final output [19]. Implementation of multiple clocks in this manner may also be tricky since the design may not pass the design rule check (DRC) if each clock input needs to be connected to a clock buffer.

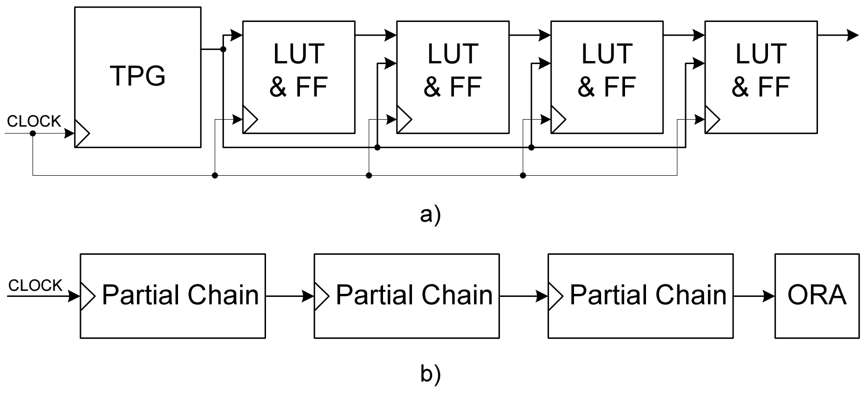

Figure 3.9: a) A partial chain, b) Connection of 3 partial chains

3.1.3 Memory Readback

Configuration memory readback is available in Xilinx Virtex series FPGAs, additionally, there are the options to capture the values in the CLB flip-flops or in the block RAM [7]. This provides the freedom of accessing the test responses through a different method other than scan chains. It is shown in [12] that response analysis can be done by memory readback through the JTAG boundary scan interface. The disadvantage of using this method is its slow speed.

Newer FPGAs such as the Virtex-4 FPGA have more options for memory readback operations, such as partial reconfiguration memory readback. There are also different ports such as the ICAP/SelectMAP interface which can operate at much faster speeds and are more flexible when compared to JTAG boundary scan [8]. In this way, the test configurations are organized such that there is a TPG and ORA for each component [22, 25, 26] and response analysis and diagnosis is done after the reconfiguration memory readback stage. Obviously, such a test would require at least three times the time overhead if compared to a single-fault detecting scheme such as array testing. However, this method provides complete observability and an excellent diagnosis resolution. Test configuration generation is also greatly simplified, since the test is reduced to testing a single component with no controllability or observability issues, but all the advantages come at the cost of a longer test time.

3.1.4 Test Configuration Minimization

A method is described in [27] that deals explicitly with the minimization of the number of required test configurations (TC). The authors start from three basic conditions for the testability of a module consisting of multiple subcomponents:

– **Condition 1:** All TCs are applied on each subcomponent in the module.

– **Condition 2:** All inputs of each subcomponent must be controllable. This is achieved by imposing constraints on the driving subcomponents.

– **Condition 3:** All outputs of each subcomponent must be observable. This is achieved by imposing constraints on the driven subcomponents.

3.1.4.1 Example TC minimization

The TC minimization algorithm is best described using an example. Consider the module in Fig. 3.10. It is a simple combinational block consisting of three multiplexers, with four inputs and one output. The first step is to derive the testability conditions of each component. Now consider MUX1; its conditions are derived as follows:

– **Condition 1:** To use all test configurations C_1 must take on both the “0” and “1” values in separate configurations.

– **Condition 2:** This condition is always satisfied because X_2 and X_3 are always controllable.

– **Condition 3:** Observability of the MUX1 output is obtained either through MUX2 with $C_2 = 1$, or through MUX3 with $C_3 = 0$.

These conditions are now combined in two boolean expressions expressing testability of MUX1 in the module shown. After the same is performed for the remaining

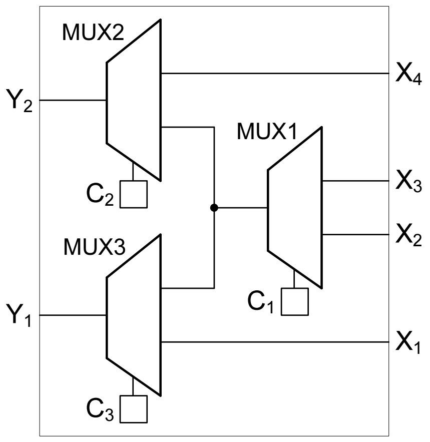

Figure 3.10: A simple module of multiplexers

two multiplexers, the following six conditions are compiled [27]. They are sufficient to express the testability of the entire module.

$$F_1(MUX1) = \overline{C}_1 \wedge (1 \wedge 1) \wedge (C_2 \vee \overline{C}_3) \tag{3.1}$$

$$F_2(MUX1) = C_1 \wedge (1 \wedge 1) \wedge (C_2 \vee \overline{C}_3) \tag{3.2}$$

$$F_3(MUX2) = \overline{C}_2 \wedge (1 \wedge (C_1 \vee \overline{C}_1)) \wedge 1 \tag{3.3}$$

$$F_4(MUX2) = C_2 \wedge (1 \wedge (C_1 \vee \overline{C}_1)) \wedge 1 \tag{3.4}$$

$$F_5(MUX3) = \overline{C}_3 \wedge ((C_1 \vee \overline{C}_1) \wedge 1) \wedge 1 \tag{3.5}$$

$$F_6(MUX3) = C_3 \wedge ((C_1 \vee \overline{C}_1) \wedge 1) \wedge 1 \tag{3.6}$$

It remains now to solve for a minimum number of configurations which satisfy all the mentioned testability conditions. For the three configuration bits C_{1-3} there are eight possible configurations as listed in Fig. 3.11. It is clear that only two test configurations (shown in red) are sufficient to satisfy testability for the whole module.

The authors from [27] then extend this method for all the other subcomponents found in a CLB and derive a minimum of five test configurations for a XILINX 4000 FPGA CLB [27]. The publication lacks, however, to present any implementation details about how the minimization problem was solved in this context and how it can be automated for larger circuits. This is remedied within the work of this thesis.

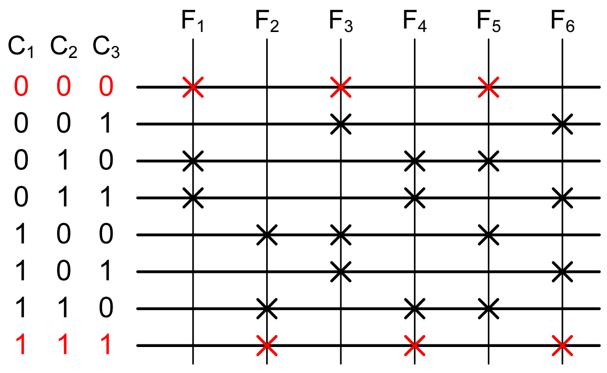

Figure 3.11: TC coverage of testability conditions

3.1.5 Summary of CLB Test Approaches

Direct comparison of the number of configurations between tests is not very meaningful when analyzing results implemented on different FPGA families or architectures. This is because the logic structure differs from one FPGA to the next. However, after analyzing the broad methods for CLB test; they can be compared against each other objectively, regardless of the number of configurations or other test metrics presented in the respective publications.

3.2 Interconnect Test Approaches

Testing interconnects (or wires) requires different considerations than those applied for logic testing. The test itself is usually quite simple, since it only consists of passing "0" and "1" patterns through the wire under test (WUT) and checking whether the output follows the input. This does not undermine its importance in any way; not only do interconnects use up more than 80% of the total FPGA configuration [4] bits but they are becoming ever more complex in newer FPGAs as well [5].

After going through basic principles and concepts of interconnect testing found in the literature, advanced methods and algorithms are discussed which would be applicable to high-end FPGAs today.

3.2.1 Basic Interconnect Testing

There are many publications [28, 29, 30, 31, 32, 33, 34] that present the concepts and definitions associated with FPGA testing without the associated technical implementation considerations and difficulties. Those are the basics for interconnect testing but do not count as methods for FPGA test, simply because the aforementioned implementation difficulties constitute the main interconnect test problem.

3.2.1.1 Interconnect/Logic Interface

Two publications [29, 30] deal with the test of the “configurable interface module” (CIM) that is present between an interconnect network and the CLBs. The authors assume two very simple implementations for the CIMs; n-input multiplexers, or pass transistors [29, 30]. Although this is a valid assumption for very simple FPGAs, it is irrelevant to today’s FPGAs in which the interconnect network has become very complicated and can no longer be abstracted to such a degree. The authors then discuss multiplexer testing and argue that the input CIMs to a CLB require n configurations, one for each possible wire. On the other hand, the output CIMs only require two TCs since many outputs can be observed simultaneously.

3.2.1.2 Interconnect Wires

The stuck-at, open and bridging fault models are widely adopted in the literature in the context of interconnect testing [28, 31, 33]. Most FPGA layouts are proprietary and unavailable to the test engineer, making tests for bridging faults very rough and involving many assumptions [31]. The main issue with interconnect test is however the configuration of the wires under test (WUT) such that they can be tested with the minimum number of FPGA reconfigurations.

An assumption is widely adopted for dated research in this area[28, 31, 32, 33, 34], and is suitable for older FPGAs. This assumption is that a programmable switch matrix (PSM) has four identical sides as shown in Fig. 3.12. That means, that the number of north/south/east/west wires are all identical in number and behavior making their test a simple task. Irregularities from this symmetric structure are then handled separately [28].

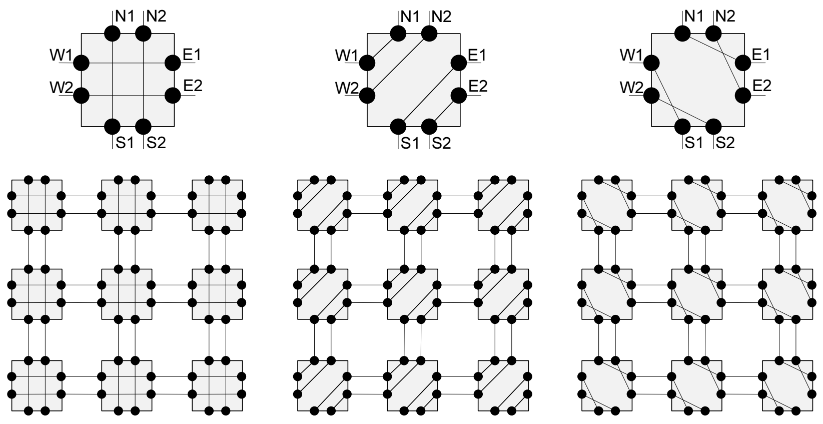

Figure 3.12: Three test configurations for non-redundant fault coverage

Fig. 3.12 shows three test configurations which are sufficient to test all the mentioned stuck-at/open/bridging faults [28]. These configurations are a lower bound for the number of configurations required to test the FPGA interconnects [28]. After the connection in such a configuration, the twelve WUT ends are simply connected to TPGs and ORAs. The same publication [28] explains that there are two categories of faults assumed; those inside the PSM logic and those on the WUTs.

Note that only single hop wires are assumed so far. When an example Xilinx 3000 FPGA is considered for FPGA test following the same concept [28], double wires are identified and they require an additional separate handling of their test. This suggests that simply finding orthogonal configurations for interconnect wires on a uniform PSM assumption is not sustainable; current FPGAs have at least five kinds of interconnect wires [5]. In addition, the configuration presented requires many IOBs for the application of the test patterns and the same for observing the responses. This shortcoming was addressed in the paper by connecting the various wires together in a longer length WUT.

The authors in [31] give a more in-depth study of FPGA interconnect testing which is also based on the simplifications mentioned in the previous paragraphs. the WUTs are configured into so-called *ladders*. It states additionally the test patterns used for testing a *bunch* of wires for stuck-at/open/bridging faults. An exhaustive 2^n patterns are applied for a group of n WUTs, furthermore; walking patterns are also utilized ("1" in a field of zeros and vice versa). Test patterns and responses are applied and collected using on-FPGA configured TPGs and ORAs, they are finally read out using FPGA boundary scan.

Seven test configurations are derived in [33] for the simple PSM assumption, but guarantees a better diagnosis. In [34] the same three basic configurations (shown in Fig. 3.12) are explained in the context of testing the interconnects in a multiple FPGA system. In [32], interconnect test time is reduced by reconfiguring the FPGA *during* testing. This is done by using a linear feedback shift register (LFSR) attached to each PSM and is therefore unsuitable to existing FPGAs without this feature.

3.2.2 Advanced Interconnect Testing

In this subsection, advanced methods for interconnect testing are described. The structure of the PSM is explicitly accounted for and the main problem in interconnect testing, namely the routing of WUTs, is tackled. Unlike basic approaches, no assumption on the PSM connections are made.

3.2.2.1 Cross-Coupled Parity BIST Approach

A systematic approach for testing the different interconnect types in Xilinx Virtex-4 FPGAs is given in [4, 25, 26]. Global routing resources are classified into their different types depending on their direction (North/South/East/West), their length, the number of hops between connections and their buffer type (unidirectional/bidirectional) [4]. Test configurations are then manually devised for each wire classification following a *divide & conquer* approach.

To account for faults in the TPG/ORA a cross-coupled parity based scheme is used [4]. The authors also claim that using this scheme the number of test configurations are minimized since a higher number of WUTs are supported as well as an odd number of WUTs; claimed to improve on previous counter and parity-based approaches [4].

The publication [4] and the corresponding theses [25, 26] give some insight on implementation details for low level configuration of Xilinx FPGAs. The authors use Xilinx Design Language (XDL) for low-level design entry, and it is automated using *C* programming for various Virtex FPGAs. The methods used however lack generality and are very specific to Virtex-4 FPGA. In addition, a lot of design time is required to derive the different configurations for each interconnect classification.

3.2.2.2 Max-Flow Approach I

To find the minimum number of test configurations for a PSM, while satisfying routing constraints, a modified max-flow algorithm is introduced in [35]. This algorithm is designed to test the PSM itself and not the wires as illustrated in Fig. 3.13. The algorithm runs on one PSM and is repeated for identical PSMs [35].

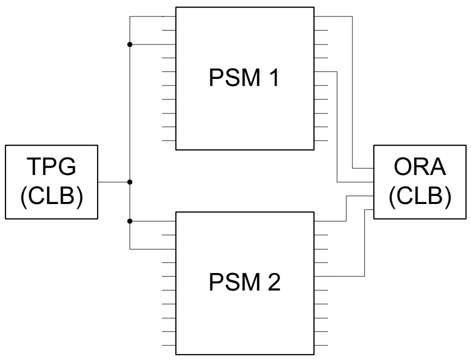

Figure 3.13: Test structure for testing PSMs using max-flow approach

PSM elements, such as pass transistors and multiplexers are mapped to a graph representation. Then they are sorted in three different groups as input to the algorithm:

1. TPG (CLB) outputs to PSM inputs.
2. PSM outputs to ORA (CLB) inputs.
3. PSM inputs to PSM outputs.

The node groups are explicitly listed in this order, when entered to the algorithm to allow the connections with lowest number of possible paths, to be routed first [35]. The modified max-flow algorithm [35] generates test configurations as summarized in the following pseudo code:

```
1 Initialize weights
2 repeat
    3 Run max-flow(group 1)
    4 Run max-flow(group 2)
    5 Run max-flow(group 3)
    6 Calculate fault coverage
    7 Increment weight on used edges
8 stop when fault coverage is 100%
```

The max-flow algorithm runs many iterations on the set of unrouted nodes; giving priority to the nodes where the faults have not yet been detected. This is implemented by associating a weight with each edge (which represents a wire segment). Whenever an edge is used in a test configuration a fault associated with it is tested for so the weight of that edge is incremented. The max-flow algorithm chooses the lower-weight edges with higher probability ensuring that untested wires are accounted for first [35]. Additional considerations include assuring that each node can only have a single driver, this is done by assigning a *capacity* of one to each node.

The authors claim a polynomial complexity for the max-flow algorithm when solving groups 1 and 2. In routing group 3 there are more possibilities causing the algorithm's complexity to increase and it becomes non-polynomial [35]. Implementation is done using Xilinx JBits Java framework for low-level configuration of FPGAs and results are presented for sample Virtex-2 FPGAs [35]. Note that JBits is no longer supported and only has access to the Virtex-2 family of FPGAs so the experimental results presented are obsolete.

3.2.2.3 Max-Flow Approach II (1-N Mapping)

A more recent approach also utilizes the max-flow algorithm to derive test configurations for FPGA interconnects [36]. It is called 1-N mapping since this is the general case of programmable interconnect points (PIP) in FPGAs; one node can connect to multiple (N) destinations. Fig. 3.14 shows three FPGA switch matrices in series with east connections, its graph representation and a potential path through the graph. This diagram is used to simplify explanation of the algorithm.

The algorithm handles one interconnect direction at a time, attempting to find paths through the graph from source to sink. The nodes and edges $s' \to s$ and $t \to t'$ are added to the graph description with capacities k, where k is the number of wires to be routed at a time limited by the number of flip-flops between switch matrices. This is because the authors opt for a *buffered* test scheme with all WUTs going through flip-flops between PSMs [36]. All other graph edges have the capacity of one.

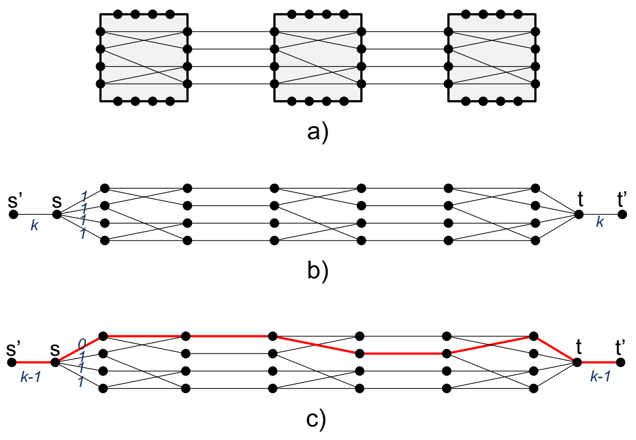

Figure 3.14: Graph representation of east interconnects inside/between three switch matrices

The lift-to-front implementation of the Ford-Fulkerson method for solving max-flow problem is used for its performance [36]. Whenever a path is selected, all the edges capacities are decremented to zero and the $s' \rightarrow s$ and $t \rightarrow t'$ capacities are decremented by one. These PIPs and wires are subsequently removed from the set of wires to be tested, then the algorithm runs again until $k = 0$. This is furthermore globally repeated with $s' \rightarrow s$ and $t \rightarrow t'$ reinitialized to k for a different test configuration until all edges are removed from the set of edges to be tested [36].

The limiting factor for this algorithm is k; the number of flip-flops per CLB [36]. This is circumvented by *interleaving* the test pipeline; that is, not every wire is buffered after every PSM. This provided a reported improvement from 60 to 8 configurations [36].

Although the authors report a fast run-time of the algorithm, it lacks any utilization of the inherent symmetry found in FPGA PSMs.

3.2.2.4 Graph Edge Coloring Approach

With Virtex FPGAs in mind, an automatic interconnect test configuration generation method based on graph edge coloring algorithms is introduced in [37]. The interconnection of CLBs to the routing network is modeled using a graph following the schematic shown in Fig. 3.15. CLB inputs pass through an input routing matrix (IRM) whereas the outputs are routed through an output routing matrix (ORM). The next stage is a global routing matrix (GSM) which corresponds to a PSM according to naming convention in this text. This connects each CLB to the global interconnect network [37].

Each switch matrix (I/O/GRM) is modeled using a bipartite graph where wire segments are represented by vertices and PIPs by edges [37]. Note that this is the opposite of what was adopted in the max-flow algorithms [35, 36]. A bipartite graph is one that has two groups of nodes and a node can only be connected to

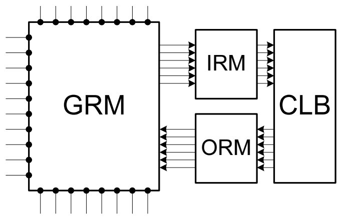

Figure 3.15: Schematic of the interconnect routing structure

another node if it is not in the same group; that is, the two groups of nodes are disjoint [37]. These two disjoint sets model the input and output wire segments to a switch matrix, while an edge between two nodes signifies that a connection is possible between them [37]. The bipartite graphs of each switch matrix are then combined into a k-partite as shown in Fig. 3.16 to model the entire interconnect network.

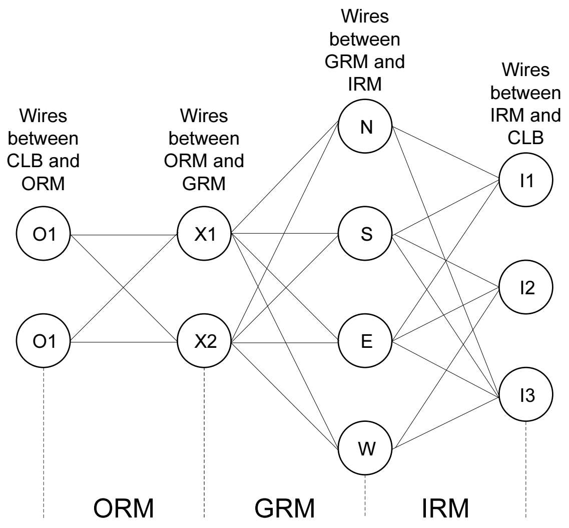
Wires between CLB and ORM
Wires between ORM and GRM
Wires between GRM and IRM
Wires between IRM and CLB
O1
O1
X1
X2
N
S
E
W
I1
I2
I3
ORM
GRM
IRM

Figure 3.16: k-partite graph representing interconnects

For finding the test configurations, and edge coloring problem is solved on the k-partite graph [37]. That means, all the edges are colored such that no two connected edges have the same color. When the minimum coloring is achieved, each color represents a different test configuration guaranteeing full coverage using a minimal or near minimal set of TCs [37].

It is unclear why the derived test configurations required two test phases in which only half the FPGA was tested at a time while the other half is configured as TPG/ORA [37]. The results were reported as 26 configurations for an unspecified model of Virtex FPGAs [37].

Chapter 4

Fault Model

Contents

The stuck-at fault model is widely adopted in the literature for FPGA testing [13]. It is suitable for a simplified abstraction of structural defects for all known structural implementation information. Unfortunately, most FPGA subcomponents such as LUTs and flip-flops have hidden implementation details since their intellectual property (IP) is proprietary. This usually results in a weak modeling of defects and a reduced number of faults.

In this thesis, the stuck-at fault model is assumed for components and interconnections in which no additional structural/functional details are relevant for fault derivation. On the other hand, any additional structural/functional knowledge is used to derive an additional list of faults which models the component in a more accurate way. For example, functional RAM faults are accounted for such as transition and coupling faults.

This chapter is organized into sections, each of which introduce the fault model used for each component; starting with the functional faults, then the stuck-at fault model is used for the remaining units. After that the faults are compiled in a list separated into structural and functional faults assumed for an entire CLB.

4.1 The Cell Fault Model

The cell fault model (CFM) [38, 39], also called the black-box fault model [40, 41] is used for modeling combinational faults for LUTs in LUT mode (function mode). In this section, after defining the fault model, the suitability of this model for LUT testing is explained.

4.1.1 Definition and Assumptions

CFM is an exhaustive functional fault model which makes no assumptions on the structure of the CUT; it models any fault which causes a deviation from correct combinational behavior. For a given "cell" under test, the CFM assumes that any output other than the expected one constitutes one or more combinational faults inside the cell under consideration [39]. This erroneous cell output is termed a cell fault. No additional information on the number of internal defects, their type, or location is available. Furthermore, only a single "cell" is allowed to be faulty at a time; but the fault can modify the cell function in any arbitrary combinational way.

The CFM is much more thorough than the traditional stuck-at fault model [39] since it models any fault that would alter the function of a cell. All single and multiple stuck-at faults inside a faulty cell are tested for as a subset of all cell faults that may occur. In addition, it is suitable for the IP design paradigm because the circuit implementation can be kept hidden. It also does not matter which vendor library is used for the implementation of the cell [39]. In addition, faults in the interconnects of a cell are implicitly accounted for.

One disadvantage of the CFM is the need for an exhaustive test of the cell under test, i.e. all the input combinations must be exercised for full coverage of CFM faults.

For any cell, such as that shown in Fig. 4.1, the number of cell faults can be derived. A cell fault is one that alters the *function* of the cell as stated previously. That is, a fault which forces a different output than the expected one. With that in mind, the number of different inputs are 2^m each of which has only 1 fault-free output but $2^n - 1$ faulty ones. The total number of cell faults are therefore $2^m(2^n - 1)$ [39].

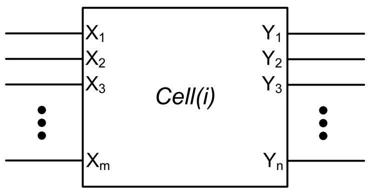

Figure 4.1: m-input, n-output cell

4.1.2 Example Fault List Derivation

The fault list is derived for a 2-input 1-output cell in Fig. 4.2; it consists of the list of faulty outputs for each different input to the cell. The fault list is shown in Table 4.1. The number of cell faults are equal $2^m(2^n - 1) = 4$ with $m = 2$ and $n = 1$; there is one faulty output per input combination since there is only one output.

Note that the possible implementation shown in Fig. 4.2 plays no role in defining the set of cell faults associated with the cell. This is to emphasize that CFM make no assumptions on the underlying structural implementation.

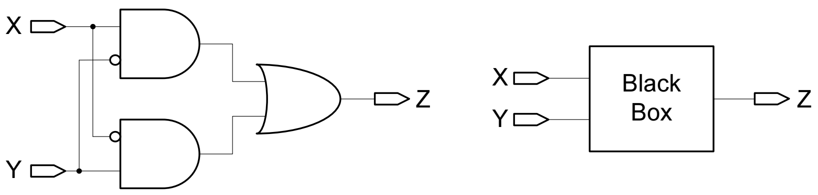

Figure 4.2: XOR gate implementation and abstraction to a black box

X	Y	*faulty* Z
0	0	1
0	1	0
1	0	0
1	1	1

Table 4.1: List of cell faults for the XOR gate

4.1.3 Lookup Table: LUT Mode Fault List

The Virtex-5 FPGA LUT is implemented as two 5-input LUTs with a multiplexer to choose between the two LUTs; the sixth input controls the multiplexer. This is shown in Fig. 4.3.

It is mentioned at the beginning of this chapter that an exhaustive CFM approach will be applied to the smallest structures of which the implementation is not known. Since this structure of the LUT is known and shown in Xilinx documentation [5], it will be adopted for the derivation of the associated fault list. Also note that there are three different modes of operation for the LUT block: LUT, shift register (SR) and RAM. For each mode, a set of faults will be derived based on the functionality of the block in the relevant mode.

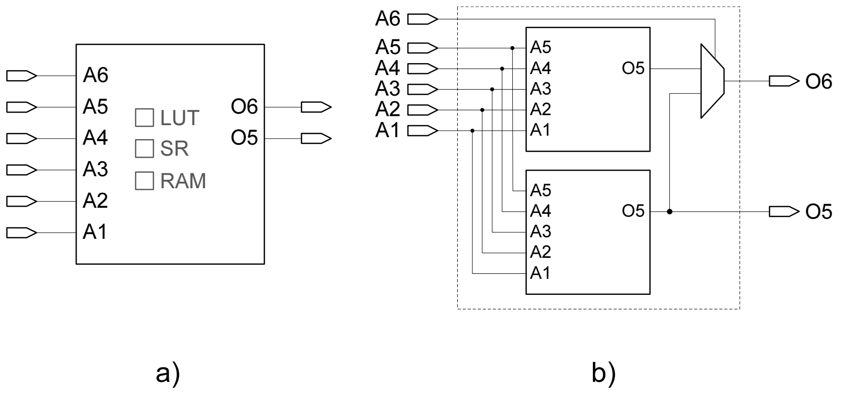

Figure 4.3: a) Virtex-5 LUT and b) details of its structure

In LUT mode the structure of the LUT is considered down to the most fine-grained description available by the manufacturer; in this case, it is the device shown in Fig. 4.3. Without that knowledge, the number of cell faults would be calculated directly as $2^6(2^2 - 1) = 192$, because there are 6 inputs and 2 outputs. After knowing these implementation details however the number of functional faults for the LUT is derived as the sum of the cell faults for each sub-component. Note that the “O5” output as well as the inputs A1-A5 have to modeled separately for stuck-at faults since the wires fan-out. In general, faults at fanout branches are not equivalent to faults at the stem and need to be explicitly handled.

The number of cell faults from each 5-input LUT are equal $2^5(2^1 - 1) = 32$ and the 2-input multiplexer has 6 stuck-at faults whereas the fanouts have 10 stuck-at faults. The total number of faults for the 6-input LUT in *LUT mode* is separated into structural stuck-at faults and functional cell faults. Structural faults are abbreviated as “SF” and functional faults are abbreviated “FF” throughout this chapter.

$$SF(6LUT) = SF(2MUX) + SF(fanouts) = 18 \tag{4.1}$$

$$FF(6LUT) = 2 \times FF(5LUT) = 64 \tag{4.2}$$

This is a significant reduction from 192 cell faults to just 64 FFs and 20 SFs, achieved by analyzing details of the available abstract implementation.

4.2 Functional RAM Fault Model

A RAM test is required to cover all functional errors of which a RAM module could suffer. This work assumes be to follow the classic fault models associated with RAM and therefore the following faults [42] are considered:

1. Address decoder faults (AF): Accessing the wrong address due to faults in the address decoder.
2. Stuck-at faults: A memory cell is stuck on a ”0” or ”1” value.
3. Transition faults: The inability of a cell to switch from $0 \rightarrow 1$ (slow to rise) or from $1 \rightarrow 0$ (slow to fall).
4. Coupling faults (CF): Memory cells assume an erroneous value depending on the switching activity in neighboring cells.
5. Data retention faults (DRF): A memory cell fails to retain its data value after some time.

Deeper analysis of RAM can lead to a more accurate set of fault models and there is already progress in that area, but the extent presented is sufficient to this work. Due to hardware overhead constraints, only the most important and dominating faults will be accounted for. The reduced functional fault set for RAM in context of this work is therefore defined as all AFs, SAFs and TFs in the memory cells. It is experimentally proven in [42] that SAFs and stuck-open faults (SOFs) abstract more than 72% of defects in a RAM module. These set of faults are assumed to dominate other possible faults which could occur on the write enable or clock circuitry, because of their functional impact on the RAM.

Consider the FPGA LUTs which can be configured in RAM mode. For an N-bit RAM, there are n address inputs, a data-in input, a data-out input as well as the clock and write-enable signals [5]. From the assumptions stated above, there are 2^n AFs, 2^{n+1} SAFs and 2^{n+1} TFs yielding a total number of functional faults:

$$FF(RAM) = 2^n + 2^{n+1} + 2^{n+1} = 5 \times 2^n = 5N \tag{4.3}$$

Where N is the number of memory cells. For example, the Virtex-5 LUT can support a 64-bit RAM with 320 functional RAM faults based on the assumptions above. There exist many march test algorithms [42] which account for all the stated RAM faults.

4.3 Functional Shift Register Fault Model

Although the same LUT structure is used for running LUT mode, SR mode and RAM mode, a different set of faults are derived for each to guarantee functionality in each specific mode. Fig. 4.4 shows a functional view of a shift register. It is an interconnection of the LUT SRAM cells in series; these connections are not used when in LUT mode or RAM mode since they are between SRAM cells and are not connected to the LUT address decoder.

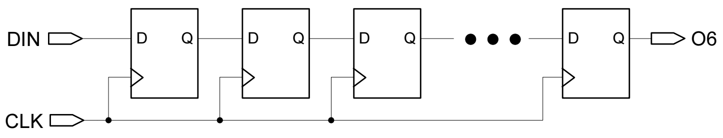

Figure 4.4: Functional view of a shift register

In SR mode, as it is for flip-flop testing, the faults assumed are both kinds of stuck-at faults in the sequential elements as well as the two transition faults (TF). There are therefore four faults per flip-flop and they dominate the possible faults on the interconnection between flip-flops. For a shift-register of length N the total number of faults are equal $4N$.

4.3.1 Flip-Flop Fault List

Similar to shift registers, transition faults and stuck at faults are considered during flip-flop testing. Virtex-5 memory elements can be configured as either flip-flop or latches and have the option of being initialized to “1” or “0”, and the reset (SR) can either be active-high or active-low. This is shown in Fig. 4.5. The different inputs and outputs are also shown, the are the clock (CK), reset, data-in (D) and data-out (Q) terminals as well as clock enable (CE) and reverse (REV); this writes an opposite (reverse) value to that inferred from the reset input.

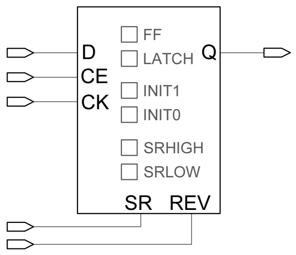

Figure 4.5: A Virtex-5 flip-flop

Two faults will be assumed on each input pin, a total of 10. In addition, there are three control inputs: the mode (FF/latch), initial value (0/1), and the active reset level (high/low). The INIT(0/1) and SR(LOW/HIGH) will be considered as inputs similar to REV and CE; they each have two associated faults. However, it will be assumed that all the inputs and outputs have to be tested in each mode (FF/latch). The structural faults equal $7 \times 2 \times 2 = 28$ since there are seven inputs, assumed to be disjoint, having two SAFs each, and it is multiplied by two again to reflect both FF and latch mode. In addition there are two transition faults introducing four additional functional faults. (SF=28, FF=4)

4.4 Stuck-At Faults

Following the discussion of functional faults, three elements from the CLB remain unaccounted for. Multiplexers, XOR cells and some of the wires which connect all the components together in a CLB. Structural SAFs are systematically derived for these components.

XOR gates are used in the carry chain in a Virtex-5 CLB. There are three terminals in an XOR gate each having two possible stuck-at faults. Unlike AND/OR gates, the XOR gate has no controlling value, so none of the six SAFs are equivalent. The faults are annotated in Fig. 4.6 as “0” for SA-0 and “1” meaning SA-1.

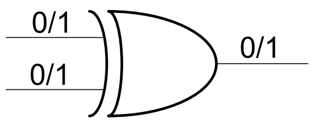

Figure 4.6: XOR gate stuck-at faults

Multiplexers are a slightly different case. For a 2-input multiplexer (Fig. 4.7), the faults at the output are dominated by the SAFs on the inputs. If the select input is equal “0”, if the output is SA-0 for example; this indicates whether a SA-0 on the data input or the output occurs, suggesting that these faults are equivalent.

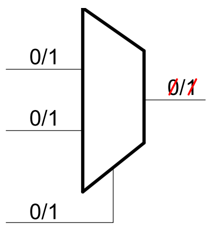

Figure 4.7: 2-input multiplexer stuck-at faults

It is assumed that all faults on the wires entering a component are dominated by the faults in the component itself. At fanouts, the faults have to be accounted for separately as shown in Fig. 4.8. At buffers, faults at the input and output are also considered equivalent.

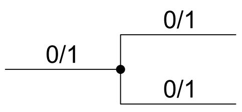

Figure 4.8: Stuck-at faults in fanouts

4.5 Complete CLB Fault List

Fig. 4.9 gives an overview of the hybrid fault model assumed for the FPGA logic elements. The stuck-at fault model is used for modeling structural faults. Functional faults are also accounted for using a set of component specific functional fault models. The CFM is used for exhaustively modeling combinational faults in the LUT whereas more specific fault models are used to abstract defects in the RAM and flip-flops.

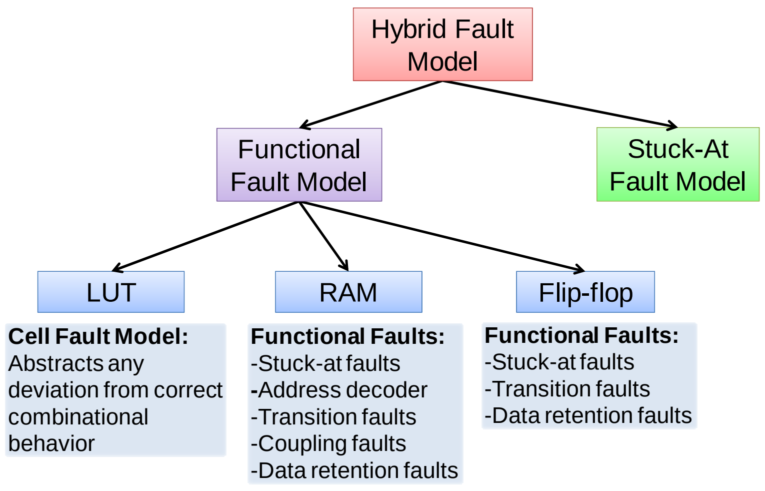

Figure 4.9: Overview of the hybrid fault model

At this point it is important to note that there are many structures tested twice when testing the LUT in the different modes stated above. For instance, SAFs are accounted for in all three modes. These faults are considered multiple times to ensure that the LUT gives a functionally correct output in each of the different modes of operation and also increases coverage of faults in the control circuitry which configures the LUT. These faults are however not included in the fault lists derived because they are assumed to be dominated by the aforementioned faults. For example, if the control circuitry is stuck on LUT mode but RAM mode is used, the output will certainly not be as expected while preforming a RAM test such as MATS+. In addition, no details of the control circuitry is available.

Now that the assumed faults are defined for each CLB component, their sum is calculated to find the total number of faults assumed for a complete CLB, to be used for fault coverage calculation. This is under the assumption that all faults are observable and all inputs are controllable so that each component is fully tested. This is taken care of later in devising the test configurations.

Fig. 4.10 shows one quarter of a slice which is one half of a CLB. Two types of slices exist: sliceM, which includes RAM/SR mode and sliceL which does not. In addition to components in Fig. 4.10, there is a clock multiplexer which has the option of inverting the clock, a carry-out buffer and a carry-in buffer [5]. Furthermore, there is a carry initialization MUX and three fixed MUXs for advanced CLB functions, but these are disregarded in this work.

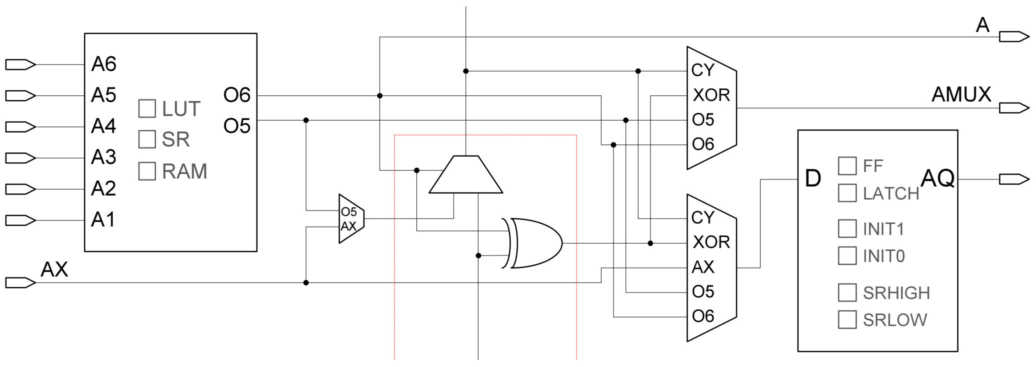

Figure 4.10: Simplified “quarter” CLB circuit diagram

The total number of faults are compiled in Table 4.2. They are split into structural faults and functional faults according to the analysis in this chapter. Faults are derived manually from the limited structural knowledge available from Xilinx. Thus, they shall only serve as a means to roughly asses the fault coverage of the test configurations presented later in this thesis. The goal from this work is to provide 100% coverage of the mentioned faults.

Component	Mode	SAFs	Cell Faults	SR/FF Faults	RAM Faults
Lookup Table	LUT	18×4	64×4	-	-
	SR	-	-	128×4	-
	RAM	-	-	-	320×4
Flip-Flop	FF	14×4	-	2×4	-
	Latch	14×4	-	2×4	-
Output MUX	-	12×4	-	-	-
Flip-flop MUX	-	16×4	-	-	-
Carry-in MUX	-	6×4	-	-	-
Carry-out MUX	-	6×4	-	-	-
Carry XOR	-	6×4	-	-	-
Wires	O6	2×4	-	-	-
	AX	2×4	-	-	-
	C_{OUT}	2	-	-	-
	CLK	2	-	-	-
	CE	2	-	-	-
	SR	2	-	-	-
CLK MUX	-	6	-	-	-
$C_{IN}MUX$	-	8	-	-	-
SYNC/ASYNC	-	2	-	-	-
Total sliceM		408	256	528	1280
Total sliceL		408	256	16	-
Total CLBLL		816	512	32	-
Total CLBLM		816	512	544	1280

Table 4.2: Summary of CLB faults

Chapter 5

CLB Test

Contents

Several CLB test approaches are found in the literature and are explained in Chapter 3. The various methods show a trade-off between test speed, diagnosability and BIST hardware overhead. The approach taken in this work is array testing for FPGA logic following the concepts presented in Section 3.1.2.

This chapter starts with an overview of the test methodology and BIST architecture used. An abstract explanation of testing iterative logic arrays is also presented. The details of testing each logic subcomponent are then explained, followed by the CLB test optimization method.

5.1 CLB Test Architecture

A Virtex-5 CLB consists of many logic components such as LUTs, multiplexers and flip-flops. They can be interconnected together in in many different combinations reflecting different variations of using a logic slice. This necessitates the use of multiple test configurations for complete controllability and observability of the components under test. The FPGA is reconfigured multiple times into so-called test configurations (TC) in which each TC is able to test a subset of the complete logic slice. Full coverage of faults inside the logic is ensured by deterministic design of TCs.

This section presents the general test methodology before going into the BIST architecture used including an explanation of the TPGs and ORAs used.

5.1.1 Test Methodology

In FPGAs, a container under test requires multiple reconfigurations for full-coverage testing. Each reconfiguration targets a specific subcomponent in each logic slice. A single TC consists of two main components:

1. Container setup: Logic slices are configured in a specific way to ensure the test of specific subcomponents.

2. BIST infrastructure: TPG and ORA for each container setup.

After designing an appropriate set of TCs, bitstreams are generated according to the container size and they are repeatedly configured onto the FPGA and tested. This configure-test cycle is repeated for the number of TCs designed for full coverage of CLB faults. The process entails the following steps (illustrated in Fig. 5.1):

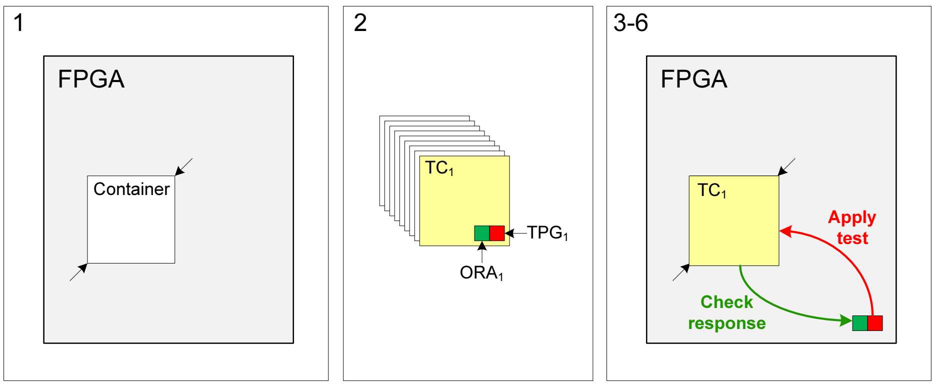

Figure 5.1: Container test procedure

1. Specify container: A container can be of any rectangular size on the FPGA specified by any two coordinates.

2. Generate TCs: Bitstreams containing the container setup and BIST hardware are generated.

3. Download TC$_i$: The container under test is configured with each TC.

4. Apply test patterns: For each TC, the suitable test patterns are applied from the TPG.

5. Evaluate test responses: For each TC, the response is gathered and input to an ORA.

6. Repeat steps 3 to 5 until all TCs are processed.

5.1.2 BIST Architecture

The need for a simple test method/architecture comes from the basic structure of the FPGA. While FPGAs are getting more complex on a CLB level, they maintain their array architecture. This array structure needs to be exploited for creating a general test architecture extensible to any FPGA architecture. Furthermore, it is discussed in Chapter 3 that array testing of FPGAs has the lowest hardware overhead and test time at the expense of a lower diagnosability. PRET is only concerned with fault detection and not localization or diagnosis making an array test for it most suitable.

Concepts from array test are used for testing the FPGA fabric. Combined with the test approach outlined in the previous section, this leads to a straightforward test technique which is easily scalable and portable to other FPGA architectures. In addition, tests are pipelined to ensure high test application speeds.

5.1.2.1 Container Setup

Each CLB is configured using the designed set of TCs. Each TC is designed such that the logic slices can be connected into a C-testable array. This is shown in Fig. 5.2 with TC_1 placed in the CLBs. A TPG then feeds the test patterns at the start of each array and the responses are collected at the end of the array using an ORA.

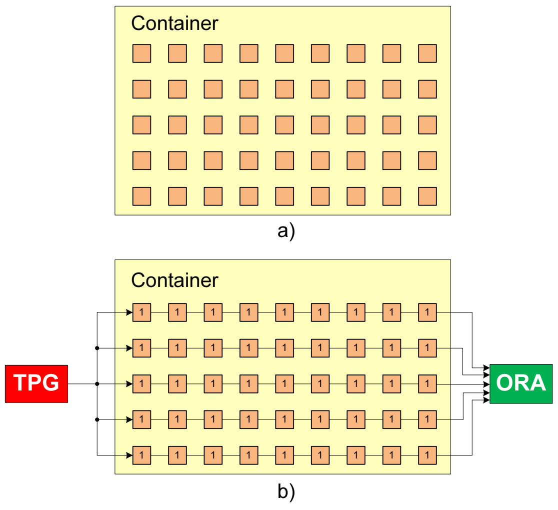

Figure 5.2: a) Empty container and b) configured into arrays

This topology allows the usage of very simple comparison-based ORAs. The ORA is therefore the same for all different TCs whereas the TPG is test-specific for the most part. Comparison-based ORAs are implemented using XOR gates for detecting single errors. For an even number of arrays, the outputs are compared using an XOR gate. For an odd number of outputs, a combination of XOR/OR gates are used as shown in Fig. 5.3 to avoid masking of errors.

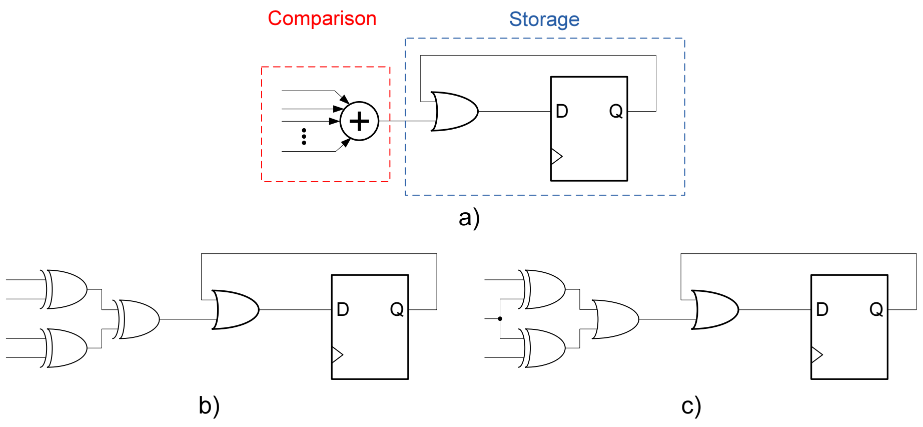

Figure 5.3: a) Comparison-based ORA for b) four and c) three array outputs

The comparator must only ensure that a "0" is output when all array outputs are identical and "1" otherwise. The logic circuit can then easily be derived from the truth table describing that functionality. After the comparison stage, there is a storage stage as shown in Fig. 5.3. This makes sure that each result is saved in the ORA flip-flop to have a single value at the end of a test-run indicating whether a fault is detected.

5.1.2.2 Test Pipelining

Configuring CLBs into arrays can lead to a very long critical path for large container sizes. This dictates the use of very slow test clocks which are additionally dependent on the container size. To avoid these limitations, the test configurations are pipelined, allowing tests to run at system speed (MHz) instead of kHz. This is demonstrated in the results section to provide an increase in test speed in the order of $1000 - 10000$.

The way to pipeline logic tests is by utilizing the sequential elements included in each logic slices. For an exhaustive test, the unregistered outputs must also be tested. To allow that, an array interleaving scheme such as that shown in Fig. 5.4 is used [20].

5.1.3 Testing Iterative Logic Arrays

The concept of testing arrays is very old [38]. It has been shown in [24, 38] that upon the fulfillment of some conditions. An array of arbitrary size is fully tested by applying exhaustive test patterns at the inputs of the first cell and observing

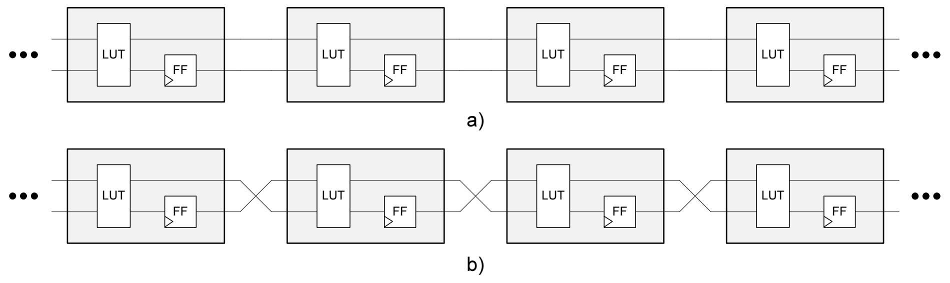

Figure 5.4: a) Partially pipelined and b) fully interleaved/pipelined CLB arrays

the output at the end of the array. Furthermore, these conditions are imposed on a unit cell which is replicated throughout the array. This is precisely the definition of C-testability. A C-testable array is one that can be fully tested with a fixed number of input patterns for any finite length of the array [24].

5.1.3.1 Conditions for C-testability

Two general conditions must hold when testing for single faults in a logic array [38]:

- **Condition 1:** Each cell in the array must have an exhaustive set of test patterns applied at its inputs.
- **Condition 2:** For each test pattern mentioned in condition 1, the output of a cell must be sensitized until the array output, or some other intermediate output present without disturbing the array structure.

Taking these conditions into account, more specific constraints can be specified for an ILA such as the one depicted in Fig. 5.5. Although this is not the general model for a logic array (potentially having intermediate outputs or being two dimensional), it is sufficient for representing the CLB components.

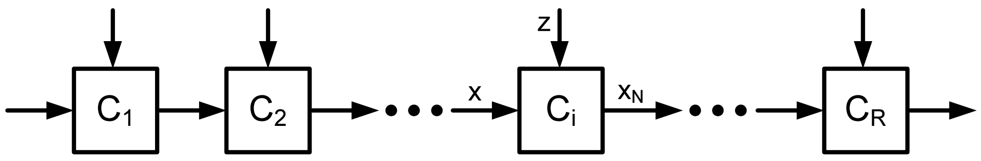

Figure 5.5: A one dimensional logic array of length R

In Fig. 5.5: x is the cell state , x_N is the next state resulting from the function of a cell C_i and z represents any number of external inputs to each cell.

The first cell C_1 in an array always has condition 1 satisfied because its inputs are controllable, being at the beginning of the array. As for the second cell C_2, it is required that the propagating value x_N takes on all possible values of x. This will guarantee that all the test patterns applied to C_1 will also be applied at the inputs of C_2, guaranteeing that it will also be exhaustively tested. This constraint also holds for all other cells and can be generalized for the entire array. In terms of the flow table for the cells C_i, this means that every input combination x must also appear in the output x_N.

The second condition states that the output of each cell must be sensitized until a primary output. This ensures observability of detected faults in each cell. In Fig. 5.5, there is only one primary output at the end of the array which is the worst case. For the last cell C_R, this is no problem since we can directly observe the cell outputs. As for the cell before the last C_{R-1}, there has to be at least one z input combination which will change the value of x_N when x changes. This is sufficient to expose an error in cell C_{R-1}, and similarly this condition is generalized for all previous cells C_i. In terms of a flow table, this means that no two rows are allowed to be identical. These conditions are now restated:

– **Condition 1:** x_N must assume all the values of x. That means that all x must appear in the flow table outputs x_N.

– **Condition 2:** There must be at least one z combination which sensitizes the path from x to x_N. That means that no two rows can be the same in the flow table.

These conditions are illustrated with an example. A very simple case of an iterative array is the parity checker shown in Fig. 5.6. The corresponding flow table is shown in Table 5.1. It is obvious from this simple example that both conditions are satisfied since all values of x show up in the table and no two rows are identical. This array is therefore C-testable. By applying the exhaustive input patterns, all errors can be observed at the array output.

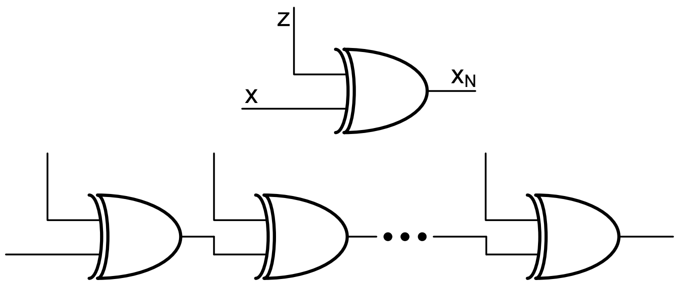

Figure 5.6: Parity checker cell and array

5.2 CLB Subcomponent Tests

In designing tests for CLBs the "divide & conquer" approach is adopted. Each CLB subcomponent is handled separately and tested for all the faults associated with it (structural/functional) in Chapter 4. The CUT consists mainly of the structure

		z	
		0	1
x	0	0	1
	1	1	0

Table 5.1: Flow table for XOR cell

illustrated in Fig. 5.7 which is repeated four times in a logic slice. It shows the basic building blocks of a CLB.

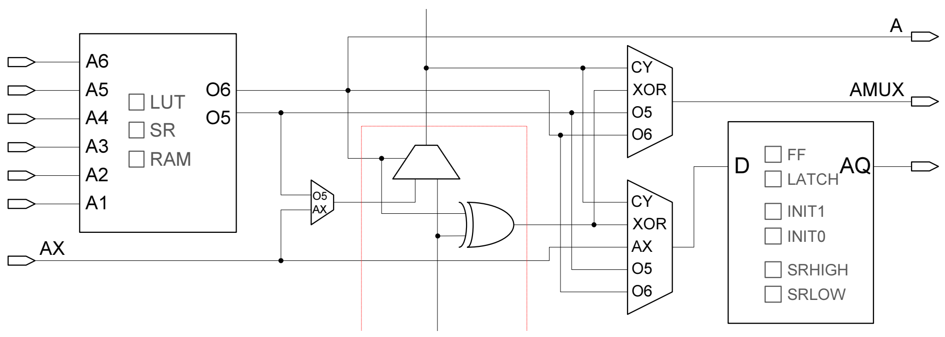

Figure 5.7: Simplified "quarter" CLB circuit diagram

In this section, the test for each of the subcomponents is presented separately. In addition, the implementation of a test optimization technique from [27] is presented.

5.2.1 Lookup Table - LUT mode

Implementation details of the LUT are proprietary and unavailable for this work. Structural testing comprises only of testing for SAFs on the inputs and outputs. However, the more comprehensive cell fault model is adopted for this component to ensure higher fault coverage by testing for all single/multiple internal combinational faults.

This dictates the application of an exhaustive set of test patterns on the inputs. There are six inputs for the Virtex-5 LUTs resulting in a total of $2^6 = 64$ different input combinations.

From a functional point-of-view, an LUT consists of memory elements in which the truth table values of an arbitrary 6-input function are stored. These truth table values are then selected by the function inputs using a large multiplexer thereby implementing that function. The stored truth table values are configuration inputs while the inputs of the function are data inputs. A conceptual illustration describing the function of a 2-input LUT is given in Fig. 5.8.

In LUT mode, the inputs are comprised of A_1, A_2 and the output A_{OUT}. These are used to implement any arbitrary 2-input function. In addition to testing that function exhaustively following the CFM, two complementary functions must be

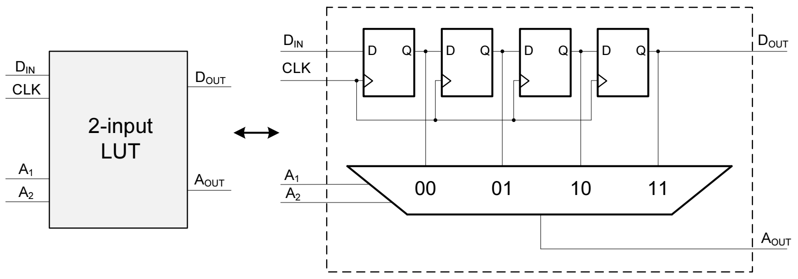

Figure 5.8: Functional view of a 2-input LUT

used to test for SAFs in the configuration bits. It would suffice to use the all 1's and all 0's functions for testing a single LUT. However, the XOR/XNOR configurations are used instead since they can be connected into C-testable arrays [13].

For an arbitrary number of inputs n, an LUT requires 2 configurations and 2^n test patterns. However, test pattern application is much faster than configuration time. A third configuration can be added such that the LUT is configured as XOR-XNOR-XOR to test additionally for all transition faults in the memory cells [13].

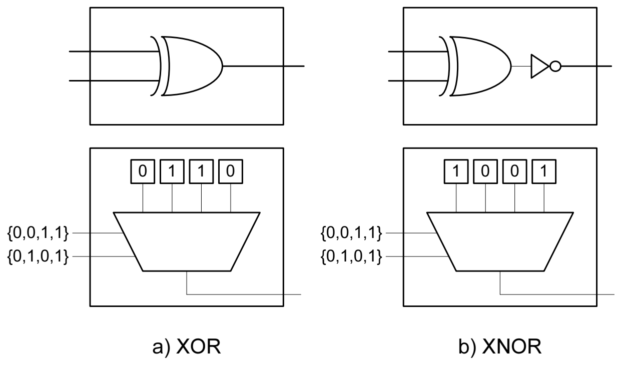

Figure 5.9: LUT testing configurations

5.2.2 Lookup Table - SR mode

In LUT mode, the multiplexer and its inputs/outputs are tested. In SR mode, the flip-flops are configured into a long shift register. According to the assumed fault model, transition faults are tested for in addition to the stuck-at faults. Well-known scan chain testing patterns [43] are used to account for the fault model. The "01100" test pattern is used because it contains the transition from $1 \rightarrow 0$ and vice versa. It also tests for a subset coupling faults.

The LUTs in SR mode are simply connected into multiple scan chains of which the outputs are compared together for response evaluation as before. To minimize test configurations, the flip-flop on each slice can be simultaneously tested in the same TC by interconnecting them between shift registers as depicted in Fig. 5.10.

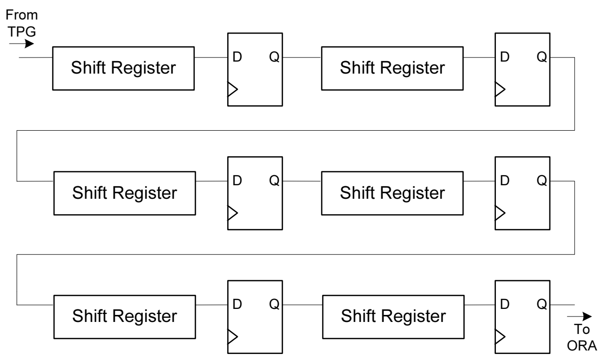

Figure 5.10: Interconnection of LUTs (SR mode) with flip-flops in arrays

This is an example of test multiplexing in which more than one CLB subcomponent is tested in the same TC. This is desirable to reduce the number of TCs as much as possible.

5.2.3 Lookup Table - RAM mode

RAM testing is quite mature. There exists march test algorithms for coverage of functional RAM faults as outlined in Chapter 4 [42, 2]. Depending on the march algorithm, there is a trade-off between test time/BIST overhead and coverage. The following table (adapted from [2]) outlines the various algorithms and coverage. (n is the size of the RAM)

Algorithm	Coverage								Cycles
	SAF	AF	TF	CF_{in}	CF_{id}	CF_{dyn}	SCF	LF	
MATS	All	Some	-	-	-	-	-	-	4n
MATS+	All	All	-	-	-	-	-	-	5n
MATS++	All	All	All	-	-	-	-	-	6n
MARCH X	All	All	All	All	-	-	-	-	6n
MARCH C-	All	All	All	All	All	All	All	-	10n
MARCH A	All	All	All	All	-	-	-	Some	15n
MARCH Y	All	All	All	All	-	-	-	Some	8n
MARCH B	All	All	All	All	-	-	-	Some	17n

Table 5.2: March tests coverage summary

Each LUT can implement a 64-bit RAM. Test patterns are generated at a global TPG implementing the MATS++ algorithm to ensure coverage of all SAFs, AFs and TFs. Note that only $5n$ operations are required instead of $6n$ because the initialization step can be specified in the TC bitstream.

Each slice contains four LUTs. Test response analysis is done by comparing the output of these four RAM components together, then all the ORA signals are collected using a long array of OR gates to the global ORA. This is possible because there are two types of slices (sliceM and sliceL). Only sliceM can implement RAM, so comparison and compaction can be interleaved and is done on the sliceL's in each CLB.

5.2.4 Multiplexer

Testing the multiplexer is simple. Exhaustive configurations are applied to test all select combinations and the data inputs/outputs are tested for SAFs by applying the 0 and 1 patterns. Multiplexer testing is always incorporated in other tests since it has to be used for routing the signals from the LUT or carry chain to the slice outputs.

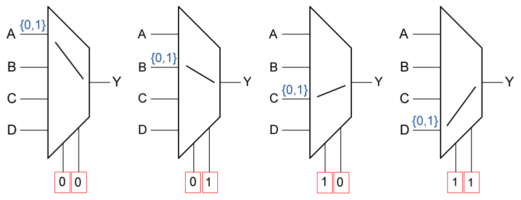

Figure 5.11: TCs for a 4 input multiplexer

The multiplexer test is identical to the LUT with the data and configuration inputs switched. For an n-input multiplexer: n configurations are required with two test patterns for each configuration.

5.2.5 Fast Carry Chain

Each slice contains a four-stage fast carry chain. It consists of static multiplexers and XOR cells. For both, the structural stuck-at fault model is assumed. The same test constraints apply, namely, interconnection in pipelined C-testable arrays.

Fig. 5.12 shows a two stage carry chain where the inputs are wither connected to "A" or "X". The figure shows that all test patterns for the XOR gate are reached using these configurations. Configuration into arrays is reduced to a simple parity tree.

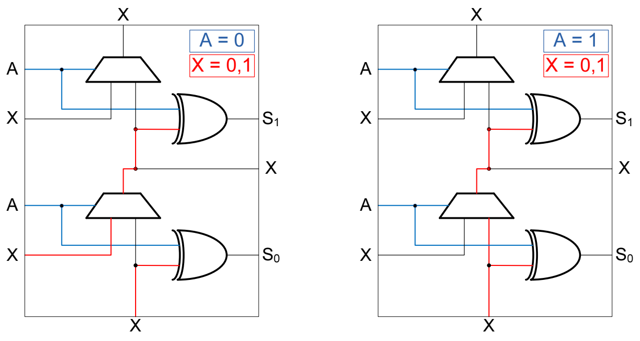

Figure 5.12: Two-stage carry chain under test

It remains to test the *carry-out* terminal of each slice. It has a dedicated interconnect to the *carry-in* terminal of the neighboring slice. This is done by connecting each column in a long carry chain and propagating the 0 and 1 values through it to test for both variants of SAFs. A flip-flop is used at the end of each column to pipeline the test as well.

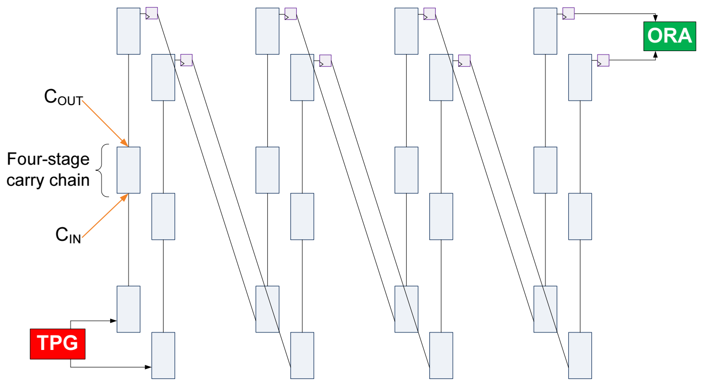

Figure 5.13: Pipelined test setup for the carry chain

5.2.6 Latches

Flip-flop testing is identical to testing the LUT in SR mode. However, the sequential elements in each slice can be additionally configured as level sensitive latches. A separate test is required to guarantee proper latch function. The test setup is shown in Fig. 5.14.

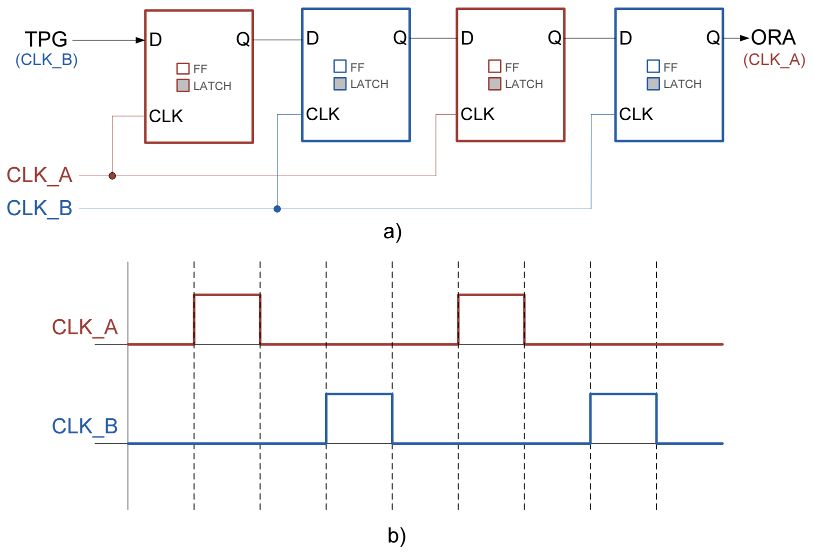

Figure 5.14: a) Scan chain of latches and b) two non-overlapping clocks for latch test

Since the function of the latches needs to be verified, two non-overlapping clocks are used as input to the scan chain. Odd-numbered latches use "CLK_A" while even-numbered latches use "CLK_B". This ensures that the function of the latches is correct in addition to testing for all SAFs and TFs. The same test pattern used for the flip-flops (01100) is also used for testing the latches [43].

Note that the opposite clocks are used in the TPG and first element to make sure that timing is satisfied between them. The same is done with the last element and the ORA.

5.3 Global CLB Test Optimization

The CLB test optimization technique [27] (Section 3.1.4) minimizes the number of TCs by deriving conditions for fault coverage and covering these conditions by a minimal number of TCs. It is demonstrated that it can automatically derive necessary configurations for a full-coverage test of a network of multiplexers. This can be extended for an entire CLB [27].

5.3.1 Generalization for CLBs

Necessary test configurations are derived for each component. For example, an LUT requires at least four different TCs (XOR, XNOR, SR, RAM) for full coverage of its operation modes. This can be encoded using two configuration bits as shown in Table 5.3.

Boolean Encoding	Test Configuration
00	XOR
01	XNOR
10	SR
11	RAM

Table 5.3: Boolean encoding of LUT TCs

The different LUT modes can then be modeled using a multiplexer as well. The same procedure is done to model the different modes of the sequential elements [27]. To prepare the circuit for deriving its boolean testability conditions, the following circuit model (Fig. 5.15) is used based on the mentioned conventions.

Note that the modes for the sequential elements use different multiplexers. This is because each of the configuration bits is disjoint with the other and they can be tested simultaneously. The paper [27] lacked an implementation for the optimization problem. A heuristic is introduced in the following subsection.

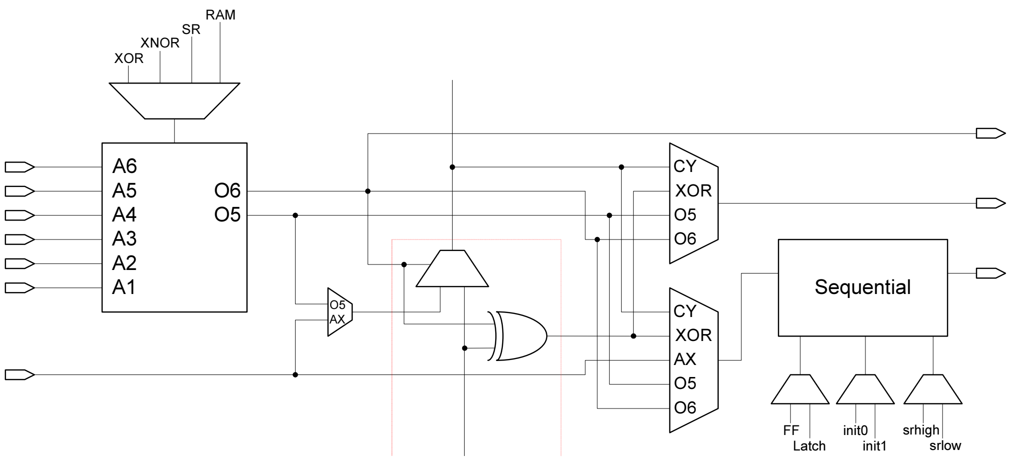

Figure 5.15: Circuit diagram with operational modes modeled as multiplexers

5.3.2 Set-Cover Heuristic

As previously shown in Fig. 3.11, the TC optimization problem translates into a set-cover problem. The goal is to find the minimum number of TCs which satisfy all testability conditions. This is an NP-complete problem which can be solved directly for small instances but requires a heuristic for practical instances to converge on the optimal solution.

The used algorithm is displayed in Listing 5.1. This is a modified greedy approach in which covered entries in the set are first removed. This gets rid of irrelevant entries and reduces the search space considerably. The remaining configurations are then filtered by systematically by removing each entry and checking if the set still covers all the testability conditions. Finally, this procedure is repeated multiple times with the set shuffled at the start of each new iteration. This is found to improve convergence on the optimal result since the algorithm depends on the ordering of elements. The number of iterations reflect the algorithm effort.

```
/* Number of iterations reflect the algorithm effort
 * In this case 25 iterations are used */
for(int iteration = 0;iteration<25;iteration++){

  /* Shuffle configSet at the start of each iteration to
   * increase likelyhood of finding the optimum results */
  Shuffle(configSet);

  /* Eliminate all configurations which are covered */
  for(int i = 0;i<configSet.size();i++)
    for(int j=0;j<configSet.size();)
```

```
      if(j!=i && areCovering(configSet.get(i),configSet.get(j))){
        configSet.remove(j);
        if(j<i) i--;
      } else j++;

  /* Refine results: Starting from the smallest weight,
   * remove the entries one-by-one and check if the
   * configSet still satisfys all conditions, else dont remove */
  for(int size=1;size<configSet.get(0).list.size();size++)
    for(int i=0;i<configSet.size();i++)
      if(getWeight(configSet.get(i).list)==size){
        ConfigEntry c = configSet.remove(i);
        if(!isSatisfyingSet(configSet))
          configSet.add(i, c);
        else i--;
      }

  /* Save minimum Set from current iteration
   * if it is better than previous solutions */
  if(configSet.size()<minList.size())
    minList=configSet.clone();
}
```

Listing 5.1: Pseudo-random set cover heuristic

The heuristic found the optimal solution on three different circuits and had a runtime in the order of seconds for larger circuits such as the Virtex-5 CLB. The results are considered satisfactory.

5.3.3 TC Optimization Shortcomings

Although the obtained configurations were optimal, they are only suitable for testing a single CLB. This conflicts with the general test approach followed in this thesis: array-testing methodology with pipelined stages.

The results could be directly used if it is possible to incorporate the C-testability/pipelining conditions in the boolean expressions for testability, this is left as an open issue for further research. Meanwhile, a very simple solution would be to place a multiplexer on the three outputs ensuring that only one of the outputs is used per TC, which in turn makes sure that the CLBs can be configured into an array.

Another shortcoming of the algorithm is the need to manually derive the boolean testability conditions for each component. This is a tedious, time-consuming and error-prone task. It would be very beneficial if these boolean expressions are automatically derived from the CLB netlist which is out of the scope of this work.

Chapter 6

Interconnect Test

Contents

FPGA interconnect architectures are becoming very complex. There are many different wire types varying in length, number of hops between connections, buffering and direction. In this work, a generic test infrastructure is developed to be able to test any of the wire types. Wires are sorted according to type in each TC for the systematic approach. An alternative implementation is the automatic approach in which the wires per TC are algorithmically selected and multiple wire types can be combined in one TC based on the same flexible BIST architecture. The critical parameter in interconnect test generation is the routing of the specific wires to be tested. Xilinx tools do not have the option of selecting which wires are used in routing so a routing algorithm is implemented for that task.

In this chapter, the BIST architecture is introduced before going into the details of the "local router" algorithm and implementation. The systematic method of wire selection is then presented in detail and the automatic approach is introduced.

6.1 Interconnect Test Architecture

As outlined in Chapter 2, Virtex-5 FPGAs have an island-style interconnect scheme consisting of at least six different interconnect types each in four directions. A simple and generic BIST architecture is devised to test any combination of wire types in the same test configuration. A more systematic approach is to test each wire type/direction combination in a separate TC.

6.1.1 Generic Test Architecture

An abstract view of the FPGA is shown in Fig. 6.1. Each PSM is shown with two slices connected, as it is the case for Virtex-5 FPGAs. Each of the slices is configured to a TPG or ORA as illustrated. This allows any wire types to be tested simultaneously or separately.

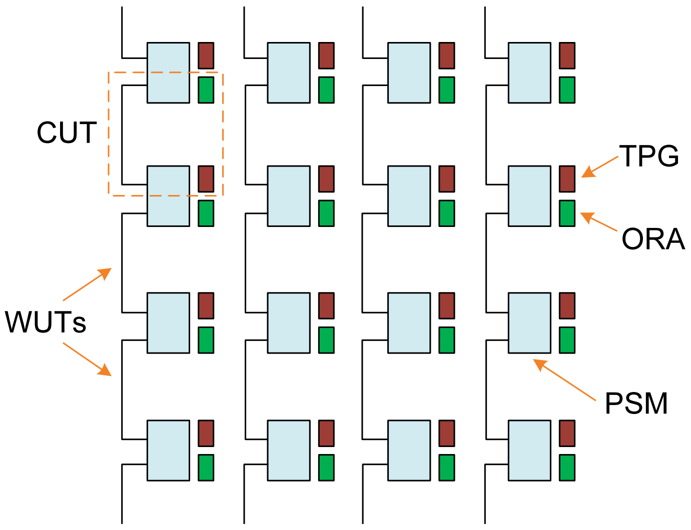

Figure 6.1: Interconnect test configuration

Fig. 6.1 shows a simple configuration of WUTs in which only single-hop, north wires are configured for test. The same BIST infrastructure can support any number or type of wires. The only constraint is the routability of these WUTs in the PSM associated with each TPG/ORA pair. This is also detailed in this chapter.

6.1.2 Test Response Compaction

The outlined architecture consists of multiple CUT setups, one for each group of WUTs. The test responses at the end of a test run are distributed over all the ORAs. The purpose of PRET is detection of the errors and not their diagnosis so all the ORA responses are compacted using a long OR array able to detect multiple errors from the CUTs. It is also possible to use a scan chain to serially read out the response data and localize fault location for diagnosis. However this is out of the scope of this work.

Only stuck-at ans stuck-open faults are being considered for interconnects. Stuck-open faults (SOF) occur when a wire is broken creating an open circuit while SAFs occur when a wire is shorted to a ground connection or VDD. No layout information is available for proprietary FPGAs so any test for bridging faults will be non-deterministic and does not guarantee good coverage of such faults. Inductive fault analysis tools, that infer possible bridging fault sites, operate on the layout of a circuit.

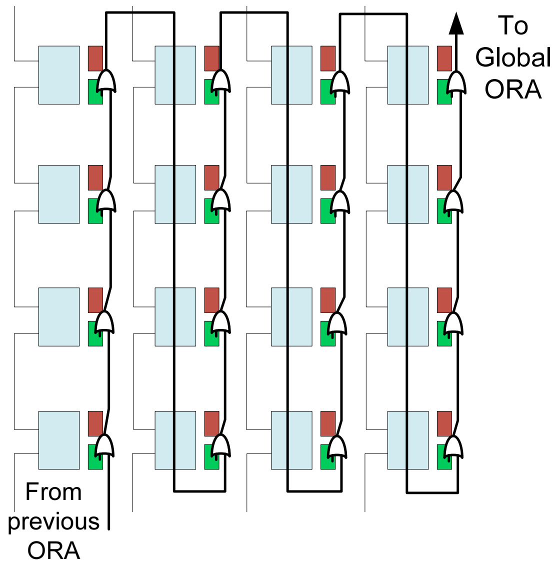

Figure 6.2: Interconnect response compaction

No CLBs outside of the container are being used for the test since all the TPG/ORA circuits are configured within the container itself. Container CLBs cannot be used by the system during test so there is no BIST hardware overhead from this test architecture.

6.1.3 Test Pattern Generator and Output Response Analyzer

For SAFs/SOFs, complete coverage can be obtained only by using the two test patterns 0 and 1. For that reason, all TPG nodes shown in Fig. 6.3 are identical. Each of these nodes generates both the 0 and 1 patterns. Additionally the three nodes can be used to generate an exhaustive set of test patterns (from 000 to 111) to test for bridging faults. Because such a test is not based on layout information, it does not guarantee coverage of realistic bridging faults.

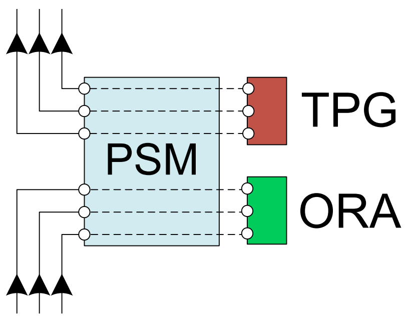

Figure 6.3: One PSM in an interconnect test configuration

The ORA is similar to the comparison-based ORA used in the CLB logic test illustrated in Fig. 5.3. Because all the TPG nodes are identical and synchronous, the same ORA can be used to compare any group of wires regardless of the type.

6.2 Local Router

The main challenge and limitation in creating an interconnect test template is routing the specific WUTs to the TPG and ORA. The PSM has a finite number of programmable resources used to route the connections between the TPG/wire beginnings and ORA/wire ends. These programmable resources are called programmable interconnect points (PIPs).

Xilinx tools do not give the option of selecting wires to be routed in a design, making it impossible to know which wires are being tested. This additional control is crucial in creating interconnect test templates. Xilinx provide, however, a low-level design language called XDL in which each net routing is specified.

A router is created to route the connections between the TPG $\rightarrow$ wire beginnings and ORA $\leftarrow$ wire ends. This "local router" operates on a single PSM according to the BIST architecture introduced and the resulting routing can be replicated for all other PSMs in a container.

6.2.1 Routing Algorithm

The goal is to make valid connections from the TPG $\rightarrow$ wire beginnings and ORA $\leftarrow$ wire ends and avoid routing conflicts. To represent the various nodes (TPG/ORA/wires), arrays are used. The arrays can have arbitrary sizes and are sorted into pairs of arrays. There are four arrays: TPG/ORA/WIRE_BEG/WIRE_END. Each of these arrays contains the respective set of nodes. The TPG/WIRE_BEG arrays are in one groups since their nodes are required to be connected together and similarly the ORA/WIRE_END arrays are in another group. Fig. 6.4 shows a representation of such arrays and their corresponding nodes in the CLB and PSM.

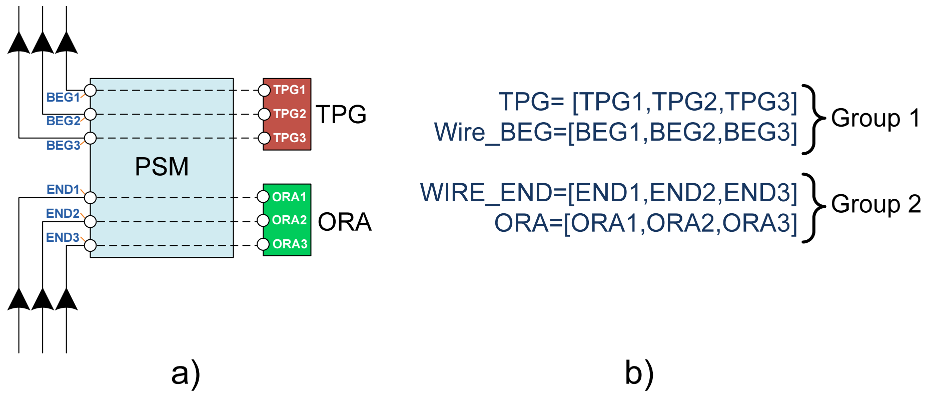

Figure 6.4: a) Interconnect test setup and b) array representation of nodes

These arrays are input to the routing algorithm which searches a graph representation of the PSM PIPs for a valid route between nodes.

Group 1 is first taken into account: all the paths are found for the first pair of nodes, for example, between TPG1 → BEG1. The algorithm then marks the used PIPs and moves on to the next two nodes in the first group: TPG2 → BEG2. All possible paths are found and routing each path is checked for conflicts or if it is valid. If a conflict exists, all possible paths and TPG2 → BEG2 cannot be routed in the current configuration. The nodes are reordered into the next permutation and all PIPs are again marked as unused. This time the algorithm will attempt to route TPG1 → BEG2 for example as shown in Fig. 6.5.

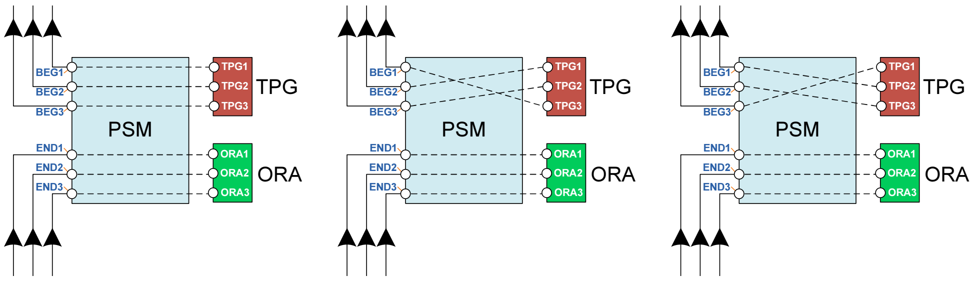

Figure 6.5: Three different permutations when routing group 1

To increase the likelihood of routing, the pairs of nodes are first sorted in ascending order from the lowest number of paths to the highest. This allows the node pairs with the weakest connections to be routed first and is shown to greatly increase routability of a set of nodes as well as to improve on the routing speed.

For example, routing of the first permutation in group 1 is being performed and the number of different paths for each pair is as it is listed in Table 6.1. To increase likelihood of success, routing will be attempted in the following order: TPG2 → BEG2 then TPG3 → BEG3 and finally TPG1 → BEG1. This is equivalent to sorting them in ascending order of the number of paths.

Node Pairs	TPG1 → BEG1	TPG2 → BEG2	TPG3 → BEG3
# Paths	30	5	17

Table 6.1: Number of paths for node pairs from group 1

After successfully routing group 1, the next group is routed in the same way. If routing fails, reordering of the groups is done (by trying all permutations) and routing is attempted again as outlined. Note that the router can have an arbitrary number of groups and nodes within each group.

To find all the possible paths between nodes, a depth-first graph traversal algorithm (DFS) is implemented using recursion. The problem of finding all possible paths between two nodes in a graph is a superset of the problem of finding the longest path between two nodes. Both algorithms are NP-complete and their respective algorithms have non-polynomial complexity. However, the PSM has a limited search space, and DFS algorithm for finding all paths runs very quickly (in the order of milliseconds). The algorithm is shown in Listing 6.1.

```java
public static void findPaths(node from, node to, Tile fromTile,
      Tile toTile, LinkedList<node> path, HashSet<Integer> usedSet){

    if(from.id == to.id){
      /* Reached destination, append to list of paths */
      pathsList.add(path);
    }
    else
    {
      /* Return all connections from this node */
      WireConnection[] wireConns = fromTile.getWireConns(from.id);

      if(wireConnections != null)
      for(WireConnection w : wireConns){

        /* Find current tile */
        Tile currTile = dev.getTile(from.getRow()-w.getRowOffset(),
                    from.getColumn()-w.getColumnOffset());

        /* Check that node was not traversed before
         * and that the connection is being made in the same PSM
         * and mark current node as traversed */
        if(((currTile == fromTile)
            || (currTile == toTile))
            && usedSet.add(w.getWire()) == true)
        {
          /* Create current node from current tile */
          node curr = new node(from, w.getWire(), currTile);
          /* Depth first graph traversal
           * Add the current node to the current path */
          path.add(curr);
          /* Recursion: traverse all children of current node */
          findPaths(curr, to, currTile, toTile, path, usedSet);
          /* Remove current from path and unmark as traversed */
          path.remove(curr);
          usedSet.remove(w.getWire());
        }
      }
    }
  }
```

Listing 6.1: Recursion to find all possible paths between two nodes

The term “routing conflicts” was mentioned but not defined so far. A routing conflict occurs if a wire segment is being used by two different paths, or if a PIP node is being driven by two different signals. Note that the same PIP can drive two nodes but not the opposite.

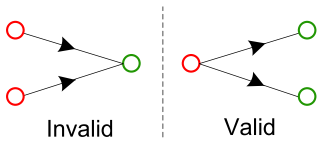

Figure 6.6: Illustration of routing conflicts and allowed fanouts

Because all TPG nodes are identical, a single TPG node can drive multiple WUTs. This option also improves on the number of WUTs per TC and provides added flexibility to the router. A screen shot of routing three north PENT wires and five south PENT wires is shown in Fig. 6.7.

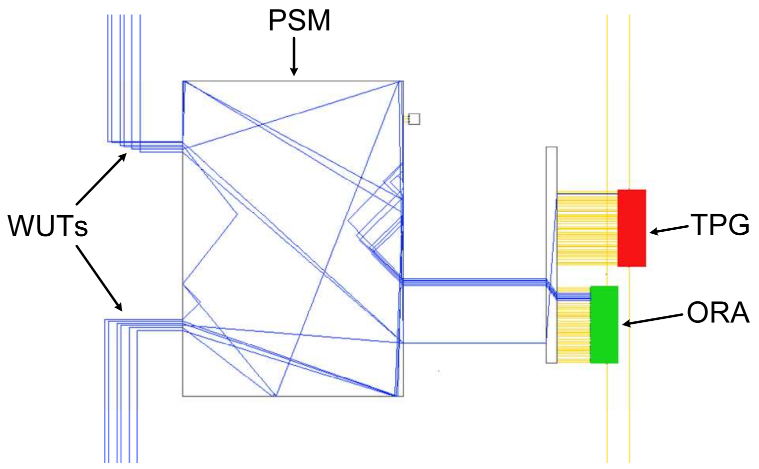

Figure 6.7: Screen shot of WUT routing (taken from FPGA Editor)

6.3 WUTs Selection

The local router and presented BIST architecture are used at the core of interconnect test generation in both wire selection methodologies presented here. Using these basics, different interconnect test approaches can be realized. The choice determines which wires are tested in each TC.

6.3.1 Systematic WUTs Selection

Wires are classified into different types based on their direction, number of connections and number of hops[1]. In the systematic approach, each wire classification is

[1]Wire classifications and naming conventions are explained in Chapter 2 and Appendix B

handled in a separate TC [4]. The number of TC per wire classification depends on the limitations in PSM routing. Furthermore, multiple wire classifications can be tested in the same TC based on the presented BIST infrastructure.

This main advantage of this *divide and conquer* systematic approach is the ease of designing and implementing the test templates. Fig. 6.8 shows a specific systematic test for double lines in the east direction.

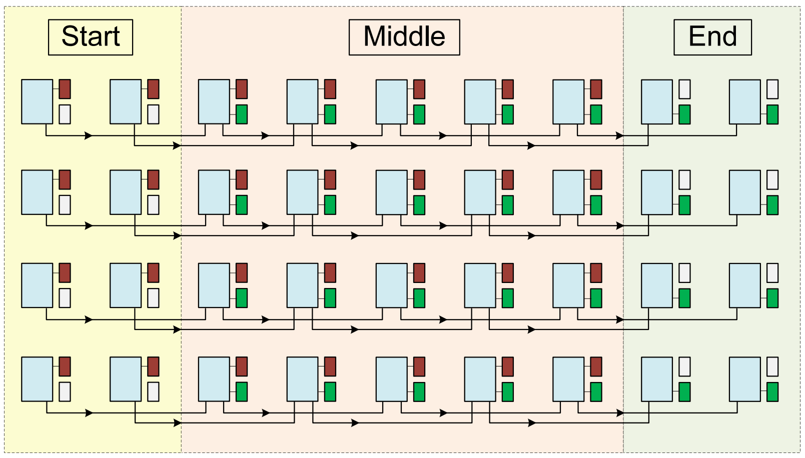

Figure 6.8: Systematic test for double east wires

Routing in the PSM is separated into three parts: start, middle and end. In the group at the start, only TPG → WIRE_BEG connections are required to be made. In the end group only WIRE_END → ORA connections are necessary. The middle group combines the connections in both the start and end groups. For double lines, start and end portions are always two PSMs wide. For pent lines, these portions have a width of five PSMs.

To determine whether a set of wires is routable in one TC, the middle group of PSMs are critical. If the PSMs in the middle group have enough resources to route the WUTs then this interconnect test template would be possible.

6.3.2 Automatic WUTs Selection

The flexible BIST architecture introduced in Section 6.1.1 supports simultaneous test of multiple wire types. This added flexibility is expected to improve on the number of TCs since it adds an additional degree of freedom to the routing algorithm. However this makes the implementation of the test templates more complicated because there are many more variables to be taken care of.

Similar to the systematic approach, wire selection is done based on a single PSM and replicated for the entire container based on the start/middle/end definitions for each wire. To check whether a group of wires are routable, the PSM with the intersection of all their middle portions must be routable. This is because it contains the highest number of connections and its connections are a superset of the other PSMs.

Heuristics can be used to sort the wires under test in a minimized number of TCs based on their routability. The idea is to write an application that utilizes the local router. This heuristic would attempt to maximize the number of WUTs per TC thereby leading to a minimum (or near-minimum) set of TCs. This part is out of the scope of this thesis and is left for future research.

Chapter 7

Implementation and Results

Contents

Low level access to the FPGA circuitry and configuration is required for the implementation of a structural test. This information is hidden from designers because it is proprietary and protected by FPGA vendors. Not to mention that the FPGA tool flow supports mainly high-level design entry to facilitate complex designs and not low level access to each structure. A suitable implementation platform achieves this much needed low level access to the FPGA for test template implementation.

This chapter contains both the implementation details and obtained results. The implementation is presented with details of the project framework and design languages used. The results of the implemented tests are also discussed with respect to test speed, BIST overhead and coverage.

7.1 Design Tools

7.1.1 Xilinx Design Language

Xilinx tools are designed to translate a hardware description language (HDL), such as VHDL or Verilog, to a valid configuration bitstream for the FPGA. The typical tool-flow is illustrated in Fig. 7.1, shown in blue.

Xilinx also provides access to the details of logic and interconnect configuration without revealing any hardware information or bitstream encoding. This is done through XDL [44]. Configuration for each component is provided in detail such that each subcomponent can be configured in any of its possible modes of operation. Exact placement and routing on the FPGA can also be specified for each instance.

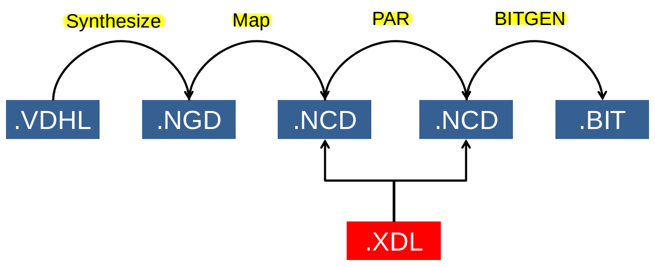

Figure 7.1: Xilinx design cycle with XDL

Fig. 7.1 shows that XDL can be directly translated into a mapped or routed netlist circuit description (NCD) file. This also brings the advantage of fast design time because synthesis and mapping steps are not performed. Appendix A shows code snippets of a slice configuration written in XDL highlighting its low level configuration.

XDL files are used for design entry of the tests, but the FPGA structure itself is defined in a set of text files called XDLRC. In these files, which are approximately 10 GB in size, each FPGA component is defined with its subcomponents, input/output ports as well as the interconnection between them all.

Xilinx provides XDL and XDLRC so that it is possible for users and designers to implement design tools. The main difficulty is that these languages are not documented. However, XDL and XDLRC are human-readable and can be parsed to extract information about the FPGA that is not possible in any other way.

7.1.2 RapidSmith Java Framework

The first step in implementing tests for the FPGA is to understand the XDLRC FPGA description and parse it into a usable data structure that can be used in designing tests. This has already been done in an open-source research tool called RapidSmith [45]. This framework is implemented in Java.

In RapidSimth, the FPGA is defined into "tiles" each containing at most one CLB and one PSM. There are functions to return the components in each tile. It is also possible to find neighboring tiles and investigate which wires connect two tiles together. In addition there are descriptions of PSMs including PIP connections and wire beginnings and ends. This information is parsed from Virtex-5 XDLRC files and are sufficient in creating low level tests.

7.2 CLB Testing

For testing the FPGA logic, a series of TCs are first generated from test templates. This is based on the RapidSmith Java framework and is also implemented in Java. The output is a set of XDL files. A script uses the Xilinx tools to translate the XDL to NCD, then routes the design and generates the corresponding bitstreams.

7.2.1 CLB PRET Tool Flow

The detailed tool-flow is shown in Fig. 7.2. First, the logic subcomponents are selected for test, an exhaustive test can also be chosen covering all TCs. The next step is to select the container size and this is done by specifying the coordinates at the lower-left and upper-right coordinates of a rectangular container on the FPGA. The coordinates correspond to CLB numbers. The tool uses this information to return the valid sites in the container for each specific test.

The PRET tool then generates the TPG and ORA which are independent of the container size because of the C-testability condition imposed on all test arrays. All container sizes require the same number of test patterns. The container setup is then created and finally placed in the valid container sites.

At this point, the nets are defined for the connections between TPG, ORA and the container. However, routing is done later using the Xilinx PAR tool. Unlike interconnect tests, routing does not contribute to the actual TC because the logic itself is tested and not the extra-CLB wires. Bitstream generation is also done using a Xilinx tool (BITGEN) because bitstream encoding is proprietary and can only be done through the provided tools.

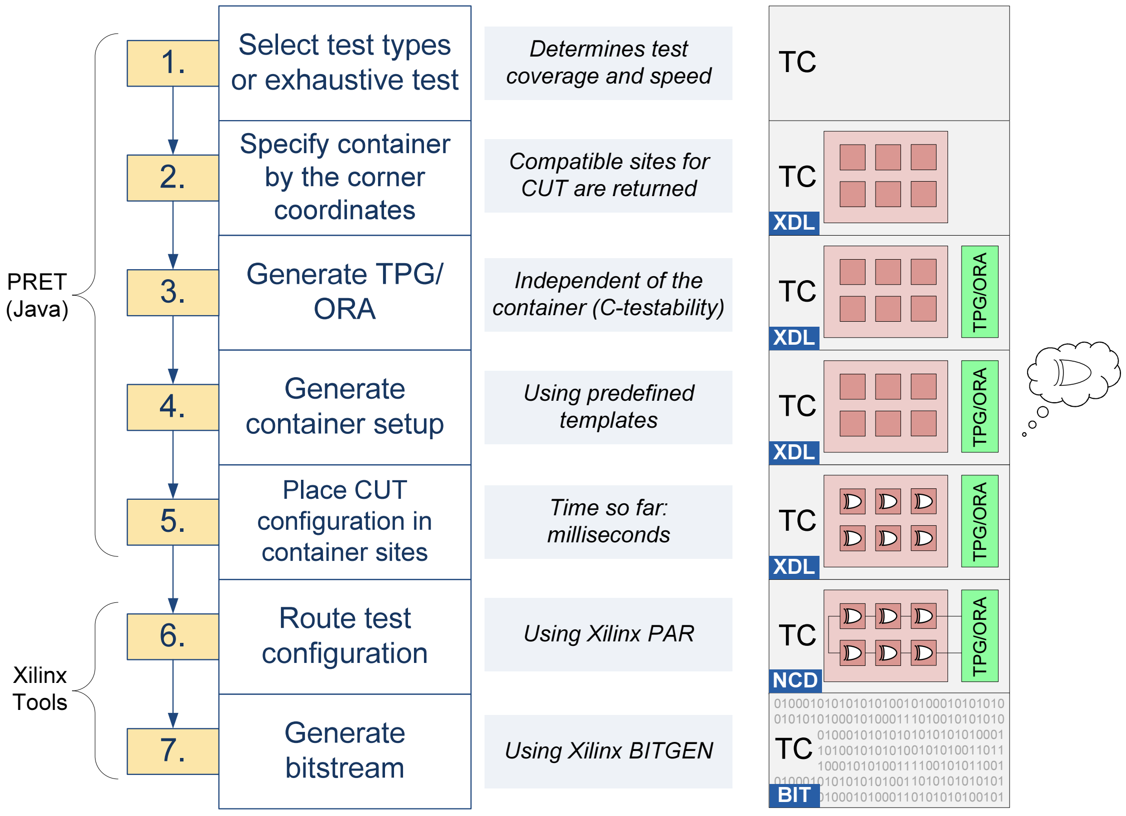
PRET (Java)
Xilinx Tools
1. Select test types or exhaustive test
2. Specify container by the corner coordinates
3. Generate TPG/ ORA
4. Generate container setup
5. Place CUT configuration in container sites
6. Route test configuration
7. Generate bitstream
Determines test coverage and speed
Compatible sites for CUT are returned
Independent of the container (C-testability)
Using predefined templates
Time so far: milliseconds
Using Xilinx PAR
Using Xilinx BITGEN
TC
XDL
TPG/ORA
NCD
BIT

Figure 7.2: CLB test implementation flow

7.2.2 CLB Test Results

Nine TCs are sufficient for full coverage detection of the structural and functional faults assumed for the CLBs. High test clock frequency is demonstrated as a result of test pipelining. Furthermore, very low hardware overhead for BIST is achieved because the array testing method is used.

7.2.2.1 CLB Results Summary

The test strategies introduced in Chapter 5 ensure a high test clock frequency because all tests are pipelined. In addition, a low BIST overhead is achieved because of the array testing method that uses just one TPG and a simple comparator-based ORA. Table 7.1 lists the nine TCs. A brief description of each configuration is stated along with the BIST overhead (in number of CLBs) and the operating test frequency.

Test frequency ranges between 154 MHz and 225 MHz. The slower tests are the ones in which flip-flop interleaving is employed. The critical path length becomes two CLBs instead of one CLB in fully pipelined designs. This accounts for the difference in test speed.

The BIST overhead is very low (one or two CLBs) for all tests except for the RAM test. This is because all tests use simple counter-based TPGs while the RAM test requires a more complex MATS+ test controller.

Table 7.2 shows the configuration of each subcomponent in the TCs. These extra details are required to highlight that each subcomponent is tested in each possible mode of operation. More details can be extracted from the actual implementation files. When the LUT is configured as a *helper* this means that it may be used to implement a transparent or inverting function, or generates specific values from its two outputs to assist testing of other subcomponents.

TC	Description of CUT	$\sim$ BIST Overhead	$\sim$ CLK Frequency
1	LUT configured as XOR, connected to FF	2 CLBs	207 MHz
2	LUT configured as XNOR, connected to FF	2 CLBs	207 MHz
3	Carry MUX, interleaved with MUX and latch	1 CLBs	182 MHz
4	Carry MUX, interleaved with MUX and latch	1 CLBs	164 MHz
5	Carry XOR, interleaved with MUX and FF	1 CLBs	182 MHz
6	Carry XOR, interleaved with MUX and FF	1 CLBs	164 MHz
7	Carry-in/-out tested, multiplexed scan chain	1 CLBs	150 MHz
8	LUT configured in SR mode with slice MUX	1 CLBs	157 MHz
9	LUT configured in RAM mode with slice output	7 CLBs	195 MHz

Table 7.1: Description of CLB TCs, BIST overhead and CLK frequency

TC	LUT	Flip-flop				OUT MUX	FF MUX	CARRY MUX	OUT USED	C_{IN}/C_{OUT} USED	CLK
		Mode	INIT	SR	RST						
1	XOR	FF	0	low	async	-	O_6	-	-	-	clk
2	XNOR	FF	1	high	async	-	O_6	-	-	-	clkinv
3	helper	Latch	0	high	sync	C_Y	O_5	AX	-	-	clk
4	helper	Latch	1	low	async	O_5	C_Y	AX	-	-	clk
5	helper	FF	1	high	sync	-	XOR	O_5	-	-	clk
6	helper	FF	0	low	sync	XOR	-	O_5	-	-	clk
7	helper	FF	1	low	sync	-	AX	-	-	yes	clk
8	SR	-	-	-	-	O_6	-	-	-	-	clk
9	RAM	-	-	-	-	-	-	-	yes	-	clk

Table 7.2: March tests configuration summary

These results excludes certain advanced LUT multiplexing features available in Virtex-5 FPGAs. Two, three or four LUTs are combined using two-input multiplexers to form larger LUTs. These multiplexers can be tested with an estimated additional two or three TCs.

7.2.2.2 CLB Test Timing Analysis

CLB tests boast a high test clock frequency because they are pipelined. However, the clock frequency drops with increasing container size. Timing reports are generated for different container sizes with both horizontal and vertical scaling of the containers. The graphs in Fig. 7.3 and Fig. 7.4 illustrate these results.

The tests analyzed in Fig. 7.3 use container sizes of ($columns \times rows$) 5×10, 5×20, 5×40, 5×80. This is to investigate vertical scaling of containers. It is shown for the four plotted TCs that the test frequency drops with increasing container size. This is expected because capacitive loading on the TPG signals increase. In addition, clock routing becomes more complex. TC7 is the test of C_{IN}/C_{OUT} in which the pipeline length increases with number of rows; it is therefore most affected by vertical scaling.

In Fig. 7.4, container sizes are 5×10, 10×10, 20×10, 40×10. TC1 and TC9 show the expected behavior in which the clock frequency drops with increasing container size. However the drop is lower than the one calculated for vertical scaling. TC7 clock frequency stays constant because the number of rows (and therefore the pipeline length) remain constant. The results for TC5 were unexpected because there is a slight increase of frequency at the start. This can be attributed to variations in the internal Xilinx tools.

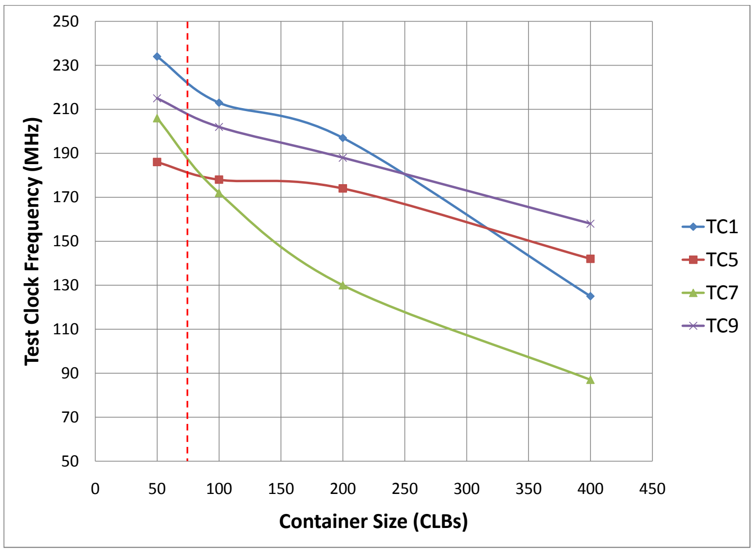

Figure 7.3: Effect of vertical scaling of container size on the test clock frequency

The red dashed line marks the size of the reconfigurable container already being

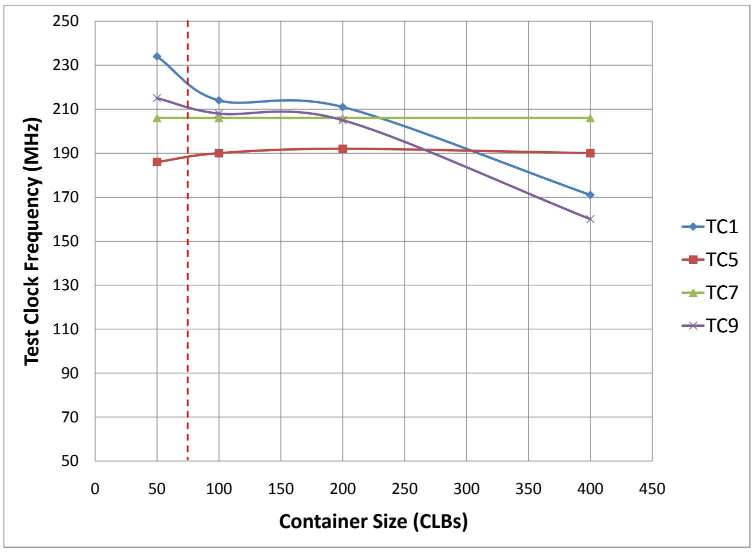

Figure 7.4: Effect of horizontal scaling of container size on the test clock frequency

used in previous works [1]. Test frequencies between 180 MHz and 220 MHz are used for testing this container size.

7.2.2.3 Container Screen Shot

Fig. 7.5 shows a snapshot of CLB TC1 obtained from FPGA Editor [46]. The TPG and ORA as well as the container are marked on the figure. This container is placed with start coordinates CLB_X20Y100 and end coordinates CLB_X40Y140. In total it contains 800 CLBs. The TPG and ORA are placed in the corner of the FPGA as illustrated, they occupy just two CLBs.

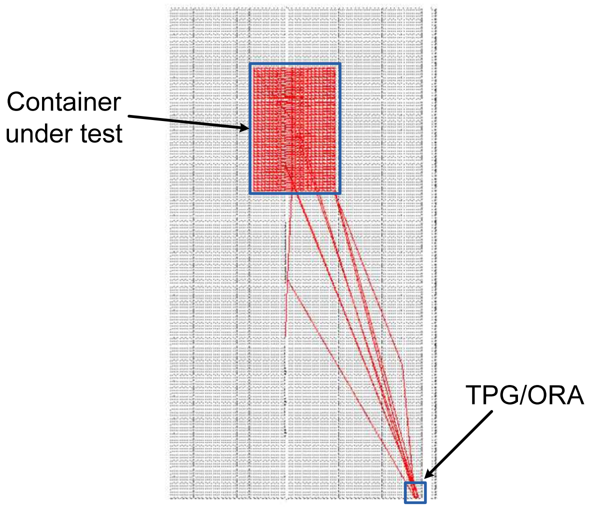

Figure 7.5: Screen shot of a container under test (taken from FPGA editor)

7.3 Interconnect Testing

Generating interconnect XDL templates is very similar to CLB tests. The main differences are in configuring the TC routing. The local router separately routes the WUTs before using Xilinx tools for routing the BIST infrastructure signals.

7.3.1 Interconnect PRET Tool Flow

Fig. 7.6 shows the detailed tool-flow for interconnect TC generation. There are two main differences to the CLB test tool-flow concerning the routing of WUTs and configuring the TPG and ORA.

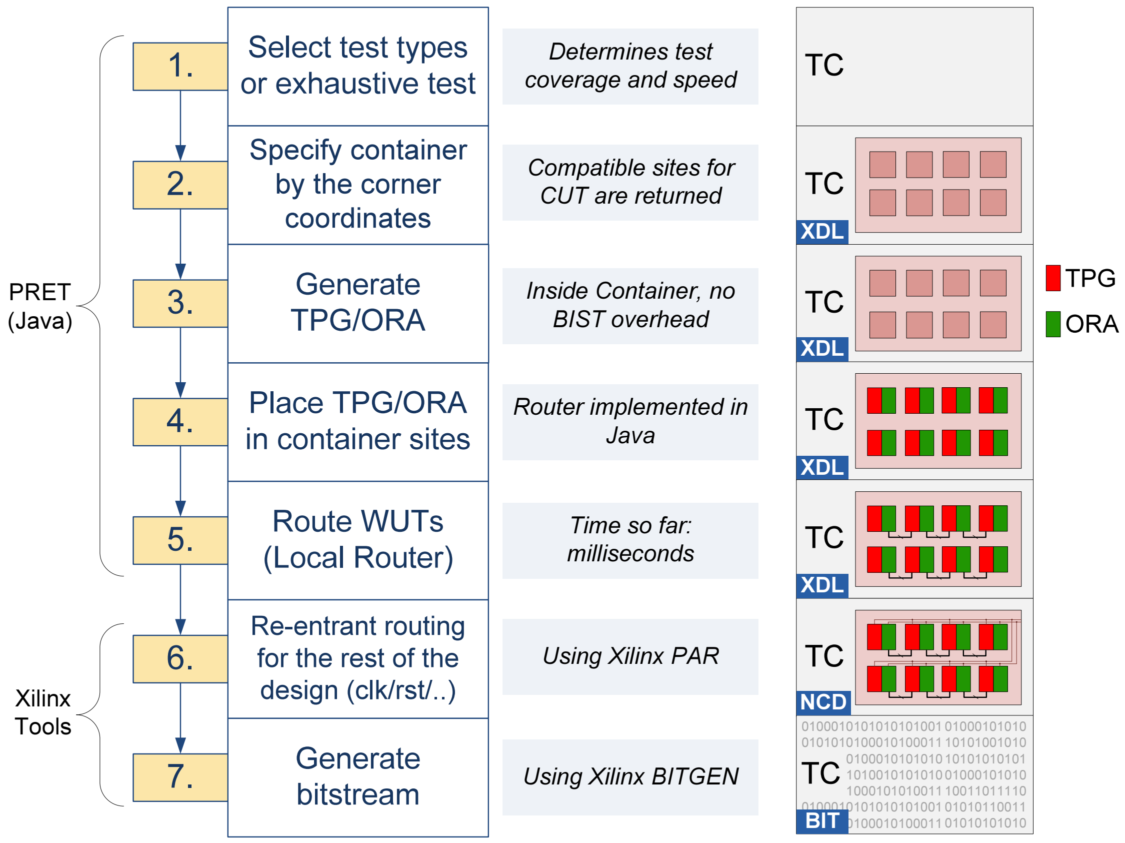
PRET
(Java)
Xilinx
Tools
1.
Select test types or exhaustive test
Determines test coverage and speed
2.
Specify container by the corner coordinates
Compatible sites for CUT are returned
3.
Generate TPG/ORA
Inside Container, no BIST overhead
4.
Place TPG/ORA in container sites
Router implemented in Java
5.
Route WUTs (Local Router)
Time so far: milliseconds
6.
Re-entrant routing for the rest of the design (clk/rst/..)
Using Xilinx PAR
7.
Generate bitstream
Using Xilinx BITGEN
TC
XDL
NCD
BIT
TPG
ORA

Figure 7.6: Interconnect test implementation flow

The first difference is that there is no additional BIST overhead because the TPG/ORA are already configured inside each container. This makes up the container setup for interconnect tests. Secondly, the routing step is split into two parts, shown as steps 5 and 6 in Fig. 7.6.

After configuring the TPG and ORA, the WUTs are routed in step 5 using the local router (presented in Section 6.2). Next, *re-entrant routing* using the Xilinx

PAR tool is performed. This leaves the WUTs routing unchanged and routes the infrastructure signals (e.g. CLK and RST) which do not contribute to wire-testing.

7.3.2 Interconnect Test Results

Interconnect testing is performed with systematic wire selection. It is implemented for north/south PENT lines as a proof of concept. Three north PENT wires and 4 south PENT wires are multiplexed onto the same test configuration. A snapshot is shown in Fig. 7.7.

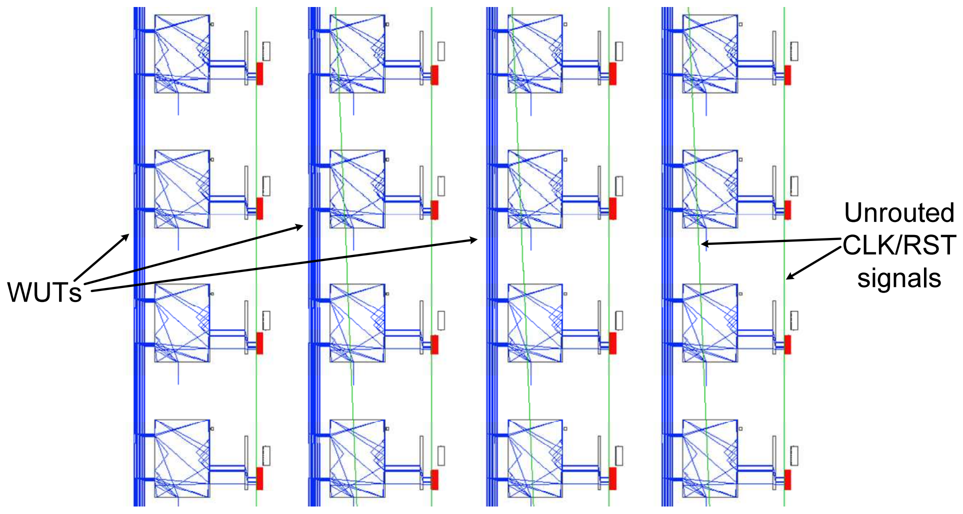

Figure 7.7: Screen shot of a container under test (taken from FPGA editor)

It is clear that the routed WUTs are very regular and in the vertical direction. This is the result of the local router algorithm running on each PSM to configure the north/south WUTs. Finally, the Xilinx PAR tool is used to route the CLK, RST and other infrastructure signals.

There are a total of 6 pent wires in each direction, this TC tests covers 58.3 % of the north and south pent lines. The test clock frequency is simulated at above 800 MHz for this test configuration. Other wires can be tested in the same procedure to provide full coverage for interconnect wires.

Chapter 8

Conclusion

Contents

8.1 Summary and Main Contributions

In the context of dynamically reconfigurable architectures, FPGA testing was researched, developed and implemented. A runtime reconfigurable system is able to reconfigure parts of the FPGA as HW accelerators during runtime. To assure reliable operation a PRET is scheduled to test the bare FPGA fabric each time before configuration.

Literature in the field of FPGA testing was thoroughly reviewed at first. Knowledge from the research, analysis and comparison of state-of-the-art FPGA test techniques was gathered to determine an optimal set of tests that provide full coverage testing of FPGA faults. The fault model itself was tailored to provide the most meaningful and extensive coverage possible. The structural stuck-at fault model was extended by a number of component specific functional fault models for RAM and sequential elements. The cell fault model was assumed for LUTs. The result was a hybrid fault model covering both structural and functional faults.

Array testing methods were used for CLB testing as it was found to have the lowest hardware overhead and fastest run time among FPGA testing techniques. An additional constraint was imposed on the test configurations to allow a high test clock frequency: all tests were pipelined.

The developed test framework was implemented in Java with the help of RapidSmith; an open-source tool which parsed FPGA descriptions into a usable data structure. Nine TCs were sufficient for a full-coverage CLB test (excluding some LUT multiplexing features). The TCs demonstrated a high test clock frequency above 170 MHz and was shown to run in hardware on Virtex-5 FPGAs. In addition, a test optimization technique was implemented using a pseudo-random set-cover heuristic shown to give optimal results for a Virtex-5 logic slice.

Interconnect testing was also considered. A flexible BIST architecture was designed for testing any number or type of wires depending on routing constraints. A routing algorithm (local router) was developed and operated on a single PSM for routing the connections between TPG → WUT-beginnings and ORA ← WUT-ends. A north/south pent-wire test template using the local router was implemented as a proof of concept.

All the implemented tests were parameterizable so that they can be created for an arbitrary container size and location on the FPGA. Furthermore, the BIST hardware overhead was very low: ranging between just 1 and 7 CLBs for all the tests.

8.2 Future Work

Throughout this thesis, regularity of the FPGA structure was extensively exploited. The implementation flow was highly regular as well and there is a high potential for automating the process of FPGA testing.

Fig. 8.1 shows a rough overview of how FPGA testing could be completely automated based on concepts used or developed during this thesis. A possible CAD tool could extract testability conditions from a CLB circuit netlist. The testability conditions would then be minimized using the TC minimization algorithm presented in Section 5.3. This step will require further research to be able to create TCs configurable into pipelined arrays. The output of step 2 is a set of abstract descriptions of the TCs which would be used to generate the actual circuit TC netlists and bitstreams using a test template generation tool.

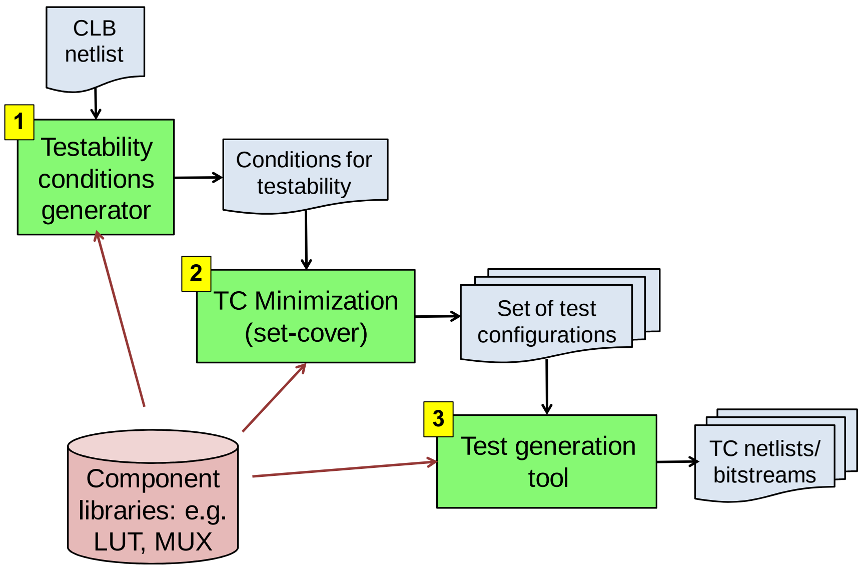
CLB netlist
1
Testability conditions generator
Conditions for testability
2
TC Minimization (set-cover)
Set of test configurations
3
Test generation tool
TC netlists/ bitstreams
Component libraries: e.g. LUT, MUX

Figure 8.1: Block diagram of possible FPGA CAD test software

Steps 2 and 3 are already implemented within this thesis and it remains to link them together. Step 1 still requires investigation. A CAD tool should be extensible with component libraries that would include a description for logic subcomponents and their interconnections together in a single slice.

More immediate future work involves fault simulation to assess the coverage of each TC accurately and quantify the implementation results. It is also worthwhile to investigate the various advanced features in Virtex FPGAs of partial reconfiguration and memory readback as they can optimize the critical FPGA test reconfiguration time.

Appendix A

XDL Syntax

Listing A.1 demonstrates a small XDL code example that shows a logic slice placed in CLB “CLBLM_X24Y5” and more specifically in sliceL “SLICE_X41Y5”. The configuration string specifies the slice programming in full detail. A routed net is also shown. The code is annotated with comments explaining basics about XDL syntax

```
# The syntax for instances is:
# instance <name> <sitedef>, placed <tile> <site>, cfg <string> ;
# or
# instance <name> <sitedef>, unplaced, cfg <string> ;

inst "slice_example" "SLICEL",placed CLBLM_X24Y5 SLICE_X41Y5 ,
  cfg " A5LUT:CY5A_rt.1:#LUT:O5=A5 A6LUT:CY6A_rt:#LUT:O6=~A6
        ACY0::AX AFF:unit401.AFF:#LATCH AFFINIT::INIT0
        AFFMUX::CY AFFSR::SRLOW AOUTMUX::O5 AUSED::#OFF
        B5LUT:CY5B_rt.1:#LUT:O5=A5 B6LUT:CY6B_rt:#LUT:O6=~A6
        BCY0::BX BFF:unit401.BFF:#LATCH BFFINIT::INIT0
        BFFMUX::CY BFFSR::SRLOW BOUTMUX::O5 BUSED::#OFF
        C5LUT:CY5C_rt.1:#LUT:O5=A4 C6LUT:CY6C_rt:#LUT:O6=~A6
        CCY0::CX CEUSED::#OFF CFF:unit401.CFF:#LATCH
        CFFINIT::INIT0 CFFMUX::CY CFFSR::SRLOW CLKINV::CLK
        COUTMUX::O5 COUTUSED::#OFF CUSED::#OFF D5LUT::#LUT:O5=A4
        D6LUT:CY6D_rt:#LUT:O6=~A6 DCY0::DX DFF:unit401.DFF:#LATCH
        DFFINIT::INIT0 DFFMUX::CY DFFSR::SRLOW DOUTMUX::O5
        DUSED::#OFF PRECYINIT::#OFF REVUSED::#OFF SRUSED::0
        SYNC_ATTR::ASYNC"
        ;

# The syntax for nets is:
# net <name> <type>,
# outpin <inst_name> <inst_pin>,
# inpin <inst_name> <inst_pin>,
# pip <tile> <wire0> <dir> <wire1> , # [<rt>]
# ;
#
# The <dir> token will be one of the following:
```

```
#
# Symbol Description
# ====== ==========================================
# == Bidirectional, unbuffered.
# => Bidirectional, buffered in one direction.
# =- Bidirectional, buffered in both directions.
# -> Directional, buffered.
#
# No pips exist for unrouted nets.

net "example_net_name" ,
  outpin "unit1" AQ ,
  inpin "unit2" A3 ,
  pip CLBLL_X34Y9 SITE_IMUX_B27 -> M_A3 ,
  pip CLBLM_X33Y9 L_AQ -> SITE_LOGIC_OUTS0 ,
  pip INT_X33Y9 LOGIC_OUTS0 -> ER2BEG1 ,
  pip INT_X34Y9 ER2MID1 -> IMUX_B27 ,
  ;
```

Listing A.1: XDL code example

Appendix B

Virtex-5 Interconnects

B.1 Pin Naming Conventions

Pin names are extensively used in XDL to represent a source or sink of an interconnect wire. The encoded information contains the pin position as well as wire classification. That is, the pin name consists of the wire type, direction and which terminal it is (for example, start, middle or end). In Virtex-5 FPGAs this applies to the wire types pent and double.

The pin name consists of seven characters. The first two letters describe the direction and they could be single directions such as north, south, east and west, or they could be diagonal directions such as north-east, south-west and so forth. For north-bound wires, for example, the direction could be north (NL or NR), north-east (NE) or north-west (NW). The same is true for other directions.

The third character is a number describing the type of wire: 2 for double wires and 5 for pent wires. This is followed by three letters stating the position of this pin in the wire: beginning (BEG), middle (MID) and end (END). Finally, there is a wire index to differentiate between different wires coming out of or going into the same PSM.

An example is "NW5BEG1". This is the beginning pin of a pent wire going two hops in the north direction until the middle connection then goes west and there hops later connects its END pin. Its index is 1. This wire is illustrated in Fig. B.4.

B.2 Interconnect Illustrations

This section illustrates five types of interconnect wires found in the Virtex-5 FPGA: global, long, pent, double and bounceacross.

Global and long lines are bidirectional and can broadcast signals to multiple CLBs depending on the configuration. Pent, double and bounceacross wires are unidirectional. Pent and double lines span five and two CLBs respectively. This distance is the Manhattan distance from source to sink and they can be in any direction. There is additionally an intermediate middle connection of distance 2 and 1 for pent and double wires respectively. A connection can either be established from the beginning (BEG) terminal to this middle (MID) connection or to the final (END) connection.

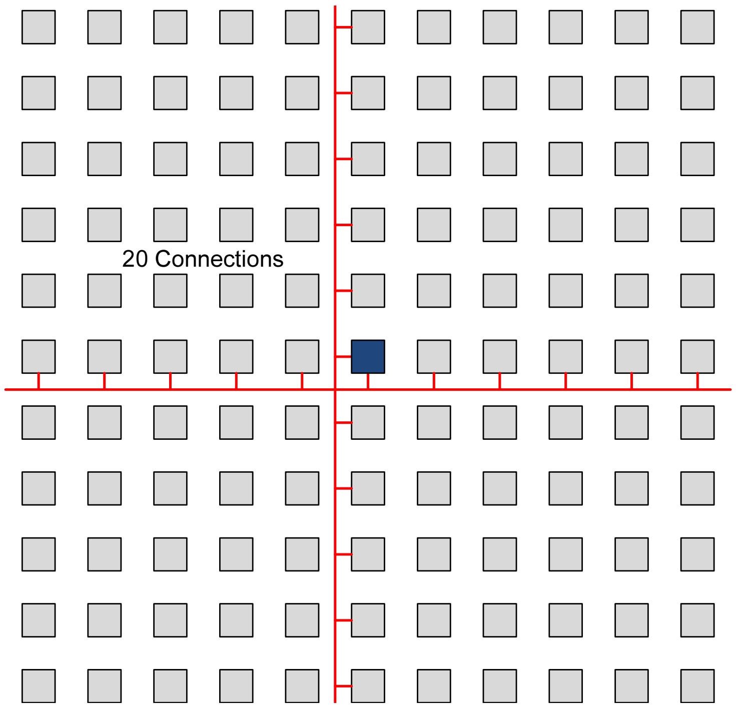

Figure B.1: Virtex-5 global wires

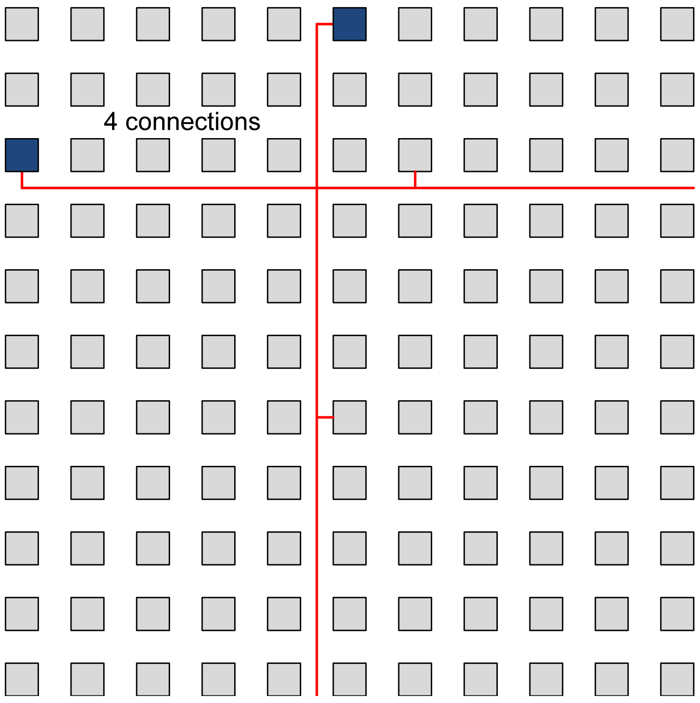

Figure B.2: Virtex-5 long wires

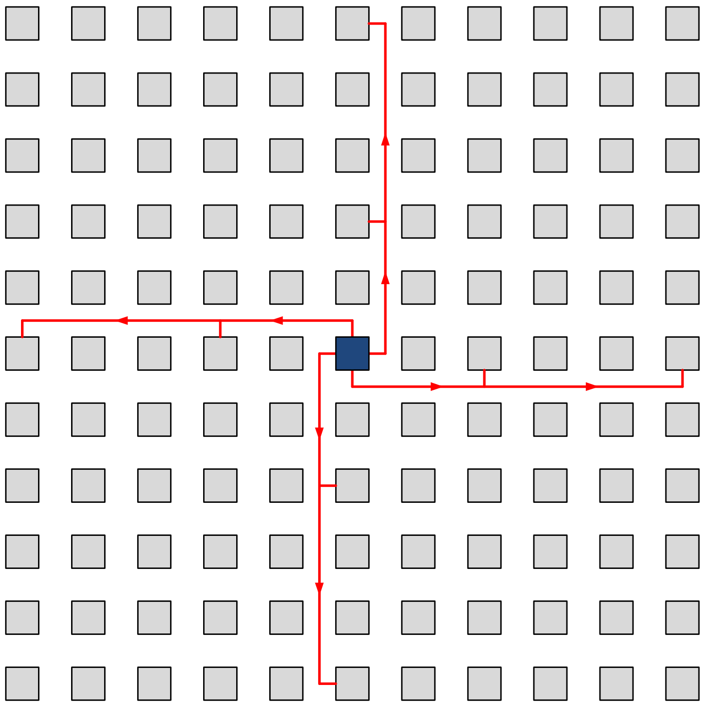

Figure B.3: Virtex-5 pent wires

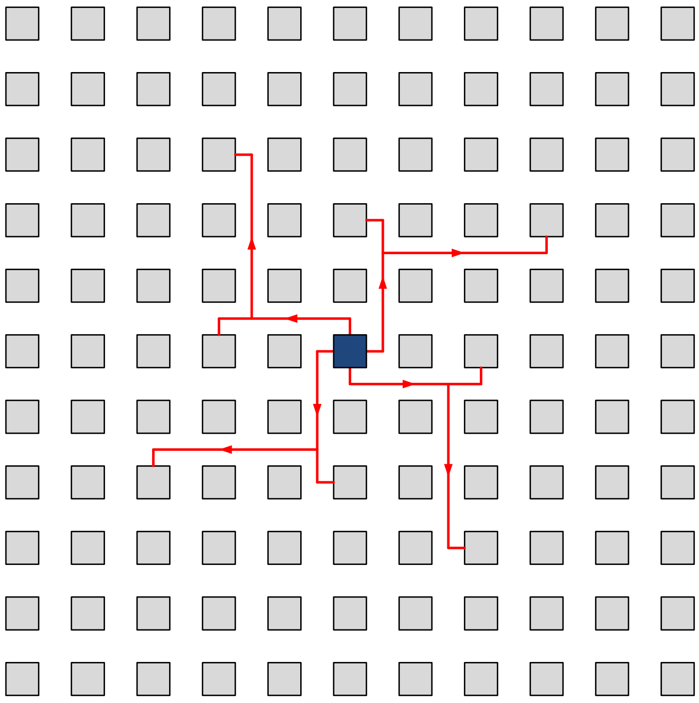

Figure B.4: Virtex-5 pent wires in diagonal connections

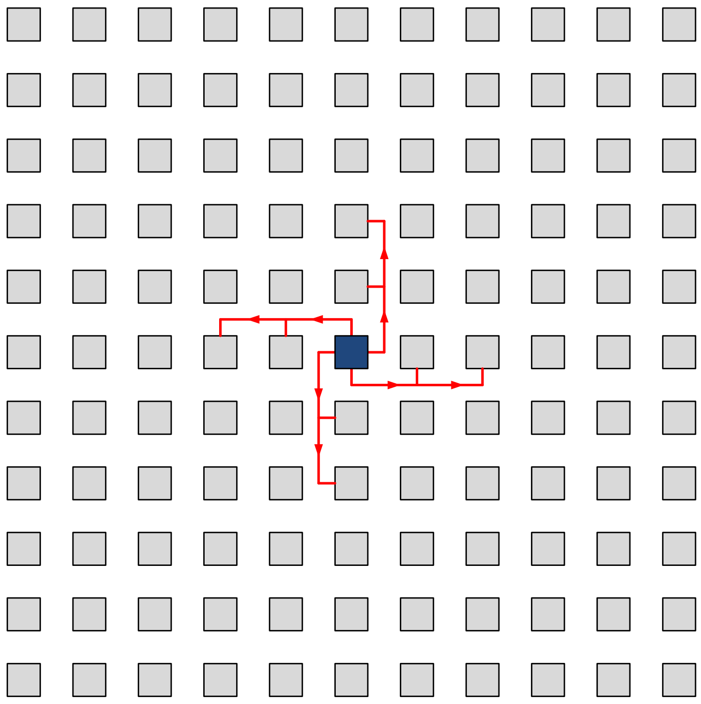
Figure B.5: Virtex-5 double wires

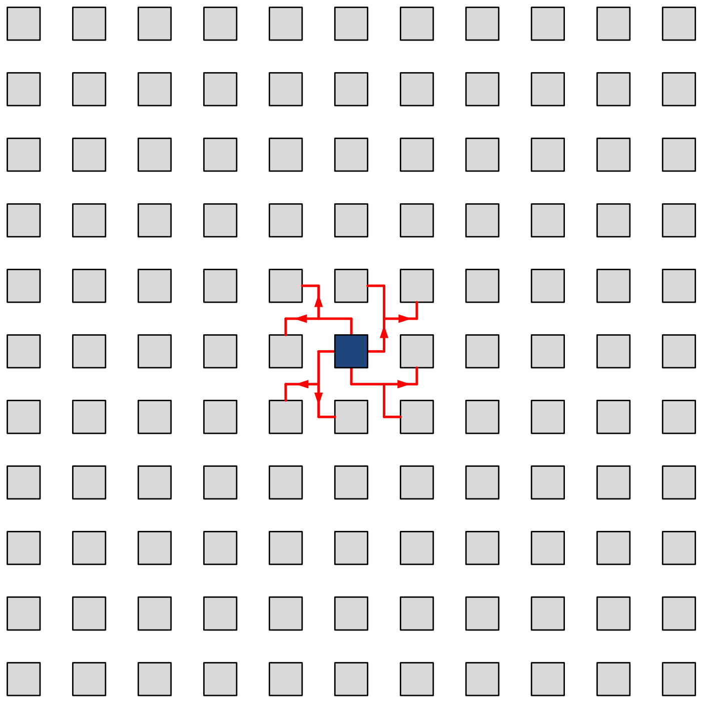
Figure B.6: Virtex-5 double wires in diagonal connections

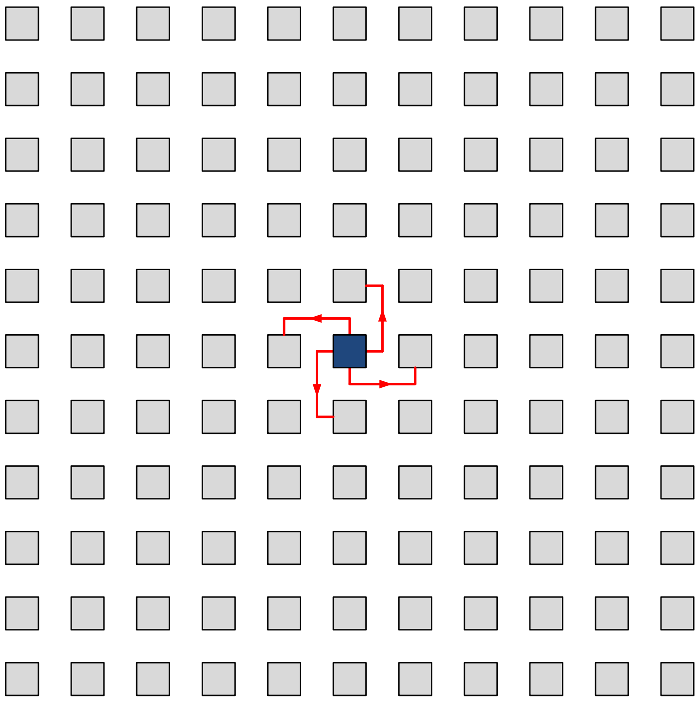

Figure B.7: Virtex-5 double wires

Bibliography

[1] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, “KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture,” in *Design, Automation and Test in Europe (DATE)*, Mar. 2010, pp. 819 –824. (Cited on pages 1 and 73.)

[2] M. Bushnell and V. Agrawal, Eds., *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*, ser. Frontiers in Electronic Testing. New York, NY, USA: Kluwer Academic Publishers, 2002, vol. 17. (Cited on pages 2 and 51.)

[3] V. Betz, J. Rose, and A. Marquardt, Eds., *Architecture and CAD for Deep-Submicron FPGAs.* Norwell, MA, USA: Kluwer Academic Publishers, 1999. (Cited on pages 2, 6, 7 and 8.)

[4] J. Yao, B. Dixon, C. Stroud, and V. Nelson, “System-level Built-In Self-Test of global routing resources in Virtex-4 FPGAs,” in *Southeastern Symposium on System Theory (SSST)*, Mar. 2009, pp. 29 –32. (Cited on pages 8, 25, 27, 28 and 64.)

[5] “Virtex-5 FPGA User Guide (UG190),” Xilinx, Inc., May 2010. (Cited on pages xi, 8, 9, 10, 11, 25, 27, 35, 37 and 40.)

[6] “PlanAhead Software Tutorial, Overview of the Partial Reconfiguration Flow (UG743),” Xilinx, Inc., Mar. 2011. (Cited on page 8.)

[7] “Configuration and Readback of Virtex FPGAs Using the JTAG Boundary-Scan (XAPP139),” Xilinx, Inc., Feb. 2007. (Cited on pages 8 and 22.)

[8] “Virtex FPGA Series Configuration and Readback (XAPP138),” Xilinx, Inc., Mar. 2005. (Cited on pages 8 and 23.)

[9] “Virtex-5 FPGA Configuration User Guide (UG191),” Xilinx, Inc., Aug. 2010. (Cited on page 8.)

[10] W. L. Huang, F. Meyer, and F. Lombardi, “Multiple fault detection in logic resources of FPGAs,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Oct. 1997, pp. 186 –194. (Cited on pages 15, 16 and 17.)

[11] W. Huang, M. Zhang, F. Meyer, and F. Lombardi, “A XOR-tree based technique for constant testability of configurable FPGAs,” in *Sixth Asian Test Symposium (ATS)*, Nov. 1997, pp. 248 –253. (Cited on pages 15, 16 and 17.)

[12] C. Stroud, S. Konala, P. Chen, and M. Abramovici, “Built-in self-test of logic blocks in FPGAs (Finally, a free lunch: BIST without overhead!),” in *VLSI Test Symposium*, May 1996, pp. 387 –392. (Cited on pages 15, 16, 17 and 22.)

[13] M. Renovell, “SRAM-based FPGAs: a structural test approach,” in *XI Brazilian Symposium on Integrated Circuit Design*, Oct. 1998, pp. 67 –72. (Cited on pages 15, 17, 18, 19, 22, 33 and 50.)

[14] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “Test pattern and test configuration generation methodology for the logic of RAM-based FPGA,” in *Sixth Asian Test Symposium (ATS)*, Nov. 1997, pp. 254 –259. (Cited on pages 15, 17 and 18.)

[15] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “SRAM-based FPGA’s: testing the LUT/RAM modules,” in *International Test Conference (ITC)*, Oct. 1998, pp. 1102 –1111. (Cited on pages 15, 17, 18 and 19.)

[16] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “RAM-based FPGAs: a test approach for the configurable logic,” in *Design, Automation and Test in Europe (DATE)*, Feb. 1998, pp. 82 –88. (Cited on pages 15, 17, 18 and 19.)

[17] M. Renovell and Y. Zorian, “Different experiments in test generation for XILINX FPGAs,” in *International Test Conference (ITC)*, 2000, pp. 854 –862. (Cited on pages 15 and 17.)

[18] C. Stroud, E. Lee, S. Konala, and M. Abramovici, “Using ILA testing for BIST in FPGAs,” in *International Test Conference (ITC)*, Oct. 1996, pp. 68 –75. (Cited on pages 15 and 21.)

[19] E. Atoofian and Z. Navabi, “A BIST architecture for FPGA look up table testing reduces reconfigurations,” in *Twelfth Asian Test Symposium (ATS)*, Nov. 2003, pp. 84 – 89. (Cited on pages 15, 21 and 22.)

[20] C. Metra, G. Mojoli, S. Pastore, D. Salvi, and G. Sechi, “Novel technique for testing FPGAs,” in *Design, Automation and Test in Europe (DATE)*, Feb. 1998, pp. 89 –94. (Cited on pages 15, 21 and 46.)

[21] W. K. Huang, F. Meyer, X.-T. Chen, and F. Lombardi, “Testing configurable LUT-based FPGA’s,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 276 –283, Jun. 1998. (Cited on pages 15, 19, 20 and 21.)

[22] S. Dhingra, “Built-in Self-Test of Logic Resources in Field Programmable Gate Arrays using Partial Reconfiguration,” Master’s thesis, Auburn University, Aug. 2006. (Cited on pages 15 and 23.)

[23] Y. Liao, P. Li, A. Ruan, W. Li, and W. Li, “Full coverage manufacturing testing for SRAM-based FPGA,” in *International Symposium on Integrated Circuits (ISIC)*, Dec. 2009, pp. 478 –481. (Cited on page 19.)

[24] A. Friedman, “Easily Testable Iterative Systems,” *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1061 – 1064, Dec. 1973. (Cited on pages 21, 46 and 47.)

[25] J. Yao, “Built-in Self-Test of Global Routing Resources in Virtex-4 FPGAs,” Master’s thesis, Auburn University, Aug. 2009. (Cited on pages 23, 27 and 28.)

[26] B. Dixon, “Built-in Self-Test of Programmable Interconnect in Field Programmable Gate Arrays,” Master’s thesis, Auburn University, Dec. 2008. (Cited on pages 23, 27 and 28.)

[27] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “Test configuration minimization for the logic cells of SRAM-based FPGAs: a case study,” in *European Test Workshop*, 1999, pp. 146 –151. (Cited on pages 23, 24, 49 and 54.)

[28] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “Testing the interconnect of RAM-based FPGAs,” *IEEE Design Test of Computers*, vol. 15, no. 1, pp. 45 –50, Jan.-Mar. 1998. (Cited on pages 25, 26 and 27.)

[29] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “SRAM-based FPGA’s: testing the interconnect/logic interface,” in *Seventh Asian Test Symposium (ATS)*, Dec. 1998, pp. 266 –271. (Cited on pages 25 and 26.)

[30] M. Renovell, J. Portal, J. Figueras, and Y. Zorian, “Testing the configurable interconnect/logic interface of SRAM-based FPGA’s,” in *Design, Automation and Test in Europe (DATE)*, 1999, pp. 618 –622. (Cited on pages 25 and 26.)

[31] C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici, “Built-in self-test of FPGA interconnect,” in *International Test Conference (ITC)*, Oct. 1998, pp. 404 –411. (Cited on pages 25, 26 and 27.)

[32] S. McCracken and Z. Zilic, “FPGA test time reduction through a novel interconnect testing scheme,” in *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, ser. FPGA ’02. New York, NY, USA: ACM, 2002, pp. 136–144. [Online]. Available: http://doi.acm.org/10.1145/503048.503069 (Cited on pages 25, 26 and 27.)

[33] S.-J. Wang and C.-N. Huang, “Testing and diagnosis of interconnect structures in FPGAs,” in *Seventh Asian Test Symposium (ATS)*, Dec. 1998, pp. 283 –287. (Cited on pages 25, 26 and 27.)

[34] M. Niamat, A. Sahni, and M. Jamali, “A built in self test scheme for automatic interconnect fault diagnosis in multiple and single FPGA systems,” in *Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2007, pp. 229 –232. (Cited on pages 25, 26 and 27.)

[35] D. Fernandes and I. Harris, “Application of built in self-test for interconnect testing of FPGAs,” in *International Test Conference (ITC)*, vol. 1, Oct. 2003, pp. 1248 – 1257. (Cited on pages 28, 29 and 30.)

[36] M. Tahoori and S. Mitra, “Application-independent testing of FPGA interconnects,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1774 – 1783, Nov. 2005. (Cited on pages 29 and 30.)

[37] C. Giasson and X. Sun, “Modeling the interconnects of Xilinx Virtex FPGAs and derivation of their test configurations,” in *Canadian Conference on Electrical and Computer Engineering*, vol. 2, May 2004, pp. 831 – 834 Vol.2. (Cited on pages 30 and 31.)

[38] W. H. Kautz, “Testing for faults in combinational cellular logic arrays,” in *Proceedings of the 8th Annual Symposium on Switching and Automata Theory (SWAT)*, ser. FOCS ’67. Washington, DC, USA: IEEE Computer Society, 1967, pp. 161–174. [Online]. Available: http://dx.doi.org/10.1109/FOCS.1967.33 (Cited on pages 33, 46 and 47.)

[39] M. Psarakis, D. Gizopoulos, and A. Paschalis, “Test Generation and Fault Simulation for Cell Fault Model using Stuck-at Fault Model based Test Tools,” *Journal of Electronic Testing*, vol. 13, pp. 315–319, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1023/A:1008389920806 (Cited on pages 33 and 34.)

[40] E. Bareiša, V. Jusas, K. Motiejūnas, and R. Šeinauskas, “BLACK BOX FAULT MODELS,” *Information Technology and Control*, vol. 35, pp. 177–186, 2006. (Cited on page 33.)

[41] E. Bareiša, V. Jusas, K. Motiejūnas, and R. Šeinauskas, “The Realization-Independent Testing Based on the Black Box Fault Models,” *Informatica*, vol. 16, pp. 19–36, Jan. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1413769.1413771 (Cited on page 33.)

[42] A. Van De Goor, “Using march tests to test SRAMs,” *IEEE Design Test of Computers*, vol. 10, no. 1, pp. 8 –14, Mar. 1993. (Cited on pages 36, 37 and 51.)

[43] S. Makar and E. McCluskey, “Functional tests for scan chain latches,” in *International Test Conference (ITC)*, Oct. 1995, pp. 606 –615. (Cited on pages 50 and 54.)

[44] J.-B. Note and E. Rannaud, “From the bitstream to the netlist,” in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, ser. FPGA ’08. New York, NY, USA: ACM, 2008, pp. 264–264. [Online]. Available: http://doi.acm.org/10.1145/1344671.1344729 (Cited on page 67.)

[45] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, “Rapid prototyping tools for FPGA designs: RapidSmith,” in *International Conference on Field-Programmable Technology (FPT)*, Dec. 2010, pp. 353 –356. (Cited on page 68.)

[46] "FPGA Editor Software Manual v13.1," Xilinx, Inc., Jan. 2011. (Cited on page 73.)

Declaration - Erklärung

Declaration

This is to certify that:

i. The thesis comprises only my original work towards the master degree

ii. Due acknowledgment has been made in the text to all other material used

Mohamed Abdelfattah
31 August 2011

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Mohamed Abdelfattah
31 August 2011